The Abstract State Machines Method for High-Level System Design and Analysis

Egon Börger

Dipartimento di Informatica, Universita di Pisa, Italy boerger@di.unipi.it

1 Introduction

In this paper we give an answer to the often asked question what characterizes the Abstract State Machines (ASM) method among the practical and scientifically well-founded systems engineering methods. The question is justified since the ASM method, which has been developed during the 1990'ies (see [23] for a historical account), is a latecomer among other well-known rigorous system design and analysis methods, including what misleadingly is called "formal" methods. For answering the question, we assume the reader to have some basic knowledge of what formal methods are and what they intend to achieve, but we sketch the major ingredients of the ASM method.[1]

We do not speak about special-purpose techniques, like static analysis, model checking etc., which draw their success from being tailored to particular types of problems. The discussion is focussed on *wide-spectrum methods*, which assist system engineers in every aspect of an effectively controllable construction of reliable computer-based systems. In particular such methods have to bridge the gap between the two ends of system development:

- the human understanding and formulation of real-world problems,
- the deployment of their algorithmic solutions by code-executing machines on changing platforms.

The activities these development disciplines have to support cover the wide range from requirements capture and analysis to writing executable code, including verification, validation (testing), documentation and maintenance (change management). As a consequence a scientifically rigorous approach, which enhances best engineering practice by adding mathematical rigour to it, calls for a smooth integration, into traditional hw/sw-system engineering

[1] However, this paper is neither an introduction to the ASM method nor a survey of its achievements. For the former see [28], for the latter [26], or chapters 2 and 9 of the AsmBook [45].

procedures and notations, of multiple ways to achieve various degrees of certifiable system trustworthiness and quality assurance.

Among such approaches the ASM method is characterized by providing a *simple practical framework, where in a coherent and uniform way the system engineer can* adopt a divide-and-conquer approach, i.e.

- *systematically separate multiple concerns, concepts and techniques*, which are inherent in the large variety of system development activities,
- *freely choose for each task an appropriate combination of concepts and techniques* from the stock of engineering (including formal) methods, at the given level of abstraction and precision where the task occurs.

As will become clear in the following sections, not a single ingredient of the ASM method is original. What is unique is the simplicity of the method and the freedom it offers the practitioner to choose for each problem an appropriate combination of concepts, notations and techniques, which are integrated by the framework in a coherent way as elements of a uniform mathematical background. Among the examples we will discuss are the following:

- abstract states, which can be richly structured, possibly unbounded or even infinite, as known from the theory of abstract data types and algebraic specifications [85, 78, 61, 9, 10], VDM [66], Z [103], COLD [65],
- abstract instructions for changing states (high-level operational definition of state changes by guarded assingments), as familiar from pseudo-code notation, Virtual Machines[2] and later RAISE [73],
- synchronous parallel execution model, including conditional multiple assignments as present also in UNITY [86] and COLD [64],
- locality principle as known from programming languages,
- functional definitions, as in mathematics and functional programming,
- declarative (axiomatic) definitions, as known from logic and declarative programming and specification languages,
- refinement concept, generalizing the method which has been introduced by Wirth [112] and Dijkstra [58] and adapted to numerous formal specification methods [11, 89, 12, 53], including Z [113, 57] and B [1],
- decomposition and hierarchy concepts, as familiar from automata theory and layered architectures,
- function classification into monitored, controlled, shared etc., as known from programming and Parnas' SCR method [94, 80],
- verification of model properties by proofs at the needed level of precision: sketched, detailed, machine assisted (interactive or fully automated),
- simulation by model execution, e.g. for model checking invariants, runtime verification of properties, testing of runs (scenarios).

The *combined separation and integration capabilities of the ASM framework*, which allow the engineer to tailor his methods to the problem under in-

[2] For example in Cremers' and Hibbard's *data spaces* [52] the operational transformations of abstract states are described by means of static functions, which form what is called there an *information structure*.

vestigation, are responsible for the successful applications of the ASM method in a variety of academic and industrial projects. They range from the design and analysis (read: verification and validation) of programming languages, computer architectures, protocols and web services to the design, reengineering and validation of industrial control systems and the definition of industrial standards. Some examples are highlighted in Sect. 7.

Among the different development and analysis tasks, which can be coherently linked together in the ASM framework, we discuss the following ones:

- *Design*. The design activities split into three major groups:
  - *Ground model construction*, i.e. definition of a system blueprint that can be justified to correctly capture the requirements. This is supported by the ASM ground model technique explained in Sect. 4.
  - *Model refinement*, reflecting one by one the various design decisions, which lead from the ground model to code. A rigorously controllable discipline of stepwise adding implementation details is supported by the ASM refinement method explained in Sect. 5.
  - *Model change*, a combination of the ground model construction and refinement task, which uses the hierarchy of models constructed during the transformation of the ground model into code. When change requirements occur, this hierarchy is analyzed to determine the level starting from where new refinements are needed to traceably incorporate the requested changes. Changing the models may trigger also new analysis tasks.
- *Analysis*. The goal of the analysis activities is to provide a documentation and justification for the steps that lead from the requirements to the ground model and its implementation. This is needed for two purposes: a) an evaluation (read: explanation, verification, validation) of each design decision by repeatable procedures, b) change management and reuse of models. The analysis activities split into two major groups:
  - *mathematical verification* of system properties, by a variety of reasoning techniques, applicable to system models at different levels of precision and under various assumptions, e.g.
    - outline of a proof idea or proof sketch
    - mathematical proof in the traditional meaning of the term
    - formalized proof within a particular logic calculus
    - computer-checked (automated or interactive) proof
  - *experimental validation* of system behaviour through simulation and testing of rigorous models, at various levels of abstraction, like system test, module test, unit test, simulation of ground model scenarios, etc.

The core of the ASM method is based upon the following three concepts we are going to explain in the following sections, starting with an illustration by three simple examples in Sect. 2.

- *Notion of ASM*, a mathematically precise substitute for the intuitive notion of high-level algorithmic processes (including what software engineers call pseudo-code) and for the nowadays omnipresent concept of Virtual Machines (VMs). Technically ASMs can be defined as a natural generalization of Finite State Machines (FSMs) by extending FSM-states to Tarski structures. Tarski structures, also called first-order or simply mathematical structures, represent truly abstract data types. Therefore, extending the special domains of FSM-computations to these structures turns *finite* state machines into *abstract* state machines, which work over possibly richly structured yet abstract states, as is explained in Sect. 3.
- *ASM ground models* as accurate high-level descriptions of given system requirements. They are expressed at a level of abstraction that is determined by the application domain and provide a requirements documentation that is to be used as authoritative reference for an objective evaluation of the requirements and the following further system development activities. The ground model must be kept synchronized with those activities, namely:[3]
  - *detailed design*,
  - *design evaluation* and *quality assurance* via analysis, including testing and an inspection and review process, focussed on certifying the *consistency, correctness* and *completeness* properties of the system that are needed to guarantee the desired degree of reliability,[4]
  - *system maintenance*, including requirements change management.

  In Sect. 4 we discuss the ASM ground model method further.
- *ASM refinements*, linking the more and more detailed descriptions at the successive stages of the system development cycle in an organic and effectively maintainable chain of rigorous and coherent system models. The refinement links serve the purpose to guarantee that the system properties of interest are preserved in going from the ground model via a series of design decisions to its implementation by the code—and to document this fact for possible reuse during maintenance and in particular for change management. We discuss the concept of ASM refinement further in Sect. 5.

The simple mathematical foundation of ASMs as FSMs working over arbitrary data types makes it easy for practitioners to understand and work with the concept. It also allows one to exploit for the ASM method the *uniform conceptual and methodological framework* of traditional mathematics,

[3] The definition of the ground model may change during the design phase, namely if it is recognized during the implementation process that some important feature is missing in the ground model or has to be changed there. The *process* of building a ground model is iterative; it ends only with the completion of the design and may be re-opened during maintenance for change management. But at each moment of the development process, there is one ground model, documenting the current understanding of the problem the system has to solve.

[4] Note that the evaluation of the design against the ground model also provides an objective, rational ground for settling disputes on the code after its completion.

where one can consistently relate standard notions, techniques and notations to express any system features or views.[5] Having as background for the ASM method not just one *a priori* chosen formal language and associated proof calculus, but the full body of usual mathematical notations and techniques "supports a *rigorous integration of common design, analysis and documentation techniques* for model reuse (by instantiating or modifying the abstractions), validation (by simulation and high-level testing), verification (by human or machine-supported reasoning), implementation and maintenance (by structured documentation)"[45, pg.1]. We discuss this in Sect. 6 and illustrate it there by a characterization of Event-B Machines as a family of specialized ASMs. In Sect. 7 we point to some application highlights of the ASM method.

2 Illustration by Examples

We illustrate here ASMs by three simple examples for a) the *construction of ASM ground models*, which can be shown to capture the requirements in application problem terms, b) their *refinements*, which can be proven to correctly reflect both b1) the implementation details and b2) the changes in the models when changes in the requirements come along. The examples are taken from [82], a book which explains very well the various descriptions one has to make and to fit together into a correctness argument, in order to show that under certain assumptions on the environment—typically reflecting the relevant domain knowledge—the behavior of the specification satisfies the requirements. What we call a ground model is a *closed* model. It includes both the specification and the statement of the environmental assumptions and domain knowledge that are needed in a correctness argument.

2.1 Sluice Gate Control

The following problem description is taken from [82, p.49], the italics are ours.

> A small sluice, with a rising and falling gate, is used in a simple irrigation system. A computer system is needed to control the sluice gate: the *requirement* is that the gate should be held in the fully open position for ten minutes in every three hours and otherwise kept in the fully closed position.
> The gate is opened and closed by rotating vertical screws. The screws are driven by a small *motor*, which can be controlled by clockwise, anticlockwise, on and off pulses.

[5] Thus the ASM method satisfies Parnas' request [93] to base the foundation for a reliable software engineering discipline on standard mathematics, avoiding the introduction of any new specification language or new theory of language semantics.

There are *sensors* at the top and bottom of the gate travel; at the top it's fully open, at the bottom it's fully shut.
The *connection to the computer* consists of four pulse lines for motor control and two status lines for the gate sensors.

Ground Model. To simplify the correctness argument to be provided for the ground model, we stick to first modeling only the *user requirements* for the equipment, abstracting from the details about the screws, the motor, the sensors and the pulses. This reduces the system to an abstract device which switches from a *fullyClosed* phase to a *fullyOpen* phase whenever the time *closedPeriod* has elapsed, and back when *openPeriod* has elapsed. To separate the issues related to (an implementation of) the timing model from the analysis of the user requirements, we use two so-called *monitored locations* (read: array or instance variables) $Passed(openPeriod)$, $Passed(closedPeriod)$. Their truth values are assumed to be controlled correctly by the environment and to indicate when the intended time periods have passed, here $openPeriod = 10\ min$ for *fullyOpen* and $closedPeriod = 3\ hrs - 10\ min$ for *fullyClosed*. We interpret the term 'for ten minutes in every three hours' as 'at the end of the closure period' and being included in the total $period = 3\ hrs$. This leads to the model in Fig. 1, which is displayed in the usual FSM-style graphical notation using circles for phases (also called control states or internal states), rhombs for test predicates (also called guards) and rectangles for actions of submachines (for a definition of these control state ASMs see Sect. 3). Due to the abstraction from the motor and the sensors, the submachines to OPEN respectively SHUT the gate do nothing and are included only to hold the place for the refinement by motor actions. Assuming appropriate

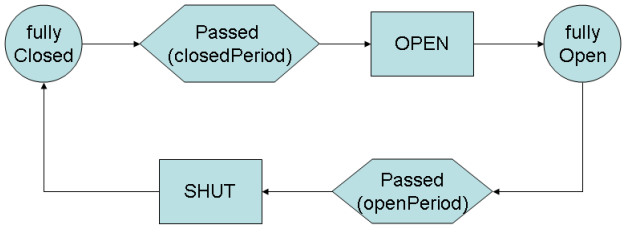

Fig. 1. SLUICEGATEGROUND Model

conditions on initial states, this abstract machine SLUICEGATEGROUND can clearly be justified, in terms of the gate being kept open (read: the device being in state *fullyOpen*) for *openPeriod* and closed (read: the device being in state *fullyClosed*) for *closedPeriod*, to correctly and rigorously reflect the above stated requirement.

A refinement step. The first refinement step reflects the domain knowledge about a screws driving *motor* and the sensors. We know that the motor can be

set *on* and *off* and has two move *dir*ection values *clockwise* (say to raise the gate) and *anticlockwise* (say to lower the gate). It is in these terms, namely of two so-called *controlled locations* *motor*, *dir* which can be updated to any of their values in $\{on, off\}$ respectively $\{clockwise, anticlockwise\}$, that the submachines OPEN and SHUT are refined by using three motor action submachines STARTTORAISE, STARTTOLOWER, STOPMOTOR. The control of these actions uses the environmental gate status information that is obtained from the two sensors. The indication by the sensors that the gate travel has reached its top (fully open) respectively bottom (fully closed) position is formalized by two monitored locations $Event(Top)$ respectively $Event(Bottom)$ taking boolean values. The time assumed for the execution of the new submachines is taken into account by refining the definition of *closedPeriod* and *openPeriod*. These two locations are examples of what we call *derived locations*, since their value is defined in a fixed manner (here by an equation) in terms of the values of other locations. This leads to the refinement SLUICEGATEMOTORCTL of SLUICEGATEGROUND as defined in Fig. 2, together with the following definition of abstract motor actions:[6]

$$\begin{aligned}
&\text{STARTTORAISE} = dir := clockwise \qquad \text{STOPMOTOR} = (motor := off)\\
&\qquad\qquad\quad motor := on\\
&\text{STARTTOLOWER} = dir := anticlockwise\\
&\qquad\qquad\quad motor := on\\
&closedPeriod = period\\
&\quad -(StartToRaiseTime + OpeningTime + StopMotorTime)\\
&\quad -(StartToLowerTime + ClosingTime + StopMotorTime)
\end{aligned}$$

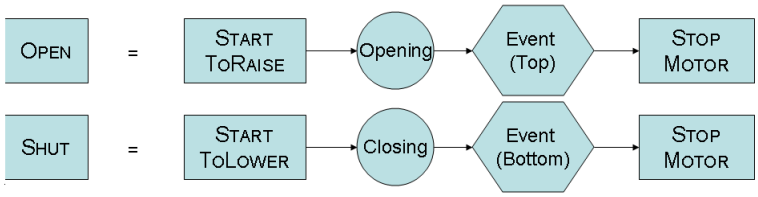

Fig. 2. SLUICEGATEMOTORCTL refinements of motor actions

The correctness proof for this refinement uses the following *Input Locations Assumption*, which relates what happens at the environmental sensors to the model events:

- When the *top* respectively *bottom* of the gate travel is detected by the corresponding sensor, $Event(Top)$ respectively $Event(Bottom)$ becomes true in SLUICEGATEMOTORCTL.

[6] In general, in an ASM all updates are executed in parallel, though in this example also a sequential reading will do.

Another refinement step. Here we introduce the four status lines connecting the controller and the physical equipment. The single abstract machine SluiceGateMotorCtl is replaced by a so-called multi-agent ASM SluiceGate consisting of two abstract machines, an environmental machine Pulses describing the equipment actions when pulses appear and a software machine SluiceGateCtl. In the definition of Pulses we use the notation **upon** *Event* **do** *Action* for **if** *Event* **then** *Action*.

Pulses = **upon** *Event*(*Clockwise*) **do** *dir* := *clockwise*
upon *Event*(*AntiClockwise*) **do** *dir* := *anticlockwise*
upon *Event*(*MotorOn*) **do** *motor* := *on*
upon *Event*(*MotorOff*) **do** *motor* := *off*

SluiceGateCtl is the same as SluiceGateMotorCtl except for a refined submachine StartToRaise, which has to Emit(*Pulse*(*Clockwise*)) and Emit(*Pulse*(*MotorOn*)); similarly for StartToLower, StopMotor.

SluiceGateCtl = SluiceGateMotorCtl **where**
StartToRaise = Emit(*Pulse*(*Clockwise*))
Emit(*Pulse*(*MotorOn*))
StartToLower = Emit(*Pulse*(*AntiClockwise*))
Emit(*Pulse*(*MotorOn*))
StopMotor = Emit(*Pulse*(*MotorOff*))

The correctness proof for this refinement step, which relates runs of the abstract and the refined machine, relies upon the following assumptions:

- *Pulse Output Assumption*: each Emit(*Pulse*(*p*)) yields *Event*(*p*) to happen in the environment.
- *Input Locations Assumption*, reinterpreted to reflect that the information detected by the sensors arrives at SluiceGateCtl as input *Event*(*Top*), *Event*(*Bottom*) via two status lines.

We also use the usual convention that events are consumed when they trigger a rule to be fired. Note that the refinement type is $(1, 2)$, meaning that every segment consisting of one step of SluiceGateMotorCtl is refined by a segment of two corresponding steps of SluiceGate, namely of one step of the software machine followed by one step of the environment machine.

Admittedly this is an elementary example. The very same technique has been applied in [40, 88] to successively refine an ASM ground model for a robot controller to a provably correct C++ control program.

2.2 One-Way Traffic Light Control

This example is about one-way traffic control:

> ...the traffic is controlled by a pair of simple portable traffic light units...one unit at each end of the one-way section...connect(ed)...to a small computer that controls the sequence of lights.

Each unit has a Stop light and a Go light.
The computer controls the lights by emitting RPulses and GPulses, to which the units respond by turning the light on and off.
The regime for the lights repeats a fixed cycle of four phases. First, for 50 seconds, both units show Stop; then, for 120 seconds, one unit shows Stop and the other Go; then for 50 seconds both show Stop again; then for 120 seconds the unit that previously showed Go shows Stop, and the other shows Go. Then the cycle is repeated.

Ground Model. From the user perspective the problem is about two light units, each equipped with a $StopLight(i)$ and a $GoLight(i)$ $(i = 1, 2)$ which can be set *on* and *off*. In the ground model we treat the latter as controlled locations to which a value *on* or *off* can be assigned directly, abstracting from the computer emitting pulses. We also abstract from an explicit time computation and treat the passing of time by monitored locations $Passed(timer(phase))$, where the function *timer* defines the requested light regime. The monitored locations are assumed to become true in the model whenever (and only when) $timer(phase)$ has elapsed in the environment since the current *phase* was entered.

For definiteness let us assume that the sequence of lights starts with both $StopLight(i) = on$ and both $GoLight(i) = off$. Let us call this phase $Stop1Stop2$. After $timer(Stop1Stop2)$ has passed, the control executes a submachine SwitchToGo2 and then enters phase $Go2Stop1$ (say), followed upon $Passed(timer(Go2Stop1))$ becoming true by a SwitchToStop2 to enter phase $Stop2Stop1$, then a SwitchToGo1 to enter phase $Go1Stop2$ and finally a SwitchToStop1 to return to phase $Stop1Stop2$. This behavior of the equipment is rigorously expressed by the sequence of four phase changing 'ASM rules' in Fig. 3, defined again in FSM-like notation and using the submachine macros defined below.

The four control states correspond to the required combinations of 'showing' Goi and $Stopj$, reflecting that in the above requirements description the values of $StopLight(i)$, $GoLight(i)$ appear to be complementary:

$Stop\ i$ means $StopLight(i) = on \wedge GoLight(i) = off$
$Go\ i$ means $GoLight(i) = on \wedge StopLight(i) = off$

The complementarity of $StopLight(i)$, $GoLight(i)$ values implies that switching them can be done in parallel. Thus the two submachines SwitchToGo, SwitchToStop are copies of one machine (which only later will be refined to different instantiations introducing a sequentialization, see below):

$$\begin{aligned} \text{SwitchToGo}i = \text{SwitchToStop}i = {} & \text{Switch}(GoLight(i)) \\ & \text{Switch}(StopLight(i)) \end{aligned}$$

where $\text{Switch}(Light) = (Light := Light')$ ($'$ for complement)

The light regime (50,120,50,120) associates to each *phase* its length (in terms of some time measurement), represented by function values $timer(phase)$. Following the requirements, for this ground model the function is assumed to be

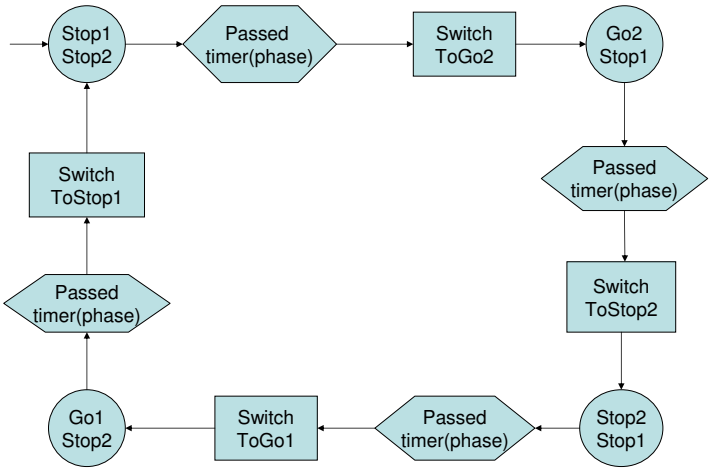

Fig. 3. 1WayTrafLightGround Model

static (set before running the machine). A change request to include the possibility to configure the time intervals associated to the phases would make it dynamic and controlled by the configuration machine.

$$\begin{aligned} timer(phase) = \textbf{case } phase \textbf{ of } & Stop1Stop2 : 50sec \\ & Go2Stop1 : 120sec \\ & Stop2Stop1 : 50sec \\ & Go1Stop2 : 120sec \end{aligned}$$

With these definitions and assumptions, 1WayTrafLightGround can be justified in application domain terms to realize the desired cyclic light sequence.

A refinement step. We used for the ground model a mixed behavioral and declarative instead of a purely declarative observational description to easen the correctness proofs for adding further details, which eventually should transform the abstract ground model into executable code. In a first refinement step we introduce the software interface feature that relates R/G Pulses of the computer to turning the light units on/off. As in the sluice gate control example, this refinement step replaces the single abstract machine 1WayTrafLightGround by a multi-agent ASM 1WayTrafLight consisting of an environmental pulse-triggered machine Pulses and a software

machine 1WayTrafLightCtl, obtained from 1WayTrafLightGround by refining the submachines SwitchTo...i to emitting pulses:

1WayTrafLightCtl = 1WayTrafLightGround **where**
  forall $i \in \{1,2\}$ SwitchTo...i = Emit($RPulse(i)$)
    Emit($GPulse(i)$)

Pulses = **forall** $i \in \{1,2\}$
  upon $Event(RPulse(i))$ **do** Switch($StopLight(i)$)
  upon $Event(GPulse(i))$ **do** Switch($GoLight(i)$)

The link between 1WayTrafLight and 1WayTrafLightGround is provided by the following *Pulse Output Assumption*, which relates the software actions to what happens in the environment:

- Emit($RPulse(i)$) yields $Event(RPulse(i))$ to happen in the environment
- Emit($GPulse(i)$) yields $Event(GPulse(i))$ to happen in the envirnoment

Using this assumption, it is easy to prove the refinement to be correct, observing that each software control step of the refined SwitchTo...i triggers an environment step of Pulses, which switches the corresponding lights. Thus one ground model step is correctly refined to two steps in the refined multi-agent machine ($(1,2)$-refinement).

Change requirement. We illustrate here two simple change requests, which lead to reusing the abstract machines defined above. The first one is:

> use simultaneous *Stop* and *Go* lights to indicate 'Stop, but be prepared to Go' [82, p.111]

To adapt the models to this request for change, it suffices to give up the view that the $StopLight(i)$, $GoLight(i)$ values are complementary to each other and to *refine* the SwitchToGoi submachines by a sequentialized version. Everything else is kept unchanged in both the above ground model and its refinement. With this instantiation of SwitchToGoi, it should be clear how to show that the new ground model correctly reflects the changed requirements and how to prove that it is correctly refined by the new refined model.[7]

SwitchToGoi_{grd} =
  $GoLight(i) := GoLight(i)'$ **seq** $StopLight(i) := StopLight(i)'$

SwitchToGoi_{ref} =
  Emit($GPulse(i)$) **seq** Emit($RPulse(i)$)

Another typical change request could be to add a simultaneous *Stop* and *Go* lights period (change time) of say 10 seconds. This comes up to refine SwitchToGoi from an atomic to a durative action, leaving everything else

[7] The definition of the **seq** constructor in the context of the parallel ASM execution model is defined in [44].

unchanged. A natural way to do this is to introduce an intermediate control state *WaitToGo* between the executions of SWITCH(*GoLight*(*i*)) and SWITCH(*StopLight*(*i*)), as indicated in Fig. 4. The new function *chgTime* indicates the length of the *WaitToGo* phase, in the example 10 seconds.

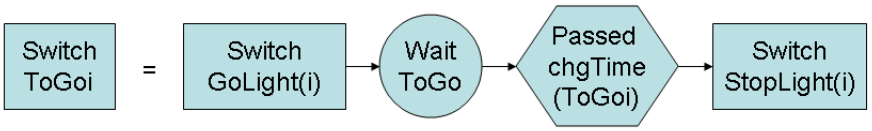

Fig. 4. Refining SWITCHTOGO by Change Time

Admittedly, these reuse examples are rather elementary, but the method is general. The reader who is interested in a more involved example may look at [37, 68, 70, 69] for a reuse for C# and .NET CLR of the ASM models built and analyzed in [106] for Java and the JVM.

2.3 Package Router Control

In this example we illustrate the use of a) the synchronous parallelism underlying the semantics of ASMs and b) an abstract event handling scheme. The problem is about the control of a package router to sort packages into bins according to their destinations and appeared in [81]. The formulation below is copied from [82, p.147][8].

> The packages carry bar-coded labels. They move along a conveyor to a reading station where their package-ids and destinations are read. They then slide by gravity down pipes fitted with sensors at top and bottom. The pipes are connected by two-position switches that the computer can flip (when no package is present between the incoming and outgoing pipes). At the leaves of the tree of pipes are destination bins, corresponding to the bar-coded destinations.
> A package cannot overtake another either in a pipe or in a switch. Also, the pipes are bent near the sensors so that the sensors are guaranteed to detect each package separately. However, packages slide at unpredicatable speeds, and may get too close together to allow a switch to be set correctly. A misrouted package may be routed to any bin, an appropriate message being displayed ...
> The problem is to build the controlling computer ... to route packages to their destination bins by setting the switches appropriately, and to report misrouted packages.

[8] For reasons of space we leave out the operator commands to stop or start the conveyor. Adding them would involve adding an operator machine.

From the layout illustration in Fig. 5 one can recognize the elements of the problem signature: a static tree structure whose nodes are decorated with elements from various sets, namely *PkgLabel*, *Pipe*, *Switch*, *Bin*. Each of these

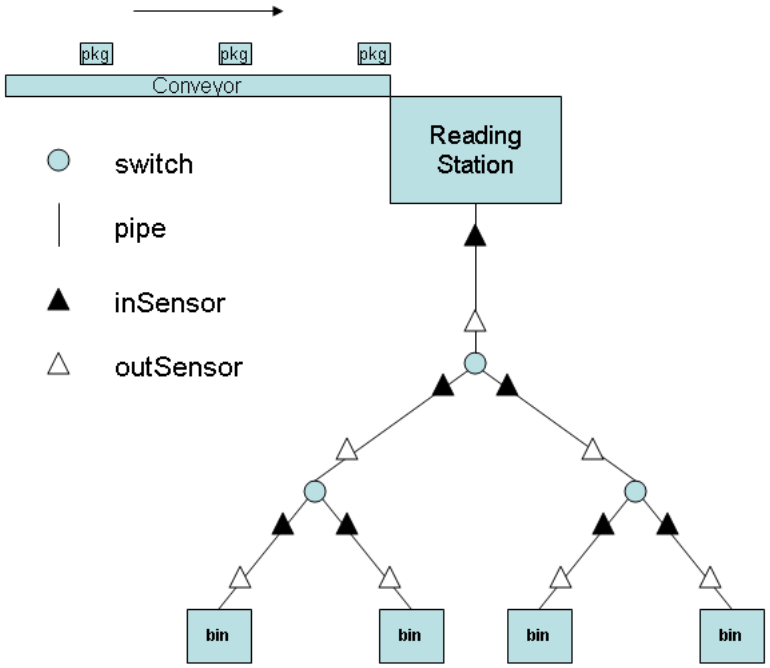

Fig. 5. Package Router Layout

domains is equipped with various functions. Assuming a unique association of bar-coded labels to packages, the reader decodes elements $l \in PkgLabel$ and stores the obtained information into locations $pkgId(l)$, $dest(l)$. To separate the decoding program from the control program, we consider these locations as static for the ground model.

Each $p \in Pipe$ comes with locations $inSensor(p)$ $outSensor(p)$ to signal package passing events. Elements $sw \in Switch$ come with a controlled location $pos(sw)$ indicating the current value of the switch position, *right* or *left*. To be able to correctly set $pos(sw)$ at runtime to reach a given bin b, a static function dir with values $dir(sw, b) \in \{left, right, none\}$ is needed indicating whether there is a path from sw to b and in the positive case where to direct sw for taking this path.

Bin comes with a static function associating to every $d \in Destination$ the corresponding $bin(d)$ where packages for that destination are requested to be routed. It is needed to define the derived predicate *MisroutedPkgInBin*, which

is defined to be true for a pair (l, b) if and only if the package identified by $pkgId(l)$ arrived in bin b, but $bin(dest(l)) \neq b$.

The static tree structure is defined by the root *reader* and *succ*essor locations that satisfy the following conditions:

$succ(reader) \in Pipe$
forall $p \in Pipe$ $succ(p) \in Switch \cup Bin$
forall $sw \in Switch$ $succ(left, sw), succ(right, sw) \in Pipe$

Using these locations we can define the following derived successor location for switches, which indicates its current, dynamically computed successor:

$$succ(sw) = succ(pos(sw), sw)$$

The information on no overtaking of packages can be formalized as first-in first-out behavior of packages, using three types of queues:

- $queue(reader)$ of labels of packages whose label has been read, but which did not yet enter the top pipe $succ(reader)$
- $queue(pipe)$ of labels of packages that entered the pipe at $inSensor(pipe)$ but did not yet exit it at $outSensor(pipe)$
- $queue(switch)$ of labels of packages that
  - entered into *switch* at its entry point $outSensor(pipe)$, where *pipe* is the predecessor of *switch*
  - did not yet exit at its end point $inSensor(succ(switch))$[9]

We define the control program ground model as a parallel composition of five submachines, each concerned with transfering a package (label) from one queue to the next one. The synchronous parallelism allows us to separate the routing functionality from the decision about the concrete scheduling mechanism needed for an implementation.

PackageRouter = IntoReader
FromReader2Pipe
FromPipe2Switch
FromSwitch2Pipe
FromPipe2Bin

Each of these submachines is triggered by an event, namely a package arriving at a sensor. It is represented by a predicate of the corresponding sensor location. For a succinct notation we adopt the convention that events are consumed by firing a rule guarded by this event, which saves us to repeat in each rule body the update through which the event predicate is reset to false. There are three types of events:

[9] This is well-defined since by assumption, *switch* can be flipped only when there is no package in it, i.e. $succ(switch)$ and thereby the exit point of *switch* can change only when $queue(switch)$ is empty.

- *Event*(*ArrPkgLab*(*l*)) becomes true when the reader has decoded the barcode of a package into its associated label *l*,
- for any *sensor*, *EventOn*(*sensor*) becomes true when the leading edge of a package arrives at *sensor*,
- for any *sensor*, *EventOff*(*sensor*) becomes true when the trailing edge of a package leaves the *sensor* behind.

The machine IntoReader simply enqueues a newly arrived package label.

```
IntoReader =
  if Event(ArrPkgLab(l)) then Enqueue(l, queue(reader))
```

Labels of packages arriving in a pipe, from the reader or from a switch, simply advance from the reader or switch queue to the queue of the pipe:

```
FromReader2Pipe =
  if EventOn(inSensor(succ(reader))) then AdvanceFrom(reader)
FromSwitch2Pipe
  forall sw ∈ Switch if EventOn(inSensor(succ(sw))) then
    AdvanceFrom(sw)
where
  AdvanceFrom(a) = Dequeue(queue(a))
                   Enqueue(fstout(queue(a)), queue(succ(a)))
```

When advancing labels of packages that leave a pipe to enter a switch, one also has to determine the correct position of that switch for that package to be routed correctly (if still possible). The correctness of the positioning submachine comes from the definition of the static function *dir* and the assumption in the requirements that a switch can be flipped only when no package is present between its incoming and outgoing pipes.

```
FromPipe2Switch = forall p ∈ Pipe
  if succ(p) ∈ Switch and EventOff(outSensor(p)) then
    AdvanceFrom(p)
    Position(succ(p), fstout(queue(p)))
      where Position(sw, e) =
        if Empty(queue(sw)) and dir(sw, bin(dest(e))) ≠ none
        then pos(sw) := dir(sw, bin(dest(e)))
```

Upon moving a package from a pipe into a bin, a submachine checks whether this package has been misrouted.

```
FromPipe2Bin = forall p ∈ Pipe
  if succ(p) ∈ Bin and EventOff(outSensor(p)) then
    MoveToBin(p)
    ReportMisrouting(fstout(queue(p)), succ(p))
where
  MoveToBin(p) = Dequeue(queue(p))
                 Insert(fstout(queue(p)), succ(p))
```

ReportMisrouting(l, b) =
  if $bin(dest(l)) \neq b$ **then** Display$(pkgId(l), b)$

The reader will have noticed that we left various submachines and auxiliary functions unspecified, for example for queue operations or insertion of elements into bins, which we consider as generally understood or not critical for the investigated problem. The ASM method allows the specifier to build models with holes, that is to leave parts of the specification either as completely abstract or as accompanied by informal explanations. For proving properties of such models, appropriate assumptions have to be made for these not furthermore specified parts and have to be proved once the holes are filled by definitions. This pragmatic separation of definition and proof concerns is an efficient technique to piecemeal solve complex problems, which is deeply rooted in the tradition of mathematics.

Refining Position. Assume that the control program cannot access the location $pos(sw)$ directly, but only send a pulse to switch $pos(sw)$. As in the preceding examples this can be reflected by letting the switching be done by an abstract machine Pulses triggered by a pulse, which is emitted by a refined ASM Position$_{ref}$. But one has to pay attention: the update $pos(sw) := dir(sw, bin(dest(e)))$ is executed also if its current value is already $dir(sw, bin(dest(e)))$, though by the semantics of ASMs this update will leave $pos(sw)$ unchanged. But switching changes the value anyway, so that for a correct refinement it must be restricted to the case that the direction needed for the newly arrived package is different from the one needed by the preceeding package. One can include this additional guard condition either in the abstract machine, so that the refinement becomes a pure data refinement, or in its refined version.

Position$_{ref}(sw, e)$ =
  if $Empty(queue(sw))$ **and** $dir(sw, bin(dest(e))) \neq none$
    and $pos(sw) = dir(sw, bin(dest(e)))'$ **then** Emit$(Pulse(sw))$

Verification. The requirement for the control program, namely "to route packages to their destination bins by setting the switches appropriately, and to report misrouted packages", does not state anything about the relation between correct routing and misrouting and when misrouting is to be expected. Given that

- reader and sensors are guaranteed to detect packages separately
- packages cannot overtake in neither conveyor nor pipes nor switches

the following can be proved for PackageRouter:

Lemma 1. *Every package from the conveyor belt eventually arrives at some bin b (if* PackageRouter *is not stopped before). If the package never meets another package in a switch, then b is its associated destination bin. It the package is misrouted,* Display$(pkgId(l), b)$ *will be activated, where l is the label associated to the package and sent by the reader to* PackageRouter.

This can be proved by an induction on the $level(sw)$ of the switches the package goes through in the router tree. For the induction step a stronger hypothesis is needed for packages that never meet another package in a switch, so that for the proof the correctness statement has to be strengthens as follows:

- For every package that never meets another package in a switch and for every switch sw, the guard of $\textsc{Position}(sw, l)$ is true when the package enters sw and $pos(sw)$ is correctly set to $dir(sw, bin(dest(l)))$.

3 Enriching FSMs to ASMs

Following [22, 30] we start by analyzing Finite State Machines, recognizing them as the archetype of Abstract State Machines[10]. In fact, from the practitioner's point of view it seems obvious that to accurately characterize Virtual Machines, it suffices to extend FSM instructions from symbol-reading/writing to reading and updating of arbitrarily structured abstract data. This idea directly leads to the simple concept of ASMs.

3.1 Generalizing FSM States

The well known interpretation of FSM instructions

in state i reading input a, go to state $\delta(i, a)$ and print output $\lambda(i, a)$

of Mealy automata (similarly for Moore automata) can be formalized as follows by simultaneous updates of a control state location (read: a variable) ctl_state and an output location out when an input event is present (read: when the input location in is defined).[11]

$\textsc{MealyFsm}(in, out, \delta, \lambda) =$ **if** $Defined(in)$ **then**
  $ctl\_state := \delta(ctl\_state, in)$
  $out := \lambda(ctl\_state, in)$

Two restrictions one sees here are characteristic for the FSM computation model, besides the strict separation of input and output:

- *only three locations* are read resp. updated (per step): in, ctl_state, out,
- *only three special data types* are used: finite sets of a) symbols representing input/output (letters of an alphabet) and of b) abstract control states representing a bounded memory (typically written as labels or integers).

[10] The original definition of ASMs in [77] was motivated by a different goal: an epistemological desire to generalize Turing's thesis.

[11] We consider here deterministic FSMs, since non-deterministic FSMs can be modeled using the ASM **choose** construct described below. For the sake of simplicity of exposition we suppose each input to be consumed by executing a transition, technically speaking in to be a monitored function as explained below.

ASMs result from withdrawing these restrictions and permitting a machine in each step to read and update *arbitrarily many, possibly parameterized, locations whose values can be of arbitrary type*. Consequently as rule guard an arbitrary condition may be used, formulated in terms of the machine signature and not restricted to the input-definedness predicate.

Stated differently, the *notion of state* is generalized from the three FSM-locations holding FSM-specific values to an arbitrary set of updatable locations where values of whatever type reside, whether atomic or structured: objects, sets, functions, trees, etc. This flat view of state (read: abstract machine memory) lends itself to standard methods for grouping of data into a modular memory structure. A common method is to group subsets of data into tables, via an association of a value to each table entry $(l, (a_1, \ldots, a_n))$, also called *location* (think of it as an array variable). Here l plays the role of the name of the table, the sequence $(a_1, \ldots, a_n)$ the role of an entry, $l(a_1, \ldots, a_n)$ denotes the value currently contained in the table entry $(l, (a_1, \ldots, a_n))$. In logic such a table is called the interpretation of a function or a predicate. The common mathematical notion of structure, as explicitly defined by Tarski [108] and since then in the center of the model theory branch of mathematical logic, is defined as a set of tables. It represents the most general notion of structure which has come up in occidental science and thus is what we need for a sufficiently general notion of Virtual Machine or ASM state.[12] Via the view of a Tarski structure as given by domains of objects coming with predicates (attributes) and functions defined on them, there is also a close relation to the object-oriented understanding of classes and their instances.

In accordance with the generalization of FSM states, consisting of three locations, to an ASM state consisting of an arbitrary set of parameterized locations, in one step an ASM can update simultaneously the value not only of two or three, but of arbitrarily many locations. This generalizes the above FSM-transition rules to *guarded update rules* (called ASM rules) of the following form:

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

where *Updates* is a finite set of assignments of the form $f(t_1, \ldots, t_n) := t$.

Such a view is also taken in [3, pg.52] "to completely separate, during the design, ... individual assignments from their scheduling". Sets of guarded update rules as above in fact constitute a normal form for a class of ASMs which suffice to define every event-B model (see the event-B-model normal form ASMs defined in Sect. 6.1) and to compute every synchronous UML activity diagram [32].

A basic ASM is therefore defined by a finite set of ASM rules, which play the role the instructions play for an FSM. They constitute a mathematical

[12] This view of Tarski structures supports a generalization of Parnas' table technique [94, 93] as a convenient notation for ASMs, as detailed in [21, 22].

substitute for the intuitive concept of UML activity diagram transitions **upon** *Event* **do** *Action*, defining actions as value changes of some locations.

The concept of computation (run) of an ASM is the same as for FSMs, except that the possibility of having multiple locations updated in one step is further enhanced by the following stipulation: an ASM, instead of executing per step one rule as do FSMs, fires in each step all its rules whose guard is true in the given state.[13] This synchronous parallelism in an ASM step helps the designer to make the independence of multiple actions explicit and to avoid introducing irrelevant sequentializations of orthogonal features. For *asynchronous multi-agent ASMs* it suffices to generalize the notion of run from sequences of moves (execution of rules) of just one basic ASM to *partial orders* of moves of multiple agents, each executing a basic ASM, subject to a natural *coherence condition*, see [45, Def.6.1.1].

The general form of ASM rules no longer shows the particular structure determined by the control states $i, j, \ldots$ of an FSM, which however is particularly useful for modeling control systems, protocols, business processes and the like.[14] We therefore proposed in [22] the name *control state ASMs* for ASMs which keep at their top level the characteristic FSM control states, as a means to model some overall status or mode guiding the execution of guarded synchronous parallel updates of the underlying rich state. Formally, control state ASMs are ASMs where all the rules have the form $\textsc{Fsm}(i, \textbf{if } cond \textbf{ then } rule, j)$, standing for the following ASM rule (where *rule* is supposed to also be an ASM rule):

if $ctl\_state = i$ **and** *cond* **then**
  rule
  $ctl\_state := j$

To display such rules often the standard graphical notation for FSMs is used, where circles represent the control states, rhombs the guard *cond*ition and rectangles the *rule* body. See Fig. 1–Fig. 4.

For pragmatic reasons—ease of modeling real-life VMs—we are going to indicate in the next subsection two further constructs to form basic ASM rules, one which makes it possible to explicitly name forms of non-determinism and one which enhances the parallelism of finitely many simultaneous updates. Similarly we freely use other standard notations, where a rigorous definition of their meaning can be given.

[13] More precisely: to execute one step of an ASM in a given state S determine all the fireable rules in S (s.t. *Condition* 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. This yields the successor state S' of S.

[14] See the use of modes in [93] as a means to structure the set of states.

3.2 Classification of Locations, Non-Determinism, Parallelism

An analysis of the different roles played by the locations and functions appearing in an FSM yields what is known as the ASM classification of locations and functions. Some locations or functions are *static*, meaning that their values do not depend on the (dynamics of) states, e.g. the two FSM-functions δ, λ that are defined by the FSM program. Static ASM locations can be given purely functional or axiomatic definitions, as done for the locations $timer(phase)$ in Sect. 2.2. Thus ASMs provide a framework for a theoretically well-founded, coherent and uniform *practical combination of abstract operational descriptions with functional and axiomatic definitions.*[15]

Locations whose values may depend not only on the values of their parameters, but also on the states where they are evaluated, are called *dynamic*. Examples are the FSM-locations in, ctl_state, out. These locations can have four different roles:[16]

- *in* is only read by the FSM and updated only by the environment. Such locations of an ASM M, that are only readable by the machine and writable only by other machines or the environment, are termed *monitored* for M. It is often convenient to describe their meaning for M axiomatically, by a list of assumptions, thus relegating the proof for these assumptions to the model of the other machine(s) where the computation of the monitored locations takes place (*divide-and-conquer* technique).
- *out* is only written by the FSM and read only by the environment. Such locations of an ASM M, that are only writable by M and readable only by other machines or the environment, are termed *output* locations of M.
- *ctl_state* is read and updated by an FSM. ASM locations that are readable and writable only by M are called *controlled* locations of M.
- ASM locations that are readable and writable by M and some other machine or the environment are called *shared*. Typically protocols are used to guarantee the consistency of updates of such locations.

This classification distinguishes between the roles different machines (e.g. the system and its environment) play in using dynamic locations for providing or updating their values. Monitored and shared locations represent two general mechanisms to specify *communication* types between different ASMs. For *modularization* purposes we also distinguish between basic and derived ASM locations. Derived locations are those whose definition in terms of basic locations is fixed and may be given separately, e.g. in some other part ("module" or "class") of the machine or by axioms, equations, etc. For an example see the function $succ(sw)$ in Sect. 2.3

[15] This avoids the alleged, though unjustified and in fact destructive, dichotomy between declarative and operational design elements, which unfortunately has been strongly advocated in the literature over the last thirty years. See the discussion at the end of Sect. 6.2.

[16] The naming is influenced by its pendant in Parnas' Four-Variable-Model [93].

Functions or predicates are called of a type if all their locations are of that type. Selection functions constitute a particularly important class of monitored functions, for which also the following special notation is provided to make the inherent non-determinism explicit.

choose x **with** ϕ **in** *rule*

standing for the rule to execute *rule* for one element x, which is arbitrarily chosen among those satisfying the selection criterion ϕ. Similarly, also the synchronous parallelism which is already present in the execution of ASM rules is extended by the following standard notation:

forall x **with** ϕ **do** *rule*

standing for the simultaneous execution of *rule* for every element x satisfying the property ϕ.

4 ASM Ground Model Technique

The concept of ASM ground model goes back to [15, 16, 18, 19, 17, 20], where it has been used to produce a faithful ASM model for the at the time to-be-defined ISO standard of Prolog [35, 42]. We describe here the foundational and technical characteristics of ASM ground models and refer for a more detailed dicussion of the concept to [24, 29].

The goal of building a ground model is to turn given informal requirements into a clear, unambiguous, accurate, complete and authoritative reference document for their intended content. This document is to be used for the evaluation, the implementation and possible changes of the requirements. Ground models have to be formulated and analyzed at the level of abstraction of the given application domain, prior to coding in any programming language. In other words ground models are specifications that precisely and authoritatively define, in rigorous application domain terms and at the level of detailing that is determined by the application, what the to-be-constructed software-controlled system is supposed to do. Ground models constitute a "blueprint" of the to be implemented piece of "real world". In the semiconductor industry they are named "golden models" [102, pg.26]. They represent what Brooks [48] calls "the conceptual construct" or the "essence" of a software system, whose definition precedes the development of its machine-managed representation by code. It must be possible to justify such a definition as

- *consistent* internally,
- *correct* and *complete* with respect to the intuitions underlying the informal requirements.

These three properties characterize specifications we call ground models. To establish these properties, every available scientific or engineering technique must be usable. This includes inspection and review of the ground

models to get the correctness and completenss properties checked by application domain experts (or potential users) and system designers. They must be enabled to use a combination of tool-supported simulation techniques—for systematic experiments (model checking and testing) with the models—and of mathematical verification techniques—to show the model to possess the properties of interest, e.g. as part of a certification procedure.

In [29] we explain the meaning of these three basic properties in more detail and show that to establish them needs a ground model language that satisfies the following two properties.

- The language is understood by all the parties involved. It mediates between the application world, where user and domain experts live, and the world of mathematical models and software-intensive systems, where software architects, programmers, testers, reviewers, maintenance persons live.
- The language provides means for a combined use of both model validation by simulation (testing or model checking) and property verification by mathematical proof.

The language of ASMs satisfies these conditions, clearly separating models and their properties. It also permits to construct ground models with the following three constituent attributes:

- *precision* to satisfy the required accuracy exactly,
- *minimality*, abstracting from details that belong only to the further design and not to the application problem,[17]
- *simplicity* to be understandable, rigorously analyzable and acceptable as contract by domain experts, system architects, reviewers and testers.[18]

Thus ground models share all the properties Parnas advocates convincingly for the software documentation provided by a good engineering discipline [93]. Obviously most of these properties—precision, accuracy, consistency, correctness, completeness, authoritative character—must be preserved for the further documentation produced on the way from ground models to code. This documentation of the detailed design process is provided by the ASM refinement method we are going to shortly characterize in the next section.

One final word on the often heard claim that such a ground model construction and analysis effort are an add-on one better avoids in an efficient sofware engineering process. This claim reflects only the fact that numerous current system development approaches consider the code as the true definition of the system, excluding any other authoritative description of the

[17] The minimality avoids to define ground models that restrict the problem solution unnecessarily. It thus helps to leave the design space open as much as possible.

[18] Experience in the following domains has confirmed that the ASM language is understandable for domain experts without computer science education: railway, control and telephony systems [31, 41, 49], business and aviation security processes [34, 7, 13, 75, 76], linguistics [83, 90, 91, 54], biology [87], social sciences [47, 46].

system-to-be-built before it has been encoded. Among the typical, rather expensive effects of this view one finds the following: a) the final system may not really do what it was required by the customer to do, b) the final system is not well-understood (first of all not by the application domain experts who have to work with it, but often also not by the software specialists who face serious difficulties in analyzing, understanding and repairing unexpected system breakdowns), c) the testing effort becomes overwhelming (without the possibility of guaranteeing a certifiable standard of reliability), d) system changes can become rather difficult to program and hard to control (e.g. with respect to their compatibility with some previously guaranteed behavior), e) maintenance becomes a nightmare once the persons who have led the development are not available any more, etc. Careful construction and analysis of ground models, combined with their stepwise detailing (refinement) to code, is a means to prevent such undesired effects of missing conceptual application-centric control over the system. Thus the effort spent on ground models is highly compensated by what is saved in later development stages like testing, inspection and maintenance. See also the remark at the end of Sect. 5 and [29] for further discussion.

5 ASM Refinement Concept for Detailed Design

Parnas [93] explains why good software documentation must record the key design decisions in a transparent and easily accessible way. Typically one has to take numerous and often orthogonal design decisions when implementing a ground model by code. Refined models document those design decisions that do not belong to the application problem and therefore by the minimality property do not appear in the ground model, but pertain to the implementation of the algorithmic problem solution.[19] In fact some authors distinguish between requirements specifications, documented by ground models, and design (also called technical) specifications with the frequent "explosion of 'derived requirements' (the requirements for a particular design solution), caused by the complexity of the solution process" [95, Fact 26].

We have generalized the classical refinement method [112, 58] to the mathematically precise notion of structure-transforming pseudo-code defined by

[19] As already observed in Sect. 1 for ground models, this distinction does not pertain to the process of model building. The classification of what belongs to the "essence" of the system and what only to its implementation may change during the design process, typically when "implementation decisions are made before specification is complete and the decisions can have a major effect on the further specification of the system" [107, p.440]. The final definition, yielding a document with the hierarchy of stepwise refined models, can be given only at the end of the design process. This document too is typically re-opened during maintenance for change management, where it has to be synchronized with the decisions taken for the system change.

ASMs (see [25] for a recent survey with further details). It allows the designer to fine-tune any given abstract construct, which can be viewed as "the already-fixed portion of a multi-step system development", to the new details needed to realize a design decision by an implementation, "the yet-to-be-done portion of a multi-step system development" [107, p.438]. In this way the ASM refinement method directly supports the practitioner's view of an implementation as "a multi-step process. Each stage of this process is a specification for what follows" [107, p.440]. For this reason and differently from other refinement notions in the literature, an ASM refinement step can simultaneously involve both the signature (the data structure) and the control structure (the flow of step-by-step operations). In fact, in choosing how to refine an ASM M to an ASM M^*, one has the freedom to define the following five concepts (see Fig. 6):

- a notion (signature and intended meaning) of *refined state*,
- a notion of *states of interest* and of *correspondence* between M-states S and M^*-states S^* of interest. Since certain intermediate M-states or M^*-states may be irrelevant (not of interest) for the refinement relation, they are hidden by defining what are the corresponding states of interest, namely those pairs of states in the runs one wants to relate through the refinement. Usually initial states in M and M^* correspond to each other, similarly for pairs of final states (if there are any),
- 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. This ASM refinement concept generalizes other more restricted refinement notions in the literature, as analysed in [96, 97], and scales to the controlled and well documented stepwise development of large systems.

In particular the ASM refinement method supports modular system descriptions, including the modularization of ASM refinement correctness proofs aimed at mechanizable proof support, for examples see [96, 106, 36, 43].

If the authoritative character of both ground and intermediate models is taken seriously, then refined models support change management, namely by

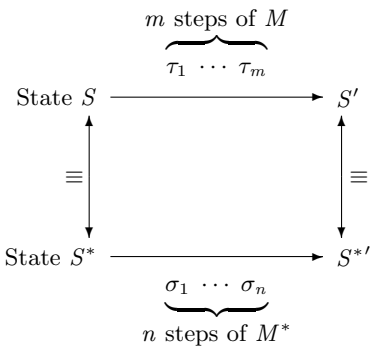

Fig. 6. The ASM refinement scheme

documenting natural points for possible design changes, e.g. to cover new cases or to optimize a solution. Obviously this implies that to maintain the series of models as documentation of a changed system, the models affected by the change have to be updated and to be kept synchronized with the ground model or other intermediate models, as already pointed out in the preceeding footnote. If this implies additional modeling work, it saves work for the coding during maintenance steps. The report [41] on the successful use of the ASM method for an industrial reengineering project shows that in certain situations, using a compiler from the underlying class of ASMs to executable code can even make re-coding completely superfluous and keep maintenance at the application-centric modeling level, where the changed requirements are formulated. More generally the report illustrates that using the ASM method produces no useless overhead: it typically shifts the most error-prone part of the development effort from the late coding and testing phases to the early modeling phase. This phase is dictated by high-level architectural concerns, major design decisions, the need to achieve a correct overall system understanding by experts and to lay the ground for a certification of the required trustworthiness. But the amount of the overall system development effort is essentially kept unchanged. See also the remark at the end of section 4.

Good intermediate models, refining an otherwise unaltered model to capture a to-be-realized design feature, provide descriptions which are easier to understand than the code. This is helpful in particular when the maintenance team has to take its information on the system behavior from the documentation, after the development team has left. The examples in the literature

(e.g. [43, 36, 41, 106]) and my own industrial experience show that, contrary to Parnas' claim [93] on the classical refinement method, good ASM refinements do not lead to long or repetitive programs, but allow the designer to succinctly document design decisions in a focussed and piecemeal manner, one by one, avoiding any irrelevant detail or repetition.

6 Integration of Multiple Design and Analysis Methods

Due to the mathematical character and the simplicity of the conceptual framework of the ASM method, the method can be combined in a semantically coherent and natural way with any other accurate (scientifically well-defined) design and analysis technique. We illustrate this in Sect. 6.1 by a characterization of event-B models as a particular class of ASMs, coming with a specialized refinement definition. In Sect. 6.2 we highlight the embedding of further (including so-called semi-formal) approaches to system design and analysis into the ASM method.

6.1 ASM-Characterization of Event-B Machines

Event-B models are presented in [2, 3, 6] as an extension of B-machines [1], which have been used with success for the development of reliable industrial control software in a variety of safety-relevant large-scale industrial applications [5]. A discussion of the foundational relations between B-machines and ASMs can be found in [22, Sect.3.2]. We limit ourselves here to characterize event-B models as a class of ASMs, following [30, Sect.4]. The three basic constituents to analyze are the notions of state, event and refinement.

States of event-B models can be viewed as Tarski structures with a static and a dynamic part. The static part, called *context*, consists of three items:

- sets s which represent the universes of discourse (domains),
- constants c which are supposed to have a fixed interpretation,
- properties (c, s) used as axioms characterizing the intended model class.

The dynamic state part consists of variables v and an environment, which is viewed as another event-B model. Inputting is captured by non-determinacy. The states are supposed to be initialized, technically via a special event with true guard.

Events come in the form

if *guard* **then** *action*

where the *guard* is a closed first-order set theory formula with equality and *action* has one of the following three forms (we use the ASM notation which is slightly different from Abrial's notation):

- Updates. The syntactical event-B-model reading of Updates is that of a simultaneous substitution $v_1, \ldots, v_n := e_1, \ldots, e_n(v)$ of v_i by e_i, whereas

its equivalent semantical ASM reading is that of a finite set of simultaneous updates $v_i := e_i$ of the values of variables (parameter-free locations) by expression values,

- **skip**,
- **choose** x **with** $P(x, v)$ **in** *Updates* where *Updates* is a simultaneous substitution $v_1, \ldots, v_n := e_1, \ldots, e_n(x, v)$

Abrial views the operational interpretation of events, for example as defined above by the semantics of ASM rules, only as an informal intuitive account, whereas the official semantical definition comes in the form of logical descriptions using pre/post conditions and invariants. For the present comparison with ASMs it suffices to consider invariants as descriptions of properties the designer wants and claims to hold in every state that is reachable from an initial state. Obviously eventually such invariants will have to be justified, by whatever available means.

The preceding definition of events represents an event-B normal form for ASM rules. The outstanding characteristic feature, namely the underlying interleaving semantics ("at each moment only one event can occur"), is reflected by a top-level choice among the finitely many events which may be applicable. For this non-deterministic choice among ASM rules $R(i)$ we use the following notation:

$$R(0) \textbf{ or } \ldots \textbf{ or } R(n-1) = \textbf{choose } i \textbf{ with } i < n \textbf{ in } R(i)$$

There is a technical consequence of this interleaving interpretation the designer should be aware of. The form in which simultaneous updates are collected under a guard into one event has an impact on the semantics of the model, differently from the basic parallelism of ASMs where in each state every applicable rule (read: event) is applied. The splitting of updates into different events implies some non-deterministic scheduling of events with overlapping guards. This is reflected by the following event-B normal form (the part in brackets [ ] is optional):

$$\begin{aligned} &\mathit{Rule}_1 \textbf{ or } \ldots \textbf{ or } \mathit{Rule}_n \textbf{ where } \textbf{forall } 1 \leq i \leq n \\ &\quad \mathit{Rule}_i = \textbf{if } \mathit{cond}_i \textbf{ then } \ [\textbf{choose } x \textbf{ with } P_i(x) \textbf{ in}] \ \ \mathit{Updates}_i \end{aligned}$$

In this normal form the limit cases are that cond_i is always true (unconditional updates) or that the set $\mathit{Updates}_i$ is empty (**skip**). We have disregarded the constraint imposed in [4] that in an event-B model, no parallel update is allowed for the same variable. This constraint is only of technical nature and prevents the case with inconsistent update sets to happen, whereas the semantics of ASMs prescribes in this case only that the computation is aborted.

In comparison to the general form of ASM rules there are no rules of the form **forall** x **with** $P(x)$ **do** *Rule* and the only external **choose** (i.e. one that is not applied directly to a set of updates) which is permitted is the top-level one on rules defining the interleaving model so that there is no further nesting of choices. This prevents the designer from using complex quantifier-change structures in his models, so that they have to be circumscribed by different

means in case they are part of a 'natural' description of intended system behavior.

Refinement of event-B models plays a crucial role for the B-method, as it does for the ASM method. However, the goal to provide definitions of event-B models together with mechanically checked (interactive or automated) proofs of the desired invariants dictates a restriction to certain forms of the general ASM refinement concept. Using the analysis of the ASM refinement concept in [96, 97] one can say that for event-B model refinements, only $(1, n)$-refinements with $n > 0$ are permitted. No $(1, 0)$-refinement is allowed, reflecting the condition that each abstract event must be refined by at least one refined event, and no (n, m)-refinement with $n > 1$ is allowed. In addition event-B model refinements must satisfy the following constraints:

- in a $(1, n)$-refinement $F_1, \ldots, F_n, F$ of E, each F_i is supposed to be a new event refining **skip**,
- the new events F_i do not diverge,
- if the refinement deadlocks, then the abstraction deadlocks.

As to the *observables* in terms of which event-B refinements are formally defined, they correspond to what we have called the *locations of interest* of an ASM [25]. Formally they can be viewed as projections of state variables. Technically speaking, in event-B refinements the observables are variables which are required to satisfy the following conditions:

- they are fresh variables with respect to state variables and to invariants,
- they are modifiable only by observer events of form $a := A(v)$,
- they depend only on state variables v,
- the abstract observables $A(v)$ can be 'reconstructed' from the refined ones by an equation $A(v) = L(B(w))$ which represents an "invariant gluing the abstract observables to the refined ones".

The only pragmatically relevant one of these technical conditions on observables is the gluing invariant. In ASM refinements any mathematically accurate scheme to relate refined and abstract observables is allowed, it need not be describable equationally.

6.2 Integrating Special Purpose Methods

In a similar way to event-B models and to UML activity diagrams and Parnas' table technique, mentioned in Sect. 3.1, all the major computation and system design models turned out to be natural instances of specific classes of ASMs (for details see [22, 27]). This confirmes the unifying-framework character of the ASM approach to systems engineering. Many specification approaches are geared for some particular type of application and equipped with specially tailored definition or proof techniques. Such features can be naturally reflected by imposing appropriate restrictions on the class of ASMs, the refinement scheme and the related proof methods.

For the general, but loose UML-based approach to system engineering, the ASM method offers a rigorous, semantically well-defined version. The ASM ground model and refinement concepts replace the loose character of human-centric UML models and of the links the UML framework offers between descriptions at different system design levels. Starting from the accurate ASM-based semantical definition of the various UML diagrams and related notations (see [32, 33, 50, 51]), this equips UML-based practice with the degree of mathematical precision that distinguishes a scientifically rooted engineering discipline worth its name.

Another way to seamlessly include so-called semi-formal design techniques into an ASM-based development process goes as follows. The less rigorous a specification is, e.g. when there are reasons to momentarily leave parts of the specification as only informally explained, the more 'holes' the ASM model shows that one has to fill by providing assumptions on the intended meaning. These assumptions have to be discharged for the refined models, where the detailed design introduces the missing elements. Within the framework of mathematics, much of which is what in computer science is called semi-formal, this is a legitimate way to proceed. A similar freedom of formality concerns the notations. As is characteristic for mathematical disciplines, the ASM method is not bound by the straitjacket of a particular formal language, but allows one to freely use any standard algorithmic and mathematical notation. The only condition for adopting any useful description technique, whether textual or tabular or graphical or whatever, is a mathematically rigorous definition of its meaning.

The ASM method also incorporates within one common modeling framework two pairs of approaches that in the literature are frequently, but erroneously, viewed as antagonistic instead of complementary, namely using so-called declarative (denotational) versus operational and state-based versus event-based system descriptions. For the definition of an ASM one can use as much of declarative or denotational characterizations as desired, using functional definitions or logico-axiomatic descriptions. But this does not exclude the use of abstract operational descriptions where the latter turn out to be simpler, more intuitive and easier to refine to code. Declarative definitions are often used to define the background signature of an ASM (via static, monitored, derived functions, see Sect. 3.2). It is also often used to define what among the class of all possible runs of an ASM is considered as a legal run, describing axiomatically a certain number of features one wants to abstract away at the investigated level of abstraction. Similarly, whatever one wants to classify as an event can be included into the declaration of the state signature and be treated correspondingly in rule guards; see for example Event-B, where events are considered as rule firings, or [14] where process-algebraic event-based structuring techniques and concurrency patterns are combined with the state-based abstraction mechanism and synchronous parallelism of ASMs.

In a similar way the general mathematical framework of the ASM method allows one the coherent separation and integration of defining a model and proving model properties. The ASM method does not force you to *simultaneously* define your models and prove properties for them, still less to do this in an *a priori* determined deductive system, but it allows you to add proofs to your definitions, where appropriate, and to do this in various ways, depending on what is needed. Obviously a price has to be paid for this generality: if one wants a machine-assisted mechanical verification of your system, one will have to formalize it in the language of the theorem prover. In this case it will be an advantage if one succeeds to define a model right from the beginning as a set of particular logical or set-theoretical formulae, as is the case for example in the B method [1]. On the other side, since ASMs are not formulae but represent executable models, the ASM method allows one to adopt for abstract models simulation, run-time verification and testing techniques, where proofs for whatever reason are not viable.

7 ASM Method Applications in a Nutshell

The proposal to use Abstract State Machines a) as a precise mathematical form of ground models and b) for a generalization of Wirth's and Dijkstra's classical refinement method [112, 58] to a practical systems engineering framework supporting a systematic separation, structuring and documentation of orthogonal design decisions goes back to [15, 16, 20]. It was used there to define what became the ISO standard of Prolog [35]. Since then numerous case studies provided ground models for various industrial standards, e.g. for the forthcoming standard of BPEL4WS [63], for the ITU-T standard for SDL-2000 [74], for the de facto standard for Java and the Java Virtual Machine [106], the ECMA standard for C# and the .NET CLR [37, 104, 70], the IEEE-VHDL93 standard [38]. The ASM refinement method [25] has been used in numerous ASM-based design and verification projects surveyed in [23].

The ASM method, due to the mathematical nature of its constituent concepts, could be linked to a multitude of tool-supported analysis methods, in terms of both experimental *validation* of models and mathematical *verification* of their properties. The validation (testing) of ASM models is supported by various tools to mechanically execute ASMs, including *ASM Workbench* [55], *AsmGofer* [99], an Asm2C++ compiler [100], C-based *XASM* [8], .NET-executable *AsmL* engine [67], *CoreASM* Execution Engine [62]. The verification of ASM properties has been performed using justification techniques ranging from proof sketches [40] over traditional [36, 39] or formalized mathematical proofs [105, 92] to tool supported proof checking or interactive or automatic theorem proving, e.g. by model checkers [111, 56, 72], KIV [98] or PVS [59, 71]. As needed for a comprehensive development and analysis environment, various combinations of such verification and validation methods

have been supported and have been used also for the correctness analysis of compilers [60, 84] and hardware [110, 109, 101, 79].

For more applications, including industrial system development and re-engineering case studies that show the method to scale to large systems, see the website of the ASM Research Center at *www.asmcenter.org* and the Asm-Book [45].

8 Concluding Remarks

The ASM method is not a silver bullet, but shares the intrinsic limitations of every engineering discipline rooted in mathematics. Whereas ASMs are easily grasped and correctly understood by system engineers and application domain experts, namely as pseudo-code or FSMs over arbitrary data types, is not an easy task to teach or to learn a judicious use of the inherent abstraction potential for constructing appropriate ground models and refinement hierarchies. There are also pragmatical limitations, which the method shares with other rigorous practical methods, as for example the B-method [5]. They have to do with the proposed shift in the current software system development process. The proposal is not to start coding right away and not to relegate the correctness and reliability issues to an ever growing testing phase. Instead it is suggested to first construct and reason about accurate ground models for the requirements and exact provably correct interfaces for the various design decisions, leading to more and more detailed models. From that and only from that basis should executable code be generated, which then comes with objectively verifiable and validatable correctness properties one can trace through the refinement hierarchy to their ground model pendant. It is by no means easy to change an established industrial practice.

Acknowledgement. Thanks to Uwe Glässer, Antje Nowack, Bernhard Thalheim, Mona Vajihollahi and three anonymous referees for their critical comments on a preliminary version of this paper.

References

1. J.-R. Abrial. *The B-Book*. Cambridge University Press, Cambridge, 1996.
2. J.-R. Abrial. Extending B without changing it (for developing distributed systems). In H. Habrias, editor, *Proc. 1st Conf. on the B Method*, pages 169–190. IRIN Institut de recherche en informatique de Nantes, 1996.
3. J.-R. Abrial. Event based sequential program development: application to constructing a pointer program. In *Proc. FME 2003*, pages 51–74. Springer, 2003.
4. J.-R. Abrial. Discrete system models. Version 2, September 2004.
5. J.-R. Abrial. Formal methods in industry: Achievements, problems, future. In *Proc.ICSE'06*, Shanghai (China), May 2006. ACM.
6. J.-R. Abrial. Event-B book. In preparation, title to be determined, 2007.

7. M. Altenhofen, E. Börger, A. Friesen, and J. Lemcke. A high-level specification for virtual providers. *International Journal of Business Process Integration and Management*, 4(1), 2006.
8. M. Anlauff and P. Kutter. Xasm Open Source. Web pages at http://www.xasm.org/, 2001.
9. E. Astesiano and G. Reggio. SMoLCS-driven concurrent calculi. In G. L. H. Ehrig, R. Kowalski and U. Montanari, editors, *Proc. TAPSOFT'87 Vol.1*, volume 249 of *LNCS*, pages 169–201. Springer, 1987.
10. E. Astesiano and G. Reggio. Labelled transition logic: An outline. *Acta Informatica*, 37(11/12), 2001.
11. R. J. R. Back. On correct refinement of programs. *J. Computer and System Sciences*, 23(1):49–68, 1979.
12. R. J. R. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998.
13. A. Barros and E. Börger. A compositional framework for service interaction patterns and communication flows. In K.-K. Lau and R. Banach, editors, *Formal Methods and Software Engineering. Proc. 7th International Conference on Formal Engineering Methods (ICFEM 2005)*, volume 3785 of *LNCS*, pages 5–35. Springer, 2005.
14. T. Bolognesi and E. Börger. Abstract State Processes. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines 2003–Advances in Theory and Applications*, volume 2589 of *LNCS*, pages 22–32. Springer, 2003.
15. E. Börger. A logical operational semantics for full Prolog. Part I: Selection core and control. In E. Börger, H. Kleine Büning, M. M. Richter, and W. Schönfeld, editors, *CSL'89. 3rd Workshop on Computer Science Logic*, volume 440 of *Lecture Notes in Computer Science*, pages 36–64. Springer-Verlag, 1990.
16. 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.
17. E. Börger. Dynamische Algebren und Semantik von Prolog. In E. Börger, editor, *Berechenbarkeit, Komplexität, Logik*, pages 476–499. Vieweg, 3rd edition, 1992.
18. E. Börger. A logical operational semantics for full Prolog. Part III: Built-in predicates for files, terms, arithmetic and input-output. In Y. N. Moschovakis, editor, *Logic From Computer Science*, volume 21 of *Berkeley Mathematical Sciences Research Institute Publications*, pages 17–50. Springer-Verlag, 1992.
19. E. Börger. A natural formalization of full Prolog. *Newsletter of the Association for Logic Programming*, 5(1):8–9, 1992.
20. E. Börger. Logic programming: The Evolving Algebra approach. In B. Pehrson and I. Simon, editors, *IFIP 13th World Computer Congress*, volume I: Technology/Foundations, pages 391–395, Elsevier, Amsterdam, 1994.
21. E. Börger. Evolving Algebras and Parnas tables. In H. Ehrig, F. von Henke, J. Meseguer, and M. Wirsing, editors, *Specification and Semantics*. Dagstuhl Seminar No. 9626, Schloss Dagstuhl, July 1996.
22. E. Börger. High-level system design and analysis using Abstract State Machines. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Current Trends in Applied Formal Methods (FM-Trends 98)*, volume 1641 of *Lecture Notes in Computer Science*, pages 1–43. Springer-Verlag, 1999.

23. E. Börger. The origins and the development of the ASM method for high-level system design and analysis. *J. Universal Computer Science*, 8(1):2–74, 2002.
24. E. Börger. The ASM ground model method as a foundation of requirements engineering. In N.Dershowitz, editor, *Verification: Theory and Practice*, volume 2772 of *LNCS*, pages 145–160. Springer-Verlag, 2003.
25. E. Börger. The ASM refinement method. *Formal Aspects of Computing*, 15:237–257, 2003.
26. E. Börger. 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.
27. E. Börger. Abstract State Machines: A unifying view of models of computation and of system design frameworks. *Annals of Pure and Applied Logic*, 133:149–171, 2005.
28. E. Börger. The ASM method for system design and analysis. A tutorial introduction. In B. Gramlich, editor, *Frontiers of Combining Systems*, volume 3717 of *Lecture Notes in Artificial Intelligence*, pages 264–283. Springer, 2005.
29. E. Börger. Construction and analysis of ground models and their refinements as a foundation for validating computer based systems. *Formal Aspects of Computing*, 2006.
30. 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, 2006. ISBN 3-925386-56-4, Proc. of 2005 Colloquium.
31. 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.
32. E. Börger, A. Cavarra, and E. Riccobene. An ASM semantics for UML activity diagrams. In T. Rus, editor, *Algebraic Methodology and Software Technology, 8th Int. Conf., AMAST 2000, Iowa City, Iowa, USA, May 20-27, 2000 Proceedings*, volume 1816 of *LNCS*, pages 293–308. Springer-Verlag, 2000.
33. E. Börger, A. Cavarra, and E. Riccobene. Modeling the dynamics of UML state machines. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines: Theory and Applications*, volume 1912 of *Lecture Notes in Computer Science*, pages 223–241. Springer-Verlag, 2000.
34. E. Börger, M. Cesaroni, M. Falqui, and T. L. Murgi. Caso di Studio: Mail From Form System. Internal Report FST-2-1-RE-02, Fabbrica Servizi Telematici FST (Gruppo Atlantis), Uta (Cagliari), 1999.
35. 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.
36. E. Börger and I. Durdanović. Correctness of compiling Occam to Transputer code. *Computer Journal*, 39(1):52–92, 1996.
37. E. Börger, G. Fruja, V. Gervasi, and R. Stärk. A high-level modular definition of the semantics of C#. *Theoretical Computer Science*, 336(2–3):235–284, 2005.
38. 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.

39. E. Börger and S. Mazzanti. A practical method for rigorously controllable hardware design. In J. P. Bowen, M. B. Hinchey, and D. Till, editors, *ZUM'97: The Z Formal Specification Notation*, volume 1212 of *Lecture Notes in Computer Science*, pages 151–187. Springer-Verlag, 1997.
40. E. Börger and L. Mearelli. Integrating ASMs into the software development life cycle. *J. Universal Computer Science*, 3(5):603–665, 1997.
41. 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.
42. E. Börger and D. Rosenzweig. A mathematical definition of full Prolog. *Science of Computer Programming*, 24:249–286, 1995.
43. 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.
44. 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, 2000.
45. E. Börger and R. F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003.
46. P. L. Brantingham, B. Kinney, U. Glässer, P. Jackson, and M. Vajihollahi. A Computational Model for Simulating Spatial and Temporal Aspects of Crime in Urban Environments. In L. Liu and J. Eck, editors, *Artificial Crime Analysis Systems: Using Computer Simulations and Geographic Information Systems*. Idea Publishing, 2006.
47. P. L. Brantingham, B. Kinney, U. Glässer, K. Singh, and M. Vajihollahi. A Computational Model for Simulating Spatial Aspects of Crime in Urban Environments. In M. Jamshidi, editor, *Proc. of 2005 IEEE International Conference on Systems, Man and Cybernetics*, pages 3667–74. IEEE, October 2005.
48. F. P. J. Brooks. No silver bullet. *Computer*, 20(4):10–19, 1987.
49. 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.
50. A. Cavarra. *Applying Abstract State Machines to Formalize and Integrate the UML Lightweight Method*. PhD thesis, University of Catania, Sicily, Italy, 2000.
51. A. Cavarra, E. Riccobene, and P. Scandurra. Integrating UML static and dynamic views and formalizing the interaction mechanism of UML state machines. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines 2003–Advances in Theory and Applications*, volume 2589 of *Lecture Notes in Computer Science*, pages 229–243. Springer-Verlag, 2003.
52. A. B. Cremers and T. N. Hibbard. Formal modeling of virtual machines. *IEEE Trans. Software Eng.*, SE-4(5):426–436, 1978.
53. W. P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge University Press, Cambridge, 1998.

54. S. Degeilh and A. Preller. ASMs specify natural language processing via pregroups. Technical Report Local Proceedings ASM'04, Department of Mathematics, University Halle-Wittenberg, 2004.
55. 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. .
56. G. Del Castillo and K. Winter. Model checking support for the ASM high-level language. In S. Graf and M. Schwartzbach, editors, *Proc. 6th Int. Conf. TACAS 2000*, volume 1785 of *LNCS*, pages 331–346. Springer-Verlag, 2000.
57. J. Derrick and E. Boiten. *Refinement in Z and Object-Z*. Formal Approaches to Computing and Information Technology. Springer-Verlag, 2001.
58. E. W. Dijkstra. Notes on structured programming. In O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors, *Structured Programming*, pages 1–82. Academic Press, 1972.
59. A. Dold. A formal representation of Abstract State Machines using PVS. Verifix Technical Report Ulm/6.2, Universität Ulm, Germany, July 1998.
60. A. Dold, T. Gaul, V. Vialard, and W. Zimmermann. ASM-based mechanized verification of compiler back-ends. In U. Glässer and P. Schmitt, editors, *Proc. 5th Int. Workshop on ASMs*, pages 50–67. Magdeburg University, 1998.
61. H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1,2*. Springer, 1985.
62. R. Farahbod et al. *The CoreASM Project*. http://www.coreasm.org.
63. R. Farahbod, U. Glässer, and M. Vajihollahi. An Abstract Machine Architecture for Web Service Based Business Process Management. *International Journal on Business Process Integration and Management*, 2006.
64. L. M. G. Feijs and H. B. M. Jonkers. *Formal Specification and Design*. Cambridge University Press, Cambridge, 1992.
65. L. M. G. Feijs, H. B. M. Jonkers, C. P. J. Koymans, and G. R. Renardel de Lavalette. Formal definition of the design language COLD-K. Technical Report No. 234/87, Philips Research Laboratories, 1987.
66. J. Fitzgerald and P. G. Larsen. *Modeling Systems. Practical Tool and Techniques in Software Development*. Cambridge University Press, Cambridge, 1998.
67. Foundations of Software Engineering Group, Microsoft Research. AsmL. Web pages at http://research.microsoft.com/foundations/AsmL/, 2001.
68. N. G. Fruja. The correctness of the definite assignment analysis in C#. *Journal of Object Technology*, 3(9):29–52, October 2004.
69. N. G. Fruja. *Type Safety of C# and .NET CLR*. PhD thesis, ETH Zürich, 2006.
70. N. G. Fruja and E. Börger. Modeling the .NET CLR Exception Handling Mechanism for a Mathematical Analysis. *J. of Object Technology*, 5(3):5–34, 2006. .
71. 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.
72. 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.

73. C. George, P. Haff, K. Havelund, A. E. Haxthausen, R. Milne, C. B. Nielsen, S. Prehn, and K. R. Wagner. *The RAISE specification language*. Prentice Hall, 1992.
74. U. Glässer, R. Gotzhein, and A. Prinz. Formal semantics of SDL-2000: Status and perspectives. *Computer Networks*, 42(3):343–358, June 2003.
75. U. Glässer, S. Rastkar, and M. Vajihollahi. Computational Modeling and Experimental Validation of Aviation Security Procedures. In S. Mehrotra, D. D. Zeng, H. Chen, B. M. Thuraisingham, and F.-Y. Wang, editors, *Intelligence and Security Informatics, IEEE International Conference on Intelligence and Security Informatics, ISI 2006, San Diego, CA, USA, May 23-24, 2006, Proceedings*, volume 3975 of *LNCS*, pages 420–431. Springer, 2006.
76. U. Glässer, S. Rastkar, and M. Vajihollahi. Modeling and Validation of Aviation Security. In H. Chen and C. Yang, editors, *Intelligence and Security Informatics: Techniques and Applications*. Springer, 2006.
77. Y. Gurevich. Evolving algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
78. J. V. Guttag, E. Horowitz, and D. R. Musser. Abstract data types and software validation. *Commun. ACM*, 21(12), 1978.
79. A. Habibi. *Framework for System Level Verification: The SystemC Case*. PhD thesis, Concordia University, Montreal, July 2005.
80. C. Heitmeyer. Software cost reduction. In J. J. Marciniak, editor, *Enc. of Software Engineering*. 2nd edition, 2002.
81. G. Hommel. Vergleich verschiedener spezifikationsverfahren am beispiel einer paketverteilanlage. Technical report, Kernforschungszentrum Karlsruhe, August 1980.
82. M. Jackson. *Problem Frames*. Addison-Wesley, 2001.
83. D. E. Johnson and L. S. Moss. Grammar formalisms viewed as Evolving Algebras. *Linguistics and Philosophy*, 17:537–560, 1994.
84. A. Kalinov, A. Kossatchev, A. Petrenko, M. Posypkin, and V. Shishkov. Using ASM specifications for compiler testing. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines 2003–Advances in Theory and Applications*, volume 2589 of *LNCS*, page 415. Springer, 2003.
85. B. H. Lisko and S. N. Zilles. Specification techniques for data abstraction. *IEEE Trans. Software Eng.*, SE-1, March 1975.
86. K. Mani Chandy and J. Misra. *Parallel Program Design. A Foundation*. Addison Wesley, 1988.
87. D. Mazzei. Ottimizzazione di un bioreattore high throughput con strategia di controllo autonoma. Master's thesis, University of Pisa, October 2006.
88. L. Mearelli. Refining an ASM specification of the production cell to C++ code. *J. Universal Computer Science*, 3(5):666–688, 1997.
89. J. M.Morris. A theoretical basis for stepwise refinement. *Science of Computer Programming*, 9(3), 1987.
90. L. S. Moss and D. E. Johnson. Dynamic interpretations of constraint-based grammar formalisms. *J. Logic, Language, and Information*, 4(1):61–79, 1995.
91. L. S. Moss and D. E. Johnson. Evolving algebras and mathematical models of language. In L. Polos and M. Masuch, editors, *Applied Logic: How, What, and Why*, volume 626 of *Synthese Library*, pages 143–175. Kluwer Academic Publishers, 1995.

92. S. Nanchen and R. F. Stärk. A security logic for Abstract State Machines. In W. Zimmermann and B. Thalheim, editors, *Abstract State Machines 2004*, volume 3052 of *LNCS*, pages 169–185. Springer-Verlag, 2004.
93. D. L. Parnas. The use of precise documentation in software development. Tutorial at FM 2006, see http://fm06.mcmaster.ca/t8.htm, August 2006.
94. D. L. Parnas and J. Madey. Functional documents for computer systems. *Sci. of Computer Programming*, 25:41–62, 1995.
95. R.L.Glass. *Facts and Fallacies of Software Engineering*. Addison-Wesley, 2003.
96. G. Schellhorn. Verification of ASM refinements using generalized forward simulation. *J. Universal Computer Science*, 7(11):952–979, 2001.
97. G. Schellhorn. ASM refinement and generalizations of forward simulation in data refinement: A comparison. *Theoretical Computer Science*, 336(2-3):403–436, 2005.
98. G. Schellhorn and W. Ahrendt. Reasoning about Abstract State Machines: The WAM case study. *J. Universal Computer Science*, 3(4):377–413, 1997.
99. J. Schmid. Executing ASM specifications with AsmGofer. Web pages at http://www.tydo.de/AsmGofer.
100. J. Schmid. Compiling Abstract State Machines to C++. *J. Universal Computer Science*, 7(11):1069–1088, 2001.
101. J. Schmid. *Refinement and Implementation Techniques for Abstract State Machines*. PhD thesis, University of Ulm, Germany, 2002.
102. Semiconductor Industry Assoc. Internat. technology roadmap for semiconductors. Design. http://www.itrs.net/Common/2005ITRS/Design2005.pdf, 2005.
103. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 1989.
104. R. F. Stärk and E. Börger. An ASM specification of C# threads and the .NET memory model. In W. Zimmermann and B. Thalheim, editors, *Abstract State Machines 2004*, volume 3052 of *LNCS*, pages 38–60. Springer-Verlag, 2004.
105. R. F. Stärk and S. Nanchen. A logic for Abstract State Machines. *J. Universal Computer Science*, 7(11):981–1006, 2001.
106. R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag, 2001. .
107. W. Swartout and R. Balzer. On the inevitable intertwining of specification and implementation. *Communications of the ACM*, 25(7):438–440, 1982.
108. A. Tarski. Der Wahrheitsbegriff in den formalisierten Sprachen. *Studia Philosophica*, 1:261–405, 1936.
109. J. Teich, P. Kutter, and R. Weper. Description and simulation of microprocessor instruction sets using ASMs. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines: Theory and Applications*, volume 1912 of *LNCS*, pages 266–286. Springer-Verlag, 2000.
110. J. Teich, R. Weper, D. Fischer, and S. Trinkert. A joint architecture/compiler design environment for ASIPs. In *Proc. Int. Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES2000)*, pages 26–33, San Jose, CA, USA, November 2000. ACM Press.
111. K. Winter. Model checking for Abstract State Machines. *J. Universal Computer Science*, 3(5):689–701, 1997.
112. N. Wirth. Program development by stepwise refinement. *Commun. ACM*, 14(4), 1971.
113. J. C. P. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice-Hall, 1996.