The Formal Execution Semantics of SpecC

Wolfgang Mueller
Paderborn University, Paderborn, Germany
wolfgang@acm.org

Rainer Dömer, Andreas Gerstlauer
University of California, Irvine, USA
doemer@ics.uci.edu, gerstl@ics.uci.edu

ABSTRACT

We present a rigorous but transparent semantics definition of the SpecC language that covers the execution of SpecC behaviors and their interaction with the kernel process. The semantics include `wait`, `waitfor`, `par`, and `try` statements as they are introduced in SpecC. We present our definition in form of distributed Abstract State Machine (ASM) rules strictly following the lines of the SpecC Language Reference Manual [4]. We mainly see our formal semantics in three application areas. First, it is a concise, unambiguous description for documentation and standardization. Second, it applies as a high–level, pseudo code–oriented specification for the implementation of a SpecC simulator. Finally, it is a first step for SpecC synthesis in order to identify similar concepts with other languages like VHDL and SystemC for the definition of common patterns and language subsets.

Categories and Subject Descriptors

F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Specification techniques; B.7.2 [**Design Aids**]: Simulation

General Terms

Systems Design, Specification Languages, Verification

Keywords

Formal Specifications, ASMs, SpecC, Simulation

1. INTRODUCTION

The SpecC language [5, 6] has been proposed as a standard system-level language for adoption in industry and academia and is promoted for standardization by the SpecC Technology Open Consortium (STOC). The SpecC language was specifically developed to address the issues involved with system design, including both software and hardware. Although the SpecC language is defined by a Language Reference Manual (LRM) [4], and comes with a freely available open source reference implementation, a compiler and a simulator, the precise meaning of the execution semantics has not been captured so far. However, a precise semantics of SpecC is mandatory for various applications including simulation, synthesis, and formal verification. If well written, it can be taken as a complementary, unambiguous documentation to significantly help understanding the language.

This article presents the formal semantics for SpecC execution. The SpecC semantics provides a concise definition of the complete execution semantics of SpecC V1.0 for potential standardization. This is an important step towards future SpecC compliant implementations and applications in various fields including formal verification. In synthesis and (co-)simulation, our formal semantics can be used as a sound basis to identify common behavioral concepts for interoperability with Verilog, VHDL, and SystemC.

We present a concise and rigorous but yet intuitive semantic definition of SpecC in terms of Gurevich's *distributed Abstract State Machines* (ASMs) [8]. Our ASM specification follows the terminology and outlines given in the SpecC LRM [4] and additionally correspond to the VHDL'93 semantics in [2]. We present a mathematical definition of SpecC in terms of a *SpecC Algebra* considering `wait`, `waitfor`, `notify`, `notifyone`, `par`, and `try` statements[1], as well as the complete interaction between the user defined behaviors and channels with the kernel process. We additionally outline how to derive a C++ implementation for a SpecC simulator from this specification and demonstrate how ASMs can be applied as a formal framework for the general specification and implementation of virtual machines such as simulators.

The remainder of this paper is organized as follows. Section 2 discusses related works. Section 3 briefly introduces distributed ASMs before Section 4 outlines basic conceptsof the SpecC language and defines its execution semantics by means of ASMs. Section 5 outlines how to derive an implementation from the specification. Section 6 finally closes with a conclusion and outlook.

2. RELATED WORKS

Over previous years, research in formal semantics in EDA mainly focused on VHDL. There exists numerous works based on temporal logics, functional semantics, denotational semantics, and operational semantics applying Boyer-Moore Logic, Process Algebras, Petri-Nets etc. [3]. Most of the approaches cover subsets dedicated for application in formal verification. In [3], the simulation semantics of complete behavioral VHDL 93 is introduced by Abstract State

[1]For space reasons, we omit the `pipe` statement here.

ISSS'02, October 2–4, 2002, Kyoto, Japan.

Machines [2]. which was later extended for VHDL-AMS in [11]. Other applications investigated VHDL-Verilog interoperability [10]. Most recently, SystemC simulation semantics have been published in [9] which is oriented towards the VHDL'93 definitions in [2].

ASMs have already been applied for formal specification in various other domains such as hardware and software architectures, protocols, and programming languages [1]. Examples for programming languages are semantics definitions of Java and C++[12]. Furthermore, the ITU standard SDL 2000 is partly underlined by an ASM definition [7].

All these investigations demonstrate that ASMs, i.e., distributed ASMs, have excellent capabilities to capture the behavioral semantics of programming and specification languages. This is particularly true for the specification of underlying virtual machines as required for the formal coverage of the SpecC simulator. In this article, we focus our investigations on SpecC V1.0 which is the latest official version at the time of writing. The model is defined along the lines of the basic concepts of the VHDL'93 and SystemC definitions in [2, 9] so that future work on interoperability with VHDL and SystemC is simplified.

3. ABSTRACT STATE MACHINES

Abstract State Machine (ASM) specifications can be understood as 'pseudocode over abstract data', without any particular theoretical prerequisites. Here, we list only the basic definitions and refer to [8] for a formal introduction.

An ASM specification comes in form of guarded function updates, called rules, of the form

$$\textbf{if } \mathit{Condition} \textbf{ then } <\mathit{Updates}> \textbf{ else } <\mathit{Updates}> \textbf{ endif}$$

Rules are are presented as nested if–then–else clauses with a set of function updates in their body. When executing the rules, the underlying ASM abstract machine executes state transitions with algebras as states. A state transition is performed by firing a set of rules in one step. Only those rules are fired whose guards ($\mathit{Condition}$) evaluate to true.

At each step, the guards evaluate to a set of function updates, each of the form $f(t_1, ..., t_r) := t_0$ where t_i are terms (including functions). Note that 0-ary functions play the role of *variables* known from imperative programming languages. A block is a set of function updates separated by a comma[2]. The individual function updates of each block are collected in a so–called update set and are are simultaneously executed in one step. Each function update changes a value at a specific location given by the left–hand–side of the update. Functions are considered to be global. Two or more simultaneous updates of the same location in one update set defines inconsistency. In the case of inconsistency no state transition is performed and no update in the update set is being executed.

ASMs are multi–sorted based on the notion of universes. We presume the standard mathematic universes of booleans, integers, lists, etc. as well as the standard operations on them without further mention. A universe can be dynamically extended with individual objects by

$$\textbf{extend } \mathit{Universe} \textbf{ with } v <\mathit{Rule}> \textbf{ endextend}$$

[2]In extension to [8], we use a comma to syntactically separate function updates. We also introduce sequential statements which are enclosed in braces and separate by semicolon, e.g., {C:=1; D:=C}. This a shortcut avoids the introduction of an additional state function with additional conditions.

where v is a variable which is bound by the **extend** constructor. As the inverse operation, a universe can be dynamically reduced with individual objects by

$$\textbf{reduce } \mathit{Universe} \textbf{ by } v <\mathit{Rule}> \textbf{ endreduce}$$

where v is a variable which is bound by the **reduce** constructor. The **choose** constructor defines an arbitrary selection of one element in a universe

$$\textbf{choose } v \textbf{ in } \mathit{Universe} <\mathit{Rule}> \textbf{ endchoose}$$

where v is non–deterministically selected from the given universe. The **choose** constructor can be qualified by a condition (**satisfying**). The **var** rule constructor defines the simultaneous instantiation of a rule:

$$\textbf{var } v \textbf{ ranges over } \mathit{Universe} <\mathit{Rule}> \textbf{ endvar}$$

Executing the constructor means to spawn and execute the rule for each element in $Universe$ simultaneously, i.e., the constructor basically spawns n rules where n is the number of elements in $Universe$.

The extension of basic ASMs to *distributed ASMs* partitions rules into modules where each module is given by its module name ν. A module is instantiated to execute by setting $Mod(a) := \nu$ for an agent a. The symbol $Self$ refers to a after the instantiation. The execution is defined by partially ordered state transitions where agents are asynchronously executed.

The SpecC algebra introduced in the next section comes in the form of two modules: One for the SpecC kernel and one for the user defined behaviors.

4. SpecC

The SpecC language [5, 6, 4] is based on ANSI-C and provides a set of additional constructs needed for modeling hardware. The added concepts include behavioral and structural hierarchy, concurrency, synchronization, exception handling, and timing. Since the execution semantics of ANSI-C are already well-defined, we focus in the following sections on the formal description of these added concepts.

4.1 Structure

A SpecC program consists of a set of *behaviors*, *channels*, and *interfaces* with *ports*. Behaviors are active blocks containing computation, whereas channels and interfaces are passive blocks encapsulating communication. Following the style of standard block diagrams, behaviors and channels are composed in form of a structural hierarchy. Thus, the basic structure of a SpecC model is a hierarchical network of behaviors and channels connected by ports. A simple example is depicted in Figure 1.

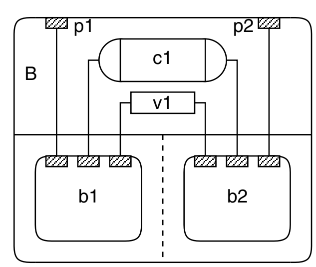

Figure 1: SpecC Example

For defining execution semantics, only the active behaviors need to be considered, the passive channels can be ignored or assumed to be inlined. In other words, in this paper, we can focus on the behavioral hierarchy of SpecC.

4.2 Execution Semantics

The next paragraphs describe the stepwise development of a formal execution semantics of SpecC V1.0, starting with the basic behavioral constructs. Afterwards, we present a formal definition of the kernel process.

4.2.1 Basic Concepts

Derived from hierarchically organized modules, SpecC establishes a hierarchical network of parallel communicating BEHAVIORs which, under the supervision of the distinguished SpecC kernel process, concurrently update new values for given $VARIABLEs$ and send and receive $EVENTs$.

After initialization of variables and program counters of $BEHAVIORs$, there is a mutually exclusive execution of the kernel process and the concurrently running behaviors. In other words, the kernel process periodically starts its execution if all behaviors are suspended, and vice versa.

Each user defined behavior is *running* until it is suspended, for example, *waiting* at a `wait` or `waitfor` statement. It will resume *running* when the kernel delivers notified events or increases the time due to an expired timeout. After executing the last statement, a behavior changes to *completed*. Furthermore, we use the state *interrupted* for behaviors that have received an event triggering an active interrupt handler. In summary, throughout the life cycle of a behavior b, we set $status(b) \in \{$*running, waiting, completed, interrupted*$\}$ (see Figure 2).

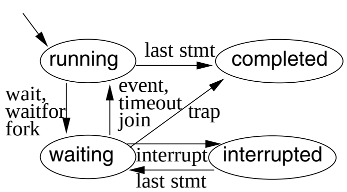

Figure 2: Life Cycle of a Behavior

When no user defined behaviors are *running*, i.e., all are *waiting*, *interrupted* or *completed*, the kernel process goes through a set of phases and resumes behaviors on events or timeouts, and advances the simulation time whenever necessary.

The rules in the following paragraphs constitute the program of ASM agents, one for the kernel process and one for each behavior. As previously outlined, agents are instantiations of ASM modules. Thus, we first define rules for the $KERNEL\_Module$. Thereafter, we define the semantics of distinguished statements executed in instantiations of the $BEHAVIOR\_Module$. For ASMs initialization, we set

$Mod(b) := BEHAVIOR\_Module$
$\forall b \in BEHAVIOR$ and
$Mod(k) := KERNEL\_Module$

for the kernel process $k \in KERNEL$ [3]. We further assume $phase = ResumeOnEvents$, current time $T_c = 0$, $status(b) = running$ and $pipe\_status(b) = init, \forall b \in BEHAVIOR$. Unless otherwise noted, all functions are assumed to be set $undef$ and all sets and lists are initially empty.

[3]The universe $KERNEL$ is introduced here for technical purpose and has only one element.

The remainder of this document first defines the execution semantics of specific SpecC statements. Thereafter, we define the execution cycle of the kernel.

4.2.2 SpecC Statements

Before we define the semantics for the SpecC statements, we need to discuss the role of the program pointer when processing a behavior during the simulation.

In order to focus on the essential behavioral semantics of SpecC, we basically assume that the continuation of the control–flow of each (sequential) behavior is determined by values of the function *programCounter* which is initially set to the first statement of each behavior b. After checking their current watching conditions, all *running* behaviors execute their statements. In order to express that a user defined behavior $Self$ can be executed only when it is *running* and the *programCounter* is assigned to the specific statement, we use the following abbreviation:

$$\begin{aligned}&Self\ executes\ \texttt{statement} \equiv\\ &\quad programCounter(Self) = \texttt{statement} \wedge\\ &\quad phase := ExecuteBehaviors \wedge\\ &\quad status(Self) = running\end{aligned}$$

After executing the last statement of a behavior b, the behavior completes and we set $status(b) := completed$. As a special case, when having completed the behavior of an interrupt handler (explained later), we additionally set the *status* of all descendent behaviors of the parent p, $b_i \in descendant(p)$, from *interrupted* back to *waiting*. The parent of a behavior is defined to be the behavior which has spawned the interrupt handler and its descendents are defined to be all child behaviors (and their children) which were also interrupted together with b.

Variable Assignment. Right-hand-side values in signal assignments are immediately assigned to the current value of variable v. Parallel write accesses to the *value* of a variable are allowable. Competing concurrent assignments to variables v are non-deterministically resolved and are individual to each implementation. We denote this by $resolve(competingValues(value(Expr)))$ which first computes the value of $Expr$ and stores it into a virtual data structure keeping all concurrent assignments. Finally, *resolve* selects non-deterministically one of these values. Note here that ASMs cannot directly express non-deterministic behavior so that we have to introduce the *resolve* function which models the non-deterministic behavior of concurrent write accesses to SpecC variables.

$$\begin{aligned}&\textbf{if}\ Self\ executes\ \langle v\ =\ Expr\rangle\\ &\textbf{then}\quad value(v) := resolve(competingValues(value(Expr))),\\ &\qquad\quad programCounter(Self) := nextStmt(Self)\\ &\textbf{endif}\end{aligned}$$

Wait Statement. On reaching a `wait` statement, a behavior simply stops execution by setting its *status* to *waiting*. The behavior also notes its *sensitivity* to the given list of events.

$$\begin{aligned}&\textbf{if}\ Self\ executes\ \langle\underline{\texttt{wait}}(EventList)\rangle\\ &\textbf{then}\quad status(Self) := waiting,\\ &\qquad\quad sensitivity(Self) := EventList,\\ &\qquad\quad programCounter(Self) := nextStmt(Self)\\ &\textbf{endif}\end{aligned}$$

Waitfor Statement. Similar to the `wait` statement, a behavior stops its execution by setting its *status* to *waiting* upon reaching a `waitfor` statement. However, instead of setting its *sensitivity*, it sets a *timeout* to the current time increased by the given delay. After the *timeout*, the behavior will then be resumed by the SpecC kernel.

$$\begin{array}{l} \textbf{if } \mathit{Self\ executes}\ \langle \underline{\texttt{waitfor}}(\mathit{Delay})\rangle \\ \textbf{then}\ \ \mathit{status}(\mathit{Self}) := \mathit{waiting}, \\ \quad\quad \mathit{timeout}(\mathit{Self}) := T_c + \mathit{Delay}, \\ \quad\quad \mathit{programCounter}(\mathit{Self}) := \mathit{nextStmt}(\mathit{Self}) \\ \textbf{endif} \end{array}$$

Notify Statement. At a `notify` statement, a behavior simply sets flags for all notified events and immediately proceeds to the next statement. Note that the notified events will be delivered later to any waiting behaviors by the SpecC kernel.

$$\begin{array}{l} \textbf{if } \mathit{Self\ executes}\ \langle \underline{\texttt{notify}}(\mathit{EventList})\rangle \\ \textbf{then}\ \ \forall e \in \mathit{EventList} : \mathit{notified}(e) := \mathit{true}, \\ \quad\quad \mathit{programCounter}(\mathit{Self}) := \mathit{nextStmt}(\mathit{Self}) \\ \textbf{endif} \end{array}$$

Notifyone Statement. Similar to the `notify` statement, a `notifyone` statement also records the notified events and proceeds its execution. Note that, in contrast to `notify`, event lists of all `notifyone` statements given in one execution cycle have to be managed by a global *notifiedonelist* which is organized as a list of event lists.

$$\begin{array}{l} \textbf{if } \mathit{Self\ executes}\ \langle \underline{\texttt{notifyone}}(\mathit{EventList})\rangle \\ \textbf{then}\ \ \mathit{notifiedonelist} := \mathit{notifiedonelist} + \mathit{Eventlist}, \\ \quad\quad \mathit{programCounter}(\mathit{Self}) := \mathit{nextStmt}(\mathit{Self}) \\ \textbf{endif} \end{array}$$

Par Statement. At a `par` statement, a behavior spawns a set of children and proceeds only after the children have terminated.

$$\begin{array}{l} \textbf{if } \mathit{Self\ executes}\ \langle \underline{\texttt{par}}\{b_1; ...; b_M\}\rangle \\ \textbf{then}\ \mathit{SPAWN}(\{b_1; ...; b_M\}, \mathit{Self}), \\ \quad\quad \mathit{programCounter}(\mathit{Self}) := \mathit{nextStmt}(\mathit{Self})\} \\ \textbf{endif} \end{array}$$

The process of spawning children consists of a fork and a join operation in sequential order. For better readability, these are defined as ASM macros as follows:

$$\begin{array}{l} \mathit{SPAWN}(\mathit{Blist}, \mathit{Self}) \equiv \\ \quad \{\mathit{FORK}(\mathit{Blist}, \mathit{Self}, \mathit{waiting}); \\ \quad\ \ \mathit{JOIN}(\mathit{Blist}, \mathit{Self}, \mathit{running})\} \end{array}$$

The fork operation extends the domain *BEHAVIOR* by the behaviors *b* which are forked. Each of the behaviors is set to *running*. For later purpose, the spawning behavior is noted as their *parent*. The list of all *b* is saved as *children* of the spawning behavior *Self*. The status of *Self* is set to *Status* which is *waiting* in the above case[4]. When all children are completed, it is reset to *running*.

$$\begin{array}{l} \mathit{FORK}(\mathit{Blist}, \mathit{Self}, \mathit{Status}) \equiv \\ \forall b \in \mathit{Blist} : \\ \textbf{extend } \mathit{BEHAVIOR} \textbf{ with } b \\ \quad \mathit{status}(b) = \mathit{running}, \quad \mathit{parent}(b) := \mathit{Self} \\ \textbf{endextend}\ , \\ \mathit{children}(\mathit{Self}) := \mathit{Blist}, \\ \mathit{status}(\mathit{Self}) := \mathit{Status} \end{array}$$

All children *b* have joined when their *status* is *completed*. Then the set of children of the parent *Self* is set empty and its new *Status* is assigned. In the context of the `par`-statement, the *Status* is set to *running* in order to continue execution. Additionally, the domain *BEHAVIOR* is reduced by the completed child behaviors. Note how the domain *BEHAVIOR* dynamically increases and shrinks within *FORK* and *JOIN* at every `par` statement.

$$\begin{array}{l} \mathit{JOIN}(\mathit{Blist}, \mathit{Self}, \mathit{Status}) \equiv \\ \textbf{if } \forall b \in \mathit{Blist} : \mathit{status}(b) = \mathit{completed} \wedge \\ \quad\quad \mathit{phase} := \mathit{ExecuteBehaviors} \\ \ \textbf{then } \forall b \in \mathit{Blist} : \\ \quad\quad \textbf{reduce}\ \ \mathit{BEHAVIOR} \textbf{ by } b \textbf{ endreduce}\ , \\ \quad\quad \mathit{children}(\mathit{Self}) := \emptyset, \\ \quad\quad \mathit{status}(\mathit{Self}) := \mathit{Status} \\ \textbf{endif} \end{array}$$

Try Statement. Finally, we define the semantics of the exception handling given by the combined try-trap-interrupt statement which basically extends the implementation of a behavior *b* encapsulated by `try` with additional exceptions $Excp_1, ..., Excp_M$ where

$$Excp_i \equiv [\underline{\texttt{trap}} \mid \underline{\texttt{interrupt}}](\texttt{Eventlist}_\texttt{i})\{\texttt{Handler}_\texttt{i};\}.$$

That means, that after keyword `trap` or `interrupt` a list of events is specified on which a behavior denoted as a *Handler* starts executing. The order of enumeration of the exceptions defines their priorities starting with the highest when multiple events are detected by the SpecC execution kernel. For our semantics, we thus define for an exception $Excp_i$ the functions $type(Excp_i) \in \{trap, interrupt\}$, $eventlist(Excp_i)$, and $behavior(Excp_i)$, where the latter two associate the list of events and the *Handler* to an exception. The semantics of the try statements defines as follows by simply 'linking' exceptions and their events to functions.

$$\begin{array}{l} \textbf{if } \mathit{Self\ executes}\ \langle \underline{\texttt{try}}\{b;\}\mathit{Excp}_1 ... \mathit{Excp}_M\rangle \\ \textbf{then}\ \ \ \mathit{status}(\mathit{Self}) := \mathit{waiting}, \\ \quad\quad \mathit{excpSensitivity}(\mathit{Self}) := \\ \quad\quad\quad \mathit{eventlist}(\mathit{Excp}_1) \cup ... \cup \mathit{eventlist}(\mathit{Excp}_M), \\ \quad\quad \mathit{exceptions}(\mathit{Self}) := \mathit{Excp}_1 + ... + \mathit{Excp}_M, \\ \quad\quad \mathit{programCounter}(\mathit{Self}) := \mathit{firstStmt}(b) \\ \textbf{endif} \end{array}$$

We set the behavior *Self* to *waiting* and accumulate all events that *Self* is *sensitive* to in *excpSensitivity*. In addition, all exceptions are stored in *exceptions*(*Self*) for later use by the kernel. Finally, the *programCounter* is advanced to the first statement of the behavior enclosed by `try`.

4.2.3. SpecC Kernel

The SpecC kernel is a separate process which is executed as soon as all user defined behaviors are not running, i.e., they are either *waiting*, *interrupted* or *completed*. We abbreviate this by:

$$\begin{array}{l} \mathit{BehaviorsRunning} \equiv \\ \exists b \in \mathit{BEHAVIOR} : \mathit{status}(b) = \mathit{running} \vee \\ \forall c \in \mathit{children}(b) : \mathit{status}(c) = \mathit{completed} \end{array}$$

When no behavior is running, the kernel goes through different sequential states (see Figure 3) determined by the function *phase*. These phases are expressed by the following rules where we have used placeholders for the individual sequential phases *ProcessEvents*, *ResetEvents*, *AdvanceTime*, and *ProcessTimeouts*.

$$\begin{array}{l} \textbf{if } \neg \mathit{BehaviorsRunning} \\ \textbf{then}\ \ \mathit{phase} := \mathit{ProcessEvents} \textbf{ endif} \end{array}$$

In details, the first $phase = ProcessEvents$ checks for events and sets behaviors to running which are sensitive

[4]Note that in order to handle also forking of exceptions, we model the state as a parameter. As we will see later, exception handling requires to set behaviors to *interrupted* or *completed*.

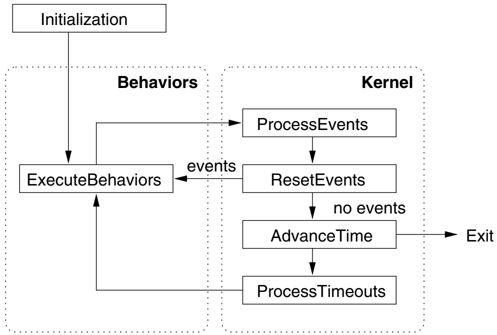

Figure 3: Phases of the SpecC Kernel

to events on `notify, notifyone`, and *exceptions*. The inner body matches behaviors b with a defined sensitivity (given by a wait statement) and the corresponding events which were notified. Therefore, the definition ranges over all $BEHAVIOR$s and $EVENT$s. The first case defines the condition when the event is supposed to trigger an exception. Then the exception handling executes which is defined in more details hereafter. Also, we have to handle *waiting* behaviors. When triggered by a `notify`, the behavior is simply reset to *running*. If any `notifyone` has been set, i.e., $notifiedonelist \neq \emptyset$, one behavior is arbitrarily chosen for each of the notifiedone sublists $nl \in notifiedonelist$ and set to *running*. Finally, the lists are reset and the next phase is set to $ResetEvents$.

```
if phase = ProcessEvents
 then
   var b ranges over BEHAVIOR
   var e ranges over EVENT
   if notified(e) = true ∧ e ∈ excpSensitivity(b)
    then HandleException
   endif
   if notified(e) = true ∧ e ∈ sensitivity(b)
    then  status(b) := running
          sensitivity(b) := ∅
   endif
   if notifiedonelist ≠ ∅
    then  ∀nl ∈ notifiedonelist :
            choose e1 in nl
            satisfying (∃b1 : e1 ∈ sensitivity(b1))
                status(b1) := running
                sensitivity(b1) := ∅
            endchoose
   endif
   endvar  endvar
   phase := ResetEvents
endif
```

Exception handling is defined in more detail by the following rule. This rule is applied when an exception is sensitive to an event and that event occurred. Then, the first matching exception denoted by *minException* is selected[5] We then have to distinguish if that exception is either of type *trap* or *interrupt*. In the first case, if behavior b is the topmost ancestor with a received exception event, all *descendants* (i.e., forked children and their children) of b are set to *completed* and the behavior of the exception is forked, and b is set to *completed*. In the second case, all *descendants* are correspondingly set to *interrupted*, and b is set to *interrupted* when the behavior of the exception is forked. As described in previous sections, the forked behavior resets the parent (i.e., b) and all its *descendants* to *running* after executing that last statement.

```
HandleExceptions ≡
if type(minException) = trap
 then if status(b) = waiting ∧ topmost(b) = true
     then ∀i ∈ descendant(b) : status(i) := completed
       FORK(behavior(minException), Self, completed)
    endif
elseif type(minException) = interrupt
      then if status(b) = waiting ∧ topmost(b) = true
           then
             ∀i ∈ descendant(b) : status(i) := interrupted
             FORK(behavior(minException), Self, interrupted)
      endif
endif
```

In *phase ResetEvents*, we simply reset all events and proceed to the execution of behaviors if any have been resumed by setting their *status* to *running*. Otherwise, we advance the time in order to resume behaviors which are waiting on the expiration of a *timeout*.

```
if phase = ResetEvents
then var e ranges over EVENT
          notified(e) := false,
      endvar ,
      notifiedonelist := ∅,
      if ∃b ∈ BEHAVIOR : status(b) = running
      then  phase := ExecuteBehaviors
      else  phase := AdvanceTime
      endif
endif
```

For advancing the time, we first have to check if all behaviors are completed since we need to exit the execution then. Otherwise, the current time T_c is advanced to the next point in time which is computed from the minimum over all timeouts.

In *phase AdvanceTime*, we exit the execution when all behaviors are *completed* and when no further timeouts are set. We also exit when all behaviors are *waiting* or are *interrupted* and when no further timeouts are set. This case is called a deadlock as there are behaviors waiting on events, but no events can be generated. Otherwise, we set the current time T_c to the next expiring timeout and proceed to $ProcessTimeouts$.

```
if phase = AdvanceTime
 then if ∀b ∈ BEHAVIOR :
          timeout(b) = undef
      then EXIT
      else
        Tc := min{timeout(b) | b ∈ BEHAVIOR∧
                                    timeout(b) ≠ undef},
        phase := ProcessTimeouts
      endif
endif
```

In the final phase, we simply set the status of all behaviors to *running* when their timeout equals the current execution time. Then, their timeout is reset and the kernel sets *phase* to $ExecuteBehaviors$ to resume the computation of the behavior's statements.

```
if phase = ProcessTimeouts
then  var b ranges over BEHAVIOR
        if timeout(b) = Tc
        then status(b) := running,
               timeout(b) := undef,
        endif
        endvar ,
        phase := ExecuteBehaviors
endif
```

[5]This is the first matching exception w.r.t. the order as they are defined in the trap statement.

5. FROM SPECIFICATION TO IMPLEMENTATION

When starting from an ASM specification, an implementation seems to be a straightforward refinement as it is summarized in Table 1. However, it has to be noted here that coding is still not trivial and still requires a lot of implementation decisions. As high level specification for coding languages which are based on virtual machines (e.g., VHDL and SpecC), ASMs can be an ideal starting point in order to check and verify language concepts before implementation. The 'closeness' of the ASM specification and an actual implementation basically guarantees stability since it leaves only little room for errors and definitely eliminates any ambiguities.

In the case to use our specification for implementation of an SpecC simulator, the translation is obvious for most of the basic patterns. Agents of behaviors map directly to threads, domains map to classes, and the kernel agent may directly map to the scheduler in the implementation. Note, however, that the kernel does not necessarily need to be a separate thread, but its basic control can be combined with the control of the individual behaviors. In the reference implementation, for example, the last thread that becomes *waiting* also executes the scheduler and selects the next thread to run after delivering all notified events and increasing the simulation time, etc. We can see it as an implementation decision here that the management of the program counter is combined with parts of the control of the SpecC kernel process. Similar decisions include, for example, the selection of the order of thread execution or the selection of waiting behaviors for 'notifyone' events.

For implementation of ASM rules, each rule set in the ASM specification generally directly maps into a function of our SpecC simulator. Nevertheless, identifying state functions and their associated state machine still requires some work which can be facilitated by a good documentation and adequate structuring of the ASM specification. A really critical issue in the translation to C++ is the selection of efficient data structures and most efficient matching/selection algorithms for implementation of quantifiers, var-constructs, etc. In particular, algorithms for the latter have to be carefully investigated in order to avoid any inefficiencies.

ASM	**C++**
agent	thread
domain	class
function	variable, method
macro	method
∀	loop/matching algorithm
∃	selection/matching algorithm
simultaneous function updates	variable assignments & method calls
if-then-else construct	state machine implementation & algorithm
var & choose construct	matching/selection algorithm & data structures
extend & reduce construct	allocation & garbage collection

Table 1: From ASMs to C++

6. CONCLUSION AND OUTLOOK

This article introduces the execution semantics of complete SpecC V1.0 by the means of ASMs. The specification has been defined along the notions given in the advanced SpecC introduction [6], the language reference manual [4] and the reference implementation. It clearly identifies basic entities and functions of the SpecC virtual machine. It can be taken as abstract pseudocode from which an implementation can be easily derived as it was outlined in Section 5. We think that ASMs provide an adequate framework for such applications, i.e., for clearly identifying execution concepts of virtual machines such as simulators and unambiguous description of the interaction of the associated concurrently communicating objects. Though our ASM specification is not directly executable, we think that it really supports and accelerates the development of simulators by providing the formal framework to reason about the validity of execution semantics of such systems.

Moreover, when a reference implementation is given–such as it was in our case–the specification really makes already implemented concepts clearer and greatly helps to relate them to the behaviroal semantics of established standard Hardware Description Languages like VHDL and Verilog. This is a very important point for the investigation of SpecC synthesis, i.e., for identification of subsets and patterns for the source language and different target languages. Therefore, our future investigations will focus on interoperability issues and equivalences between VHDL'93, SystemC, and SpecC models.

7. REFERENCES

[1] E. Börger. Annotated Bibliography on Evolving Algebras. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1994.

[2] E. Börger, U. Glässer, and W. Müller. Formal Definition of an Abstract VHDL'93 Simulator by EA-Machines. In C. Delgado Kloos and P. T. Breuer, editors, *Formal Semantics for VHDL*, pages 107–139. Kluwer, 1995.

[3] C. Delgado Kloos and P. T. Breuer. *Formal Semantics For VHDL*. Kluwer, Boston/London/Dordrecht, 1995.

[4] R. Doemer, A. Gerstlauer, and D. Gajski. *SpecC Language Reference Manual, Version 1.0*. SpecC Technology Open Consortium, March 2001.

[5] D. Gajski, J. Zhu, R. Doemer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, March 2000.

[6] A. Gerstlauer, R. Doemer, J. Peng, and D. Gajski. *System: Design: A Practical Guide with SpecC*. Kluwer Academic Publishers, June 2001.

[7] U. Glaesser, R. Gotzhein, and A. Prinz. Towards a new formal SDL semantics based on Abstract State Machines. In R. Dssouli, G. Bochmann, and Y. Lahav, editors, *Proc. of the 9th SDL Forum* . Elsevier Science B.V., 1999.

[8] Y. Gurevich. Evolving algebra 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, Oxford, 1994.

[9] W. Mueller, J.Ruf, D. Hofmann, J. Gerlach, T. Kropf, and W.Rosenstiehl. The Simulation Semantics of SystemC. In *Proc. of DATE 2001*. IEEE CS Press, March 2001.

[10] H. Sasaki. A Formal Semantics for Verilog-VHDL Simulation Interoperability by Abstract State Machine. In *Design, Automation and Test in Europe*, 1999.

[11] H. Sasaki, K. Mizushima, and T. Sasaki. Semantic Validation of VHDL-AMS by an Abstract State Machine. In *IEEE/VIUF International Workshop on Behavioral Modeling and Simulation*, 1997.

[12] C. Wallace. The Semantics of the C++ Programming Language. In E. Börger, editor, *Specification and Validation Methods*.