Linking architectural and component level system views by Abstract State Machines *

Egon Börger
Dipartimento di Informatica, Università di Pisa, boerger at di.unipi.it

March 5, 2004

Abstract

In hardware and software design model checkers are nowadays used with success to verify properties of system components [23]. The limits of the approach to cope with the size and the complexity of modern computer-based systems are felt when it comes to provide evidence of the trustworthiness of the entire system that has been built out of verified components. To achieve this task one has to experimentally validate or to mathematically verify the composition of the system. This reveals a gap between the finite state machine (FSM) view of model-checkable components and the architectural system view. In this paper we show how Abstract State Machines (ASM) can be used to fill this gap for both design and analysis, using a flexible concept of ASM component.

1 Introduction

Often model-checking and theorem proving are viewed as competing system verification methods, and as being in contrast to experimental validation methods which are based upon simulation and testing. However, all these methods are needed to cope with complex hw/sw systems, which need high-level models to exhibit the behavior that goes beyond what can be defined and analyzed by looking at the Finite State Machine (FSM) components. We explain in this paper how the framework of Abstract State Machines (ASMs) allows one to smoothly integrate current verification and validation methods into a design and analysis approach which permits to uniformly link abstract and detailed system views.

ASMs can be introduced as a natural extension of FSMs, in the form of Mealy-ASMs or of control-state ASMs as defined in [9], namely by allowing a) *states* with arbitrarily complex or abstract data structures and b) *runs* with transitions where multiple components execute simultaneously (synchronous parallellism). Concerning the notion of run, *basic synchronous ASMs* come with the classical FSM notion of sequential run, characterized by sequences of successive computation steps of one machine. However, each single ASM step may involve multiple simultaneous actions. The example in Fig. 1, which is taken from [36], defines the top-level sequential structure of the execution semantics of so-called pipe statements in the language SpecC, an extension of C by system-level features which are used in industrial hardware design. These statements are parameterized by an *Init*ialization statement, a *cond*ition which guards an iterative process, by an *Incr*ementing statement used to advance the iteration, and by finitely many subprocesses which are spawned and deleted in a synchronized manner to fill, run and eventually flush the pipe. One finds numerous applications of such synchronous ASM models in hardware-software co-design, see [22, Ch.9] for a survey.

Asynchronously interacting FSMs, for example the globally asynchronous, locally synchronous Codesign-FSMs [35], are captured by asynchronous ASMs where runs are partial orders of moves of multiple agents. Multiple Mealy-ASM or control-state ASM components of an *asynchronous ASM* exhibit locally synchronous behavior, whereas the globally asynchronous system character is reflected by the generalization of the notion of sequential run to a partial order of execution

*Draft. Final version to appear as Chapter 16 of: Christoph Grimm (Ed.), *Languages for System Specification and Verification*, CHDL Series, Kluwer, 2004, pages 247-269.

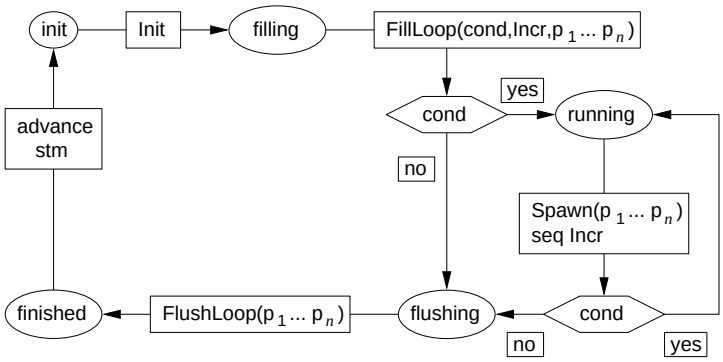

Figure 1: Control state ASM for SpecC pipe statements

steps of the components. For a simple example see the Production Cell ASM with six component machines in [17].

Concerning the extension of the notion of FSM-state, with ASMs the system designer has arbitrary structures available to reflect any given notion of state; we call this for short *freedom of abstraction*. This freedom of abstraction in combination with the powerful refinement notion offered by ASMs (see [11]) provide a uniform conceptual framework for effectively relating different system views and aspects in both design and analysis. In particular the framework supports a rigorous mediation between the architectural system view and the detailed view of the components out of which the system is built.

Based upon the extension of FSMs by ASMs built from FSM components, *model-checking* can be applied to the FSM components, as has been done using the connection of the SMV model checker to the ASM Workbench reported in [49, 25, 50]. To the global ASM, which describes the overall system built from the components, one can apply *theorem proving*, using as assumptions the model-checked local component properties. In addition *experimental validation* by simulation and testing can be applied to the high-level ASM as well as to its lower level components, due to the (mental as well as machine supported) executability of ASMs.

In Section 2 we explain how the abstraction and refinement mechanism coming with ASMs provides the flexibility that is needed for the navigation between different system levels and for a coherent combination of different system aspects. In Section 3 we show that a powerful component concept derives from a natural ASM submachine concept that unifies the architectural and the component level system view. It is based upon submachine replacement and refinement and provides various widely used composition schemes. For the benefit of the reader who does not know the notion of ASMs, in Section 2.1 we also provide a working definition which shows how ASMs capture in a natural and simple way the basic intuition of the concept of Virtual Machine. For further details, including numerous modeling and verification case studies and complete historical and bibliographical references, we refer to the AsmBook [22][1].

2 Relating high-level and component-level system views

To effectively and reliably link the architectural system view to its component-level view, and in general high-level models to more detailed ones, one needs two things:

- a uniform conceptual framework (read: a sufficiently general language) to express the notions involved at different levels of abstraction,
- a flexible method to navigate between the multiple levels of detailing.

Concerning the expressivity, from the short review we provide in Section 2.1 for the language of ASMs it should become clear that its few and simple basic constituents are general enough to integrate a great variety of useful system design and analysis patterns, leading from the requirements capture level down to the level of executable code. It is crucial that with the ASM method the same language can be used to define and analyse the various constituents of models, at all levels:

[1] Some of the material (in particular slides for lectures on the themes treated in the book) can also be downloaded from the AsmBook website at http://www.di.unipi.it/AsmBook/.

the rules for the dynamics, whether for the global system or for its components, as well as the auxiliary procedures or functions on the variety of underlying data structures. In Section 2.2 we explain how the ASM refinement method allows one to cross those multiple abstraction levels in a way that coherently and organically links them and makes these connections documentable.

2.1 The language of ASMs

The language of ASMs is the language of mathematics, the language par excellence of any scientific discourse. It is the same language where ASMs and their constituents are defined and analysed, using appropriate notational conventions to achieve simplicity. As machines exhibiting dynamic behavior ASMs are as general as virtual machines and algorithms can be, due to their definition as transitions systems that transform abstract states. In fact the *states* of ASMs are arbitrary mathematical *structures*: data are abstract objects (read: elements of sets or instances of classes) equipped with basic operations and predicates (attributes or relations). A familiar view of such structures is to treat them as given by tables. The entries of such tables are called *locations* and come as pairs of a function or predicate name f and an argument $(v_1, \ldots, v_n)$, which is formed by a list of parameter values v_i of whatever types. These locations represent the abstract ASM concept of basic object containers (memory units), which abstracts from any specific memory addressing and object referencing mechanism.

Location-value pairs (loc, v) are called *updates* and represent the basic units of state change in ASM computations. In fact ASMs transform abstract states by multiple simultaneous conditional updates that represent control-structure-free "If-Then" directives, the most general form of virtual machine instructions. Technically speaking these instructions come as finite set of ordinary transition *rules* of the following general form

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

where the guard *Condition* under which a rule is applied is an arbitrary expression evaluating to *true* or *false*. *Updates* is a finite set of assignments of the form

$$f(t_1, \ldots, t_n) := t$$

whose simultaneous execution is to be understood as redefining the values of the indicated functions f at the indicated arguments to the indicated values. More precisely in the given state, for each rule with true guard, first all parameters t_i, t are evaluated to their values, say v_i, v, then the location $(f, (v_1, \ldots, v_n))$ is updated to v, which represents the value of $f(v_1, \ldots, v_n)$ in the next state.

Thus ASMs represent a form of "pseudo-code over abstract data" where the instructions are guarded function updates. The abstract understanding of memory and memory update allows the application domain modeler or the system designer to combine the operational nature of the concepts of location and update with the freedom of tailoring them to the level of detail which is appropriate for the given design or analysis task. The simultaneous execution of multiple updates provides a rather useful instrument for high-level design to *locally describe a global state change*, namely as obtained in one step through executing a set of updates of some locations. The local description of global state change implies that by definition the next state differs from the previous state only at locations which appear in the update set. This basic parallel ASM execution model easens the specification of macro steps (using refinement and modularization as explained below), it avoids unnecessary sequentialization of independent actions and it helps to develop parallel or distributed implementations. The synchronous parallelism is enhanced by a notation for the simultaneous execution of a rule R for each x satisfying a given condition ϕ:

forall x **with** ϕ
  R

We illustrate this here by the ASM rule defined in [14] for the Occam instruction that spans subprocesses: in one step the currently running process a creates k new subprocesses, activates them and puts itself to sleeping mode, where the process activation provides the current environment and positions each subagent upon the code it has to execute. In this example we use the object-oriented notation to denote the instantiation of some of the state functions.

OccamParSpawn =
  if $a.mode = running$ **and** $instr(a.pos) = par(a, k)$ **then**
    forall $1 \leq i \leq k$ **let** $b = new(Agent)$ **in**
      Activate(b, a, i)
      $parent(b) := a$
    $a.mode := idle$
    $a.pos := next(a.pos)$
  where Activate(b, a, i) =
    $\{b.env := a.env,\ b.pos := pos(a, i),\ b.mode := running\}$

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

choose x **with** ϕ
  R

where ϕ is a Boolean-valued expression and R is a rule. The meaning of such an ASM rule is to execute rule R with an arbitrary x chosen among those satisfying the selection property ϕ.

In defining an ASM one is free to choose the abstraction level and the complexity and the means of definition of the auxiliary functions, which are used to compute locations and values in function updates. The following standard terminology, which is illustrated by Fig. 2 and is known as classification of ASM functions, names the different roles these functions can assume in a given machine M and provides a strong support to modular system development.

Static functions never change during any run of M so that their values for given arguments do not depend on the states of M, whereas *dynamic* functions may change as a consequence of updates by M or by the environment, so that their values for given arguments may depend on the states of M. Defining the static functions is independent from the description of the system dynamics. This supports to separate the treatment of the statics of a system from that of its dynamic behavior to reduce the complexity of the overall development task. Furthermore, whether the meaning of these functions is determined by a mere signature ("interface") description, or by axiomatic constraints, or by an abstract specification, or by an explicit or recursive definition, or by a program module, depends on the degree of information-hiding the specifier wants to realize.

Dynamic functions can be thought of as a generalization of array variables or hash tables. They are divided into four subclasses, depending on whether the machine or its environment (in general: other agents) update the function in question. *Controlled* functions (of M) are dynamic functions which are directly updatable by and only by M, i.e. functions f which appear in at least one rule of M as the leftmost function (namely in an update $f(s) := t$ for some s, t) and are not updatable by the environment. These functions are the ones which constitute the internally controlled part of the dynamic state of M. *Monitored* functions are dynamic functions which are read but not updated by M and directly updatable only by the environment. They appear in updates of M,

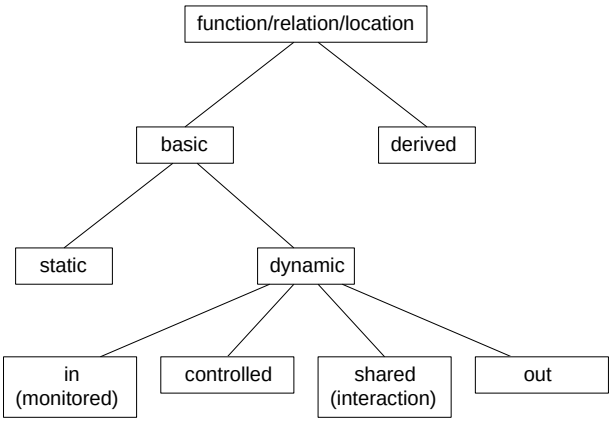

Figure 2: Classification of ASM functions, relations, locations

but not as the leftmost function of an update. These monitored functions constitute the externally controlled part of the dynamic state of M. To describe combinations of internal and external control of functions, *interaction* functions are used, also called *shared* functions and defined as dynamic functions which are directly updatable by the rules of M and by the environment and can be read by both. *Out* functions are dynamic functions which are updated but not read by M and are monitored by the environment.

The concepts of monitored, shared and out(put) functions allow one to separate in a specification the computation concerns from the communication concerns. The definition does not commit to any particular mechanism (e.g. message passing via channels) to describe the exchange of information between an agent and its environment. The only (but crucial) assumption made is that in a given state the values of all functions are determined.

Another pragmatically important distinction is that between *basic* and *derived* functions. Basic functions are functions which are taken for granted (declared as "given"); derived functions are functions which even if dynamic are not updatable either by M or by the environment, but may be read by both and yield values which are defined by a fixed scheme in terms of other functions. Thus derived functions are a kind of global method with read-only variables; to orthogonalize system descriptions one can specify and analyze them separately from the main machine.

Many other system design patterns can be conveniently integrated into ASMs via the two above classifications of functions or analogously locations (see [13] for details). For example if an object-oriented notation is desired, one can naturally support it by parameterizing functions $f(x)$ (similarly for rules representing methods) by a parameter a denoting an object (or an agent), thus creating 'instances' $\lambda x\ f_a(x)$ of f. As example see the relativization of the state functions *mode*, *pos*, *env* in the machine OccamParSpawn above. An illustration for introducing rule instances is given below (FIFO-buffer example in Section 3).

It is pragmatically important that the definitions are all phrased in terms of basic mathematical or algorithmic notions and thus can be grasped by every system engineer, relying only upon fundamental operational concepts of computing that are familiar from programmming practice, independently from the knowledge of any specific computational platform or programmming/design language or underlying logic. However, also a detailed formal definition of the semantics of ASMs, which is useful as basis for advanced analysis methods and for the development of well-documented tool support for ASMs, is available in textbook form in the AsmBook [22, Ch.2]. It is formulated there in terms of a first-order-logic-based derivation system.

2.2 Navigation between levels of detail

The method of ASMs exploits various means to provide smooth but rigorous links between different levels of detail in a system description. As explained in the previous section, already the language of ASMs via the function classification supports to separately describe the statics of a system and its dynamics[2], similarly to separately specify the internal behavior of a component and its interface to the environment—and to link them together in the overall ASM model. It also supports to smoothly incorporate into a system model specific language constructs, e.g. procedural or purely functional or logico-axiomatic descriptions or even full-fledged programs of a given design or programming language, all of which can be given separately from the ASM rules where they are used.

The natural companion of abstraction is refinement. Thus it should not come as a surprise that in the same way in which ASMs provide maximal freedom of abstraction, as explained above, they also provide a *most general notion of refinement*, a kind of meta-framework which can be tailored to given practical needs of componentwise system development. This framework integrates well-known more specific notions of refinement which have been defined in the literature (see [11, 39] for details). More importantly it supports to pass in a well-documented and checkable manner from an abstract model to a more detailed model, in a chain of such models, thus linking the outcome of the different phases of system design and making these links communicatable to the various experts involved. Here is a list of the main levels and experts involved.

- *Different levels of precision for accurate high-level human understanding* of a system. The language of ASMs allows one to fine-tune the models to the needed degree of rigour. Numer-

[2]The C-based *XASM* system [1] has been used in [2, 3, 4] to support a uniform machine-executable ASM-specification of the static and dynamic semantics of programming languages, and similarly in an architecture and compiler co-generation project [48].

ous experts are involved who speak different languages, think at different levels of abstraction and yet have to come to a common understanding of a system:

  - Application domain expert and requirements engineer. The natural interpretation of ASMs as control-structure-free system of "If-Then" directives supports their successful use for building *ASM ground models* which allow one to rigorously capture system requirements and help ensuring a correct understanding of the resulting model by all parties involved. Since some stakeholders, for example the application domain experts, are usually not familiar with the principles and concepts of software design, it is crucial that the language of ASMs permits to formulate ground models in terms of the application domain so that one can recognize in the model the structure of the real-world problem. The language of ASMs allows one to combine in a uniform framework data model features, which are typically of conceptual and application oriented nature, with functional features defining the system dynamics and with user-interface features defining the communication between the data and functional model and neighboring systems or applications. The systematic representation of the requirements in ASM ground models provides a basis for requirements inspection and thus makes the correctness and the completeness of the requirements checkable for all the stakeholders. Due to the rigorous character of ASM models this may include the verification of overall system properties. Due to the executable character of ASMs, via ASM ground models one also obtains a possibility for early system validation, typically by simulation of user scenarios or of components. In addition, the frugal character of the ASM language helps to cope effectively with the continuous change of requirements by adaptations of the ground model abstractions, see [10].
  - Requirements engineer and system designer. Here ground models play the role of an accurate system blueprint where all the technical essentials of the software system to be built are layed down and documented as basis for the high-level design decisions.
  - System designer and programmer. The component definition in Section 3 shows how ASMs mediate between the overall system view defined by the designer and the local view of model-checkable components developed by the programmer.
  - System designer or programmer and tester or maintenance expert. The hierarchy of ASM models leading from the ground model to code allows one to identify the place where something may or did go wrong or where a desired extension can be realized appropriately. From the sequence of models and from the descriptions of model-based runtime assertions appearing in the test reports one can read off the relevant design features to locate bugs or trace versioning effects.
  - System designer and system user. The abstractions built into ASM ground models help to provide an understanding of the overall functionality of the system, to avoid erroneous sytem use.

- *Human understanding and implementation.* These two levels can be linked by an organic, effectively maintainable refinement chain of rigorous coherent models. At each level the divide and conquer method can be applied to prove a system property for the detailed model, namely by proving it from appropriate assumptions in the abstract model and showing that the refinement is correct and satisfies those assumptions. For this the practicality of the ASM refinement method is crucial, namely for faithfully reflecting an intended design decision (or reengineering idea) and for justifying its correct implementation in the refined model. The practicality stems from the generality and flexibility of the ASM refinement notion, as explained in [11]. By providing this possibility the ASM method fills a gap in the UML framework.
- *Design and analysis.* This well-known separation of different concerns ("You can analyze only what has been defined already") helps not to restrict the design space or its structuring into components by proof principles which are coupled to the design framework in a fixed a priori defined way. The ASM method permits to apply any appropriate proof principle (see the discussion below) once the model is defined. A typical class of examples where this distinction is crucial is the rigorous definition of language or platform standards. An outstanding ASM model for such a language standard is the one built for SDL-2000 in [31]. Another example is provided by the ASM model for C# defined in [15, 44] to reflect as much as possible the intuitions and design decisions underlying the language as described in the

ECMA standard [28] and in [33]. An example for the combination of an ASM definition and of its detailed mathematical analysis has been developed in [46] for Java and its implementation on the Java Virtual Machine. The model is the basis for a detailed mathematical analysis, including proofs that Java is type-safe, that the compiler is correct and complete and that the bytecode verifier is complete and sound.

- *Different analysis types and levels.* Such a distinction is widely accepted for system design levels and methods, but only rarely is it realized that it also applies to system analysis. The language of ASMs allows one to calibrate the degree of precision with respect to the needs of the application, whether data oriented (e.g. using the entity relationship model) or function oriented (e.g. using flow diagrams) or control oriented (e.g. using automata of various kinds). Here are the major levels of analysis the ASM method allows one to treat separately and to combine their results where needed.
  - Experimental *validation* (system simulation and testing) and mathematical *verification*. For example the *ASM Workbench* [24] has been extensively used in an industrial reengineering project at Siemens [18] for testing and user-scenario simulation in an ASM ground model. The *AsmGofer* system [41, 42] has been used in connection with the ASM models for Java and the JVM in [46] for testing Java/JVM language constructs. In a similar way the .NET-executable *AsmL* engine [29] is used at Microsoft Research for ground model validation purposes. The possibility of combining where appropriate validation with verification in a uniform framework helps not to be at the mercy of only testing.
  - *Different verification levels* and the characteristic concerns each of it comes with. Each verification layer has an established degree of to-be-provided detail, formulated in an appropriate language. E.g. reasoning for human inspection (design justification by mathematical proofs) requires different features than using rule-based reasoning systems (mechanical design justification). Mathematical proofs may come in the form of proof ideas or proof sketches (e.g. for ground model properties like in [7, 17, 19]) or as completely carried out mathematical proofs (see for example the stepwise verification of a mutual exclusion protocol [16] or of the compilation of Occam programs to Transputer code [14] that is split into 16 refinement proofs). Formalized proofs are based on inference calculi which may be operated by humans (see for example various logics for ASMs which have been developed in [32, 43, 38, 30, 45, 37]) or as computerized systems (see for example the KIV verification [40, 39] of the ASM-based WAM-correctness proof in [20] or the ASM-based PVS-verification of compiler back-ends in [26, 27, 30]). For mechanical verification one has to distinguish interactive systems (theorem proving systems like PVS, HOL, Isabelle, KIV) from automatic tools (model checkers and theorem provers of the Otter type). Each verification or validation technique comes with its characteristic implications for the degree of detail needed for the underlying specification, for the language to be used for its formulation and for the cost of the verification effort. Importantly, all these techniques are integratable into the ASM framework.

We resume the above explanations by Fig. 3 which is taken from [19] and illustrates the iterative process to link ASM specifications to code via model-supported stepwise design.

3 Submachine-based component concept

We present in this section a component concept for ASMs that goes beyond the concept of components of an asynchronous ASMs. In asynchronous ASMs multiple synchronous ASM components are put together to a globally asynchronous system with partial order runs or interleaving runs instead of sequential runs. A simple example is given in Section 3.3, more involved examples are provided by the class of globally asynchronous, locally synchronous Codesign-FSMs [35] mentioned in the introduction. In Section 3.1 we extend this notion of 'ASM component' by providing some widely used operators to compose ASMs out of submachines. These operators maintain the atomic-action-view which is characteristic for the synchronous parallelism of basic ASMs. Other operators allow one to build ASMs out of components with structured or durative 'actions' and are surveyed in Section 3.2.

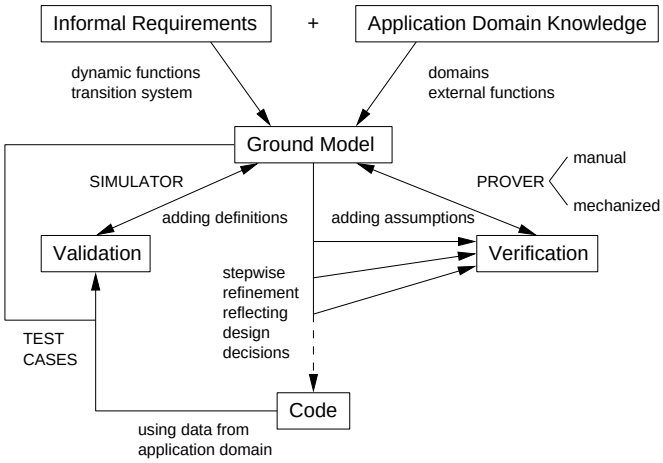

Figure 3: Models and methods in the development process

3.1 Operators for the Composition of Components

Since the dynamics of ASMs is defined by rules, the most general form of an ASM component is that of a rule[3]. Therefore the most general form of composition of a complex ASM out of a simpler one is by *rule replacement*, namely of a (simpler) *rule* occurring as subrule of say M by another (more complex) $rule'$. The result of such a substitution of $rule'$ for *rule* in M is written $M(rule'/rule)$. The ASM method leaves the freedom to the designer to impose further restrictions on the form of the component rules or on their use, guided by the needs of the particular application. We survey some examples in Section 3.2.

We discuss now some particularly frequent special cases of rule replacement which are used to build complex machines out of components.

Macro refinement. A typical use of rule replacement is made when in a sequence of stepwise model refinements it comes to *replace a macro definition* by a new one. For this case we use the following special notation where $Redef(macro)$ stands for the underlying new definition of *macro*.

$$M(\textbf{redef } macro) = M(Redef(macro)/macro)$$

Rule refinement. Another frequent use of rule replacement occurs upon the *refinement of an abstract rule by a concrete ASM*, a generalization of macro refinement in case the macro is a rule. For example the abstract submachines which appear in Fig. 1 can be defined in more detail as independent modules, see [36]. Another example can be found in the sequence of stepwise refined models for the Java Virtual Machine in [46]. There, one ASM defines a diligent JVM by adding to a bytecode interpreter component *trustfulVM* a bytecode verifier component, which calls method per method an abstract rule *verifyVM*, as illustrated in Fig. 4.

The abstract rule *verifyVM* is then refined by a submachine *verifyVM* which successively checks every instruction to satisfy the necessary type conditions and in the positive case propagates them to all successor instructions, using three components *check, propagate, succ* as illustrated in Fig. 5. For this form of refinement one can reuse the notation introduced for macros, writing **redef** $M = Redef(M)$ where $Redef(M)$ stands for the refinement of M.

Adding/deleting rules. Another common case of submachine replacements is the *addition or deletion of rules*. Due to the synchronous parallelism of ASMs no special notation is needed for adding rules, whereas for the deletion of R from M we write M **minus** R. Adding new rules is characteristic of purely incremental (also called conservative) refinements. A recent example is provided in [15, 44] to structure C# into layered components of by and large orthogonal language features, similarly to the decomposition in [46] of Java and the JVM into a hierarchy of components, as illustrated for the components *propagateVM*, *check*, *succ* of the JVM verifier in Fig.5. The model for the entire language is a parallel composition of all submachines. This incremental system development allows one to successivly introduce the following language layers, a feature one can

[3]This is analogous to the situation in TLA^+ [34] where a specification is given by a formula in which components are represented by subexpressions.

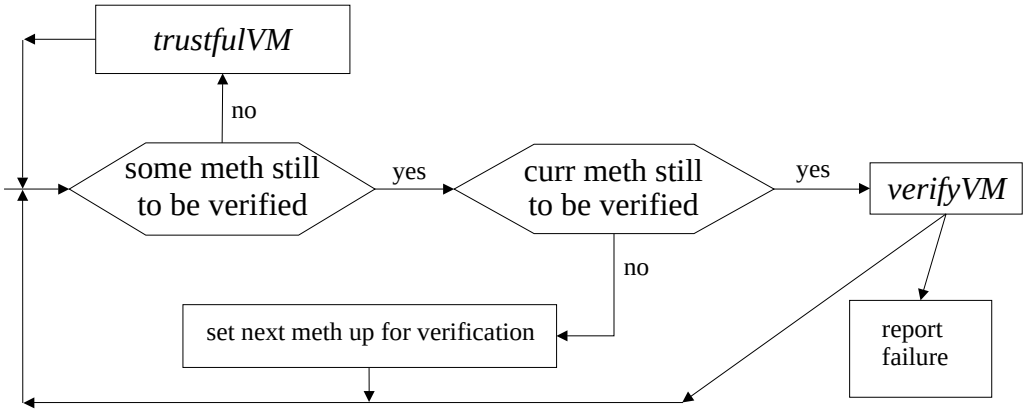

Figure 4: Building diligent JVM from components trustfulVM and verifyVM

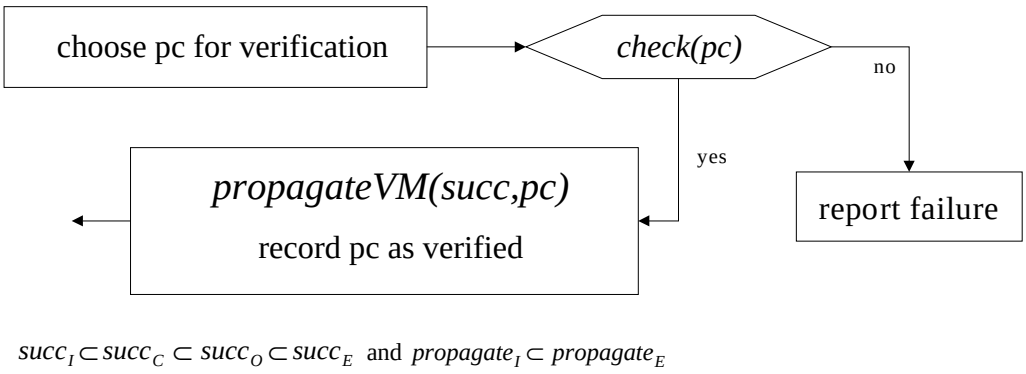

Figure 5: Building verifyVM from components propagateVM, check, succ

exploit not only for a piecemeal validation and verification of the entire language (as done for Java and its implementation on the JVM in [46]), but also for a systematic approach to teaching a complex programming language to beginners (see [12]):

- *imperative core*, related to sequential control by while programs, built from statements and expressions over the simple types of C#,
- *static class features* realizing procedural abstraction with class initialization and global (module) variables,
- *object-orientation* with class instances, instance methods, inheritance,
- *exception handling*,
- *concurrency*,
- *delegates* together with events (including for convenience here also properties, indexers, attributes),
- *unsafe* code with *pointer arithmetic*.

Adding guards/updates. The refinement of rules is often defined at the level of the two constituents of rules, namely guards and updates. For a machine which executes a given machine M only if a certain guard condition G is satisfied we write

addGuard G **to** $M =$ **if** G **then** M

For adding a set of updates to the updates of a rule in 'normal form' **if** *Cond* **then** *Updates* we write as follows:

M **addUpd** $U =$
  let $M =$ **if** *Cond* **then** *Updates* **in**

if *Cond* **then**
  Updates
  U

In Section 3.3 we illustrate the use of these two operators by a small example for a componentwise system definition.

3.2 Specific ASM component concepts

In this section we briefly review some more specific component concepts which have been defined in terms of ASMs and have been implemented using some of the major systems for the execution of large classes of ASMs.

All the operators defined in Section 3.1 maintain the atomic-action-view which is characteristic for the synchronous parallelism of basic ASMs. In [21] three fundamental standard operators are defined for ASMs with non-atomic behavior, namely sequential composition, iteration and recursive submachines. These three concepts have been implemented in the publicly available system AsmGofer [41] for executing ASMs. The definitions guarantee that one can build global ASMs, out of component ASMs, whose synchronous parallel computation may be viewed both in a global atomic way and in a local non-atomic way. For each of these three operators, the global view of the complex machine treats the component as a black-box that performs an atomic action. The global properties can be formulated on the basis of the analysis the local view allows one to perform, namely by looking in a white-box manner at the internal details of the non-atomic (iterative or otherwise durative) behavior of the component. This double view allows one to perform a hierarchical system development, where the model checking of the components is the basis for a global system analysis by mathematical proof and experimental validation (simulation) techniques.

In [47] an ASM component concept is defined which treats components in terms of the services they export or import.

The open source XASM tool [1] for the execution of ASMs is built around a component-based C-like language. It has been used in [4] for the implementation of component interaction. It also implements the above mentioned notion of sequential submachines. However, it provides no support for hierarchical modeling.

An specific ASM component concept that follows the pattern of Mealy automata is defined in [42, Ch.2] and has been applied for the high-level verification of VHDL-based hardware design. It comes with a component-based verification technique that has been used in two industrial projects at Siemens reported in [42]. Roughly speaking, a component is defined there as a basic ASM whose rule may contain submachine calls and whose interface consists of VHDL-like in/out functions; they are used for providing input respectively output at successive computation steps to compute input/output behaviour in the way known from Mealy FSMs. Composition of such components is defined not by name sharing of input and output functions, but by connecting inputs and outputs: several inputs may get the same input value; a connected input is connected either with an ouput or with an input which in the transitive closure is not connected to the original input (so that one can determine for connected inputs the input value determining source).

The tool AsmL [29] developed at Microsoft is used in [6, 5] to implement on the .NET platform behavioral interface specifications by ASMs, including component interfaces, and to test COM components against these specifications.

3.3 Componentwise system development: an example

We illustrate the above two operators for adding guards and updates by a small example taken from [34, Ch.3]. We use the occasion to also show the above mentioned painless introduction of object-oriented notation by parameterization of rules. The example deals with a stepwise definition of a bounded FIFO buffer from a basic component for a 2-phase handshake protocol.

The first step consists in defining this protocol to transmit a data *val*ue from a sender to a receiver who has to acknowledge the receipt before the next data value can be sent. Sending is possible only if *ready* = *ack* (first handshaking), the action includes flipping the boolean variable *ready*. An acknowledgement can be made only if *ready* = *ack* is false (second handshaking), the action includes flipping the boolean variable *ack*. This is expressed by the following ASM which

defines a channel with rules for sending and receiving data, using a submachine FLIP to flip the value of a boolean variable. The sending rule is parameterized by the data to be sent.

CHANNEL = {SEND, RECEIVE} **where**
  SEND(d) = **if** *ready* = *ack* **then**
    FLIP(*ready*)
    val := d
  RECEIVE = **if** *ready* ≠ *ack* **then** FLIP(*ack*)

The second step consists in parameterizing this handshake protocol machine by an agent **self**, which can be instantiated to produce independent copies of the machine. Since the state of CHANNEL is made out of the three 0-ary functions *ready*, *ack*, *val*, also these functions have to be relativized to **self**. This comes up to replace CHANNEL by **self** .CHANNEL, which in ordinary mathematical notation is written CHANNEL(**self**); similarly for SEND, RECEIVE and the functions *ready*, *ack*, *val*.

The third step consists in defining an unbounded FIFO-buffer from two instances of CHANNEL as its components, one for placing an input into the queue and one for emitting an output from the queue. Since appending an element to the queue and deleting from it an element are largely independent from each other, we want to avoid a possibly premature design decision about the particular scheduling for these two operations. As a consequence the FIFO-buffer is formulated by an asynchronous ASM, which consists of two synchronous ASM components built from instances of CHANNEL: the instantiation FIFOBUFFERIN by an agent *in* appends elements to the queue, the instantiation FIFOBUFFEROUT by an agent *out* deletes elements from the queue.

When an element is sent that should be put into the buffer, upon acknowledging the receipt channel agent *in* has to append the received value to the (tail of the) queue. This is achieved by extending the set of updates performed by *in*.RECEIVE.

FIFOBUFFERIN = {*in*.SEND, BUFRCV} **where**
  BUFRCV = *in*.RECEIVE **addUpd** APPEND(*in.val*, *queue*)

When an element is sent as output from the buffer, channel agent *out* has also to delete the head element from the queue. This is achieved by adding an update to *out*.SEND. In addition this operation should only be performed when the queue is not empty, so that we also add an additional guard to *out*.SEND to check this property.

FIFOBUFFEROUT = {BUFSEND, *out*.RECEIVE} **where**
  BUFSEND = **addGuard** *queue* ≠ *empty* **to**
    out.SEND(*head*(q)) **addUpd** DELETE(*head*(*queue*), *queue*)

These two ASMs may operate asynchronously, forming an asynchronous ASM FIFOBUFFER whose components share the common data structure *queue*. They could also be combined by interleaving, as is done in [34, Ch.3]), resulting in the following interleaved FIFOBUFFER version.

FIFOBUFFER = FIFOBUFFERIN **or** FIFOBUFFEROUT

The operator **or** for the non-deterministic choice among rules $R(i)$ is defined as follows[4]:

$R(0)$ **or** ... **or** $R(n-1)$ = **choose** $i < n$ **do** $R(i)$

The fourth and last step of our illustration of stepwise building a machine out of components consists in refining the unbounded buffer into a bounded one with queue of maximal length N. It suffices to substitute for the rule BUFRCV its refinement by the additional guard *length*(*queue*) < N, thus preventing its application when the buffer is full.

BOUNDEDFIFOBUFFER(N) =
  FIFOBUFFER(**addGuard** *length*(*queue*) < N **to** BUFRCV/BUFRCV)

[4]For further discussion of how to combine process algebra constructs with ASMs see [8].

4 Conclusion

We have illustrated how the ASM method can help to bridge the gap between specification and design by rigorous high-level (hw/sw co-) modeling which is linked seamlessly to executable code, in a way the practitioner can verify and validate. We have explained how based upon the clear notions of ASM state and state transition and their refinements, system verification and validation at different levels of abstraction and rigor can be combined by the ASM method in a uniform way. We have defined some ASM composition operators which allow one to combine, in design and analysis, the global system view with the local view of the components. In particular this allows one to verify overall system features by theorem proving methods which exploit the model-checked properties of the components.

References

[1] M. Anlauff and P. Kutter. Xasm Open Source. Web pages at http://www.xasm.org/, 2001.

[2] M. Anlauff, P. Kutter, and A. Pierantonio. Montages/Gem-Mex: a meta visual programming generator. TIK-Report 35, ETH Zürich, Switzerland, 1998.

[3] M. Anlauff, P. Kutter, and A. Pierantonio. Enhanced control flow graphs in Montages. In D. Bjoerner and M. Broy, editors, *Proc. Perspectives of System Informatics (PSI'99)*, volume 1755 of *Lecture Notes in Computer Science*, pages 40–53. Springer-Verlag, 1999.

[4] M. Anlauff, P. Kutter, A. Pierantonio, and A. Sünbül. Using domain-specific languages for the realization of component composition. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering (FASE 2000)*, volume 1783 of *LNCS*, pages 112–126, 2000.

[5] M. Barnett and W. Schulte. The ABCs of specification: AsmL, behavior, and components. *Informatica*, 25(4):517–526, 2002.

[6] M. Barnett and W. Schulte. Contracts, components and their runtime verification on the .NET platform. *J. Systems and Software*, Special Issue on Component-Based Software Engineering, 2002.

[7] C. Beierle, E. Börger, I. Durdanović, U. Glässer, and E. Riccobene. Refining abstract machine specifications of the steam boiler control to well documented executable code. In J.-R. Abrial, E. Börger, and H. Langmaack, editors, *Formal Methods for Industrial Applications. Specifying and Programming the Steam-Boiler Control*, number 1165 in LNCS, pages 62–78. Springer, 1996.

[8] 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 *Lecture Notes in Computer Science*, pages 22–32. Springer-Verlag, 2003.

[9] E. Börger. High-level system design and analysis using Abstract State Machines. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Current Trends in Applied Formal Methods (FM-Trends 98)*, volume 1641 of *LNCS*, pages 1–43. Springer-Verlag, 1999.

[10] E. Börger. The ASM ground model method as a foundation of requirements engineering. In N.Dershowitz, editor, *Manna-Symposium*, volume 2772 of *LNCS*. Springer-Verlag, 2003.

[11] E. Börger. The ASM refinement method. *Formal Aspects of Computing*, 14, 2003.

[12] E. Börger. Teaching ASMs to practice-oriented students. In *Teaching Formal Methods Workshop*, pages 5–12. Oxford Brookes University, 2003.

[13] E. Börger. Abstract State Machines: A unifying view of models of computation and of system design frameworks. *Annals of Pure and Applied Logic*, 2003, to appear.

[14] E. Börger and I. Durdanović. Correctness of compiling Occam to Transputer code. *Computer Journal*, 39(1):52–92, 1996.

[15] E. Börger, G. Fruja, V. Gervasi, and R. Stärk. A complete formal definition of the semantics of C#. Technical report, In preparation, 2003.

[16] E. Börger, Y. Gurevich, and D. Rosenzweig. The bakery algorithm: Yet another specification and verification. In E. Börger, editor, *Specification and Validation Methods*, pages 231–243. Oxford University Press, 1995.

[17] E. Börger and L. Mearelli. Integrating ASMs into the software development life cycle. *J. Universal Computer Science*, 3(5):603–665, 1997.

[18] E. Börger, P. Päppinghaus, and J. Schmid. Report on a practical application of ASMs in software design. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines: Theory and Applications*, volume 1912 of *LNCS*, pages 361–366. Springer-Verlag, 2000.

[19] E. Börger, E. Riccobene, and J. Schmid. Capturing requirements by Abstract State Machines: The light control case study. *J. Universal Computer Science*, 6(7):597–620, 2000.

[20] E. Börger and D. Rosenzweig. The WAM – definition and compiler correctness. In C. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, volume 11 of *Studies in Computer Science and Artificial Intelligence*, chapter 2, pages 20–90. North-Holland, 1995.

[21] E. Börger and J. Schmid. Composition and submachine concepts for sequential ASMs. In P. Clote and H. Schwichtenberg, editors, *Computer Science Logic (Proceedings of CSL 2000)*, volume 1862 of *Lecture Notes in Computer Science*, pages 41–60. Springer-Verlag, 2000.

[22] E. Börger and R. F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003.

[23] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[24] 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.

[25] G. Del Castillo and K. Winter. Model checking support for the ASM high-level language. In S. Graf and M. Schwartzbach, editors, *Proc. 6th Int. Conf. TACAS 2000*, volume 1785 of *Lecture Notes in Computer Science*, pages 331–346. Springer-Verlag, 2000.

[26] A. Dold. A formal representation of Abstract State Machines using PVS. Verifix Technical Report Ulm/6.2, Universität Ulm, Germany, July 1998.

[27] A. Dold, T. Gaul, V. Vialard, and W. Zimmermann. ASM-based mechanized verification of compiler back-ends. In U. Glässer and P. Schmitt, editors, *Proc. 5th Int. Workshop on Abstract State Machines*, pages 50–67. Magdeburg University, 1998.

[28] C# Language Specification. Standard ECMA–334, 2001. http://www.ecma-international.org/publications/standards/ECMA-334.HTM.

[29] Foundations of Software Engineering Group, Microsoft Research. AsmL. Web pages at http://research.microsoft.com/foundations/AsmL/, 2001.

[30] A. Gargantini and E. Riccobene. Encoding Abstract State Machines in PVS. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines: Theory and Applications*, volume 1912 of *LNCS*, pages 303–322. Springer-Verlag, 2000.

[31] U. Glässer, R. Gotzhein, and A. Prinz. Formal semantics of sdl-2000: Status and perspectives. *Computer Networks*, 42(3):343–358, June 2003.

[32] R. Groenboom and G. R. Renardel de Lavalette. A formalization of evolving algebras. In S. Fischer and M. Trautwein, editors, *Proc. Accolade 95*, pages 17–28. Dutch Research School in Logic, ISBN 90-74795-31-5, 1995.

[33] A. Hejlsberg, S. Wiltamuth, and P. Golde. *C# Language Specification*. Addison-Wesley, 2003.

[34] L. Lamport. *Specifying Systems*. Addison-Wesley, 2003.

[35] L. Lavagno, A. Sangiovanni-Vincentelli, and E. M. Sentovitch. Models of computation for system design. In E. Börger, editor, *Architecture Design and Validation Methods*, pages 243–295. Springer-Verlag, 2000.

[36] W. Mueller, R. Dömer, and A. Gerstlauer. The formal execution semantics of SpecC. Technical Report TR ICS 01-59, Center for Embedded Computer Systems at the University of California at Irvine, 2001.

[37] S. Nanchen and R. F. Stärk. A security logic for Abstract State Machines. In *TR 423 CS Dept ETH Zürich*, 2003.

[38] A. Poetzsch-Heffter. Deriving partial correctness logics from evolving algebras. In B. Pehrson and I. Simon, editors, *IFIP 13th World Computer Congress*, volume I: Technology/Foundations, pages 434–439, Elsevier, Amsterdam, 1994.

[39] G. Schellhorn. Verification of ASM refinements using generalized forward simulation. *J. Universal Computer Science*, 7(11):952–979, 2001.

[40] G. Schellhorn and W. Ahrendt. Reasoning about Abstract State Machines: The WAM case study. *J. Universal Computer Science*, 3(4):377–413, 1997.

[41] J. Schmid. Executing ASM specifications with AsmGofer. Web pages at `http://www.tydo.de/AsmGofer`.

[42] J. Schmid. *Refinement and Implementation Techniques for Abstract State Machines*. PhD thesis, University of Ulm, Germany, 2002.

[43] A. Schönegge. Extending dynamic logic for reasoning about evolving algebras. Technical Report 49/95, Universität Karlsruhe, Fakultät für Informatik, Germany, 1995.

[44] R. F. Stärk and E. Börger. An ASM specification of C# threads and the .NET memory model. Technical report, Computer Science Department, ETH Zürich, 2003.

[45] R. F. Stärk and S. Nanchen. A logic for Abstract State Machines. *J. Universal Computer Science*, 7(11):981–1006, 2001.

[46] R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag, 2001. .

[47] A. Sünbül. *Architectural Design of Evolutionary Software Systems in Continuous Software Engineering*. PhD thesis, TU Berlin, Germany, 2001.

[48] 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.

[49] K. Winter. Model checking for Abstract State Machines. *J. Universal Computer Science*, 3(5):689–701, 1997.

[50] K. Winter. *Model Checking Abstract State Machines*. PhD thesis, Technical University of Berlin, Germany, 2001.