The Formal Execution Semantics of SpecC

Wolfgang Mueller,
Rainer Dömer,
Andreas Gerstlauer

Technical Report ICS-01-59
November 30, 2001

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

wolfgang@acm.org
doemer@cecs.uci.edu
gerstl@cecs.uci.edu

The Formal Execution Semantics of SpecC

Wolfgang Mueller,
Rainer Dömer,
Andreas Gerstlauer

Technical Report ICS-01-59
November 30, 2001

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

wolfgang@acm.org
doemer@cecs.uci.edu
gerstl@cecs.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`, `pipe`, and `try` statements as they are introduced in SpecC. We present our definition in form of distributed Abstract State Machine (ASM) rules reflecting the specification given in the SpecC Language Reference Manual [5]. We mainly see our formal semantics in three application areas. First, it can be taken as a high–level, pseudo code–oriented specification for the implementation of a SpecC simulator which is outlined in a separate section. Second, it is a concise, unambiguous description for documentation and standardization. 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.

Contents

List of Figures

The Formal Execution Semantics of SpecC

Wolfgang Mueller[*], **Rainer Dömer**[**], **Andreas Gerstlauer**[**]
[*]Paderborn University, Germany
[**]University of California, Irvine, USA

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`, `pipe`, *and* `try` *statements as they are introduced in SpecC. We present our definition in form of distributed Abstract State Machine (ASM) rules reflecting the specification given in the SpecC Language Reference Manual [5]. We mainly see our formal semantics in three application areas. First, it can be taken as a high–level, pseudo code–oriented specification for the implementation of a SpecC simulator which is outlined in a separate section. Second, it is a concise, unambiguous description for documentation and standardization. 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.*

1. Introduction

The SpecC language [6, 7] 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. Built on top of the C language, the de-facto standard for software development, SpecC supports additional concepts needed in hardware design and allows IP-centric modeling. SpecC allows to map modeling concepts onto language constructs in a one to one fashion. Unlike other system-level languages, the SpecC language precisely covers the unique requirements for embedded systems design in an orthogonal manner.

Although the SpecC language is defined by a Language Reference Manual (LRM) [5], and a reference implementation, a compiler and simulator, are freely available as open source, 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 the user understanding the language.

This article is the first publication of a formal SpecC semantics. Our semantics description is intended to provide 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 the domain of system synthesis and simulation, our formal semantics can be used as a sound basis to identify common behavioral concepts for interoperability with Verilog, VHDL, and SystemC. This is a first step for identifying common language patterns and subsets for SystemC synthesis.

We present a concise and rigorous but yet intuitive semantic definition of SpecC as defined in [5] in terms of Gurevich's *distributed Abstract State Machines* (ASMs) [9]. ASMs allow us to produce our specification following the terminology and the definitions given in the SpecC LRM [5] and corresponding to the VHDL'93 semantics in [2]. We develop a mathematical definition of SpecC in terms of a *SpecC Algebra* considering `wait`, `waitfor`, `notify`, `notifyone`, `par`, `pipe`, and `try` statements, 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. In Section 3, we briefly review the formalism of distributed ASMs. Then, Section 4 introduces the SpecC language and defines its execution semantics in terms of our *SpecC Algebra*. Section 5 outlines how to transform the given SpecC ASM specification

into a C++ implementation. Section 6 closes with a conclusion and outlook.

2. Related Work

Over previous years, research in formal semantics in EDA mainly focused on VHDL. There were quite a couple of approaches based on temporal logic, functional semantics, denotational semantics, and operational semantics applying Boyer-Moore Logic, Process Algebras, Petri-Nets etc. [4]. Most of the approaches cover subsets dedicated for application in formal verification. Olcoz et al., Reetz et al., and Boerger et al. have covered the complete VHDL language. Their definitions were based on Colored Petri-Nets, Flow Graphs, and Abstract State Machines [4]. The latter covered VHDL'93 and was extended for VHDL-AMS in [12]. Other applications investigated VHDL-Verilog interoperability [11]. Most recently, SystemC simulation semantics have been published in [10] which is oriented towards the VHDL'93 definitions in [2].

ASMs have 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 [3] and C++ [13]. Furthermore, the ITU standard SDL 2000 will be partly underlined by an ASM definition [8].

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, 10] 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 [9] for a formal introduction.

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

if *Condition* **then** $<Updates>$ **else** $<Updates>$ **endif**

Rules are basically nested if–then–else clauses with a set of function updates in their body. When executing the rules, the underlying ASM abstract machine performs 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 (*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* in imperative programming languages. A block is a set of function updates separated by a comma[1]. The individual function updates of each block are collected in a so–called update set. The individual updates of the update set are simultaneously executed in one step. Each function update changes a value at a specific location given by the left–hand–side of the assignment. 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 an inconsistency no state transition is performed and no update in the update set is being executed.

We demonstrate a simple guarded update by the following example:

if *true* **then** $A := B, B := A$ **endif**

That definition gives an simultaneous update of the 0-ary functions *A* and *B*. Since both updates are simultaneously executed, the values are swapped (*A* becomes the value of *B*, and vice versa). Due to its true condition, the rule fires at each step.

ASMs are multi–sorted based on the notion of universes. We assume 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

extend *Universe* **with** *v* $<Rule>$ **endextend**

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

reduce *Universe* **by** *v* $<Rule>$ **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

choose *v* **in** *Universe* $<Rule>$ **endchoose**

[1] In extension to [9], we use a comma in order to have an explicit separator between single updates. We also introduce sequential statements and enclosing a block in braces and separate them by semicolon, e.g., {C:=1; D := C}. This is a shortcut avoiding the introduction of an additional state function with additional conditions.

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:

var v **ranges over** $Universe$ $<Rule>$ **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 $Universe$. This can be outlined by the following example. It defines a rule which specifies that each non–empty l from the domain $LIST$ is replaced by the list's tail, i.e., deleting the first element of a list. l refers to any valid instance of $LIST$.

var l **ranges over** $LIST$
if $l \neq \langle\rangle$ **then** $l := tail(l)$**endif endvar**

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 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 [6, 7, 5] 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.

We first give a brief introduction to the structural SpecC aspects. Thereafter, we introduce the behavioral aspects by the means of a distributed ASM specification.

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. 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. 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.

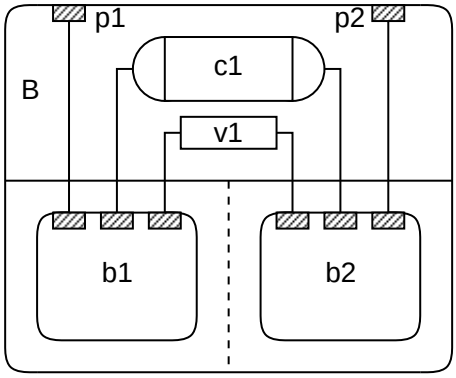

Figure 1. SpecC Example

The example shows a behavior `B` which has two ports, `p1` and `p2`, through which it can communicate with its environment. Internally, these ports are connected to two child behaviors, `b1` and `b2`, which execute concurrently. These child behaviors can communicate in two ways. First, both are connected to a shared variable `v1` which, for example, could be written by `b1` and then read by `b2`.

Second, `b1` and `b2` can communicate by use of a communication protocol provided by the channel `c1`. For example, the behavior `b1` could call a function `send` provided by the left interface of channel `c1`. Then, when behavior `b2` calls the `receive` function provided by the right interface, the communication protocol implemented in the channel will ensure that the data is transferred correctly, for example, by use of explicit hand shaking or some specific synchronization mechanism and timing.

Please note that Figure 1 only shows one level of the structural hierarchy of the system. The child behaviors `b1` and `b2` could again consist of a network of behaviors and channels. On the other hand, the behavior `B` can be part of a bigger system as well.

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, namely `wait`, `waitfor`, `notify`, `notifyone`, `par`, `pipe`, and `try` statements. Afterwards, we present a formal definition of

the kernel process. We presume a basic knowledge of the ANSI C and SpecC syntax here and refer the reader to [6, 7, 5] for more details.

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).

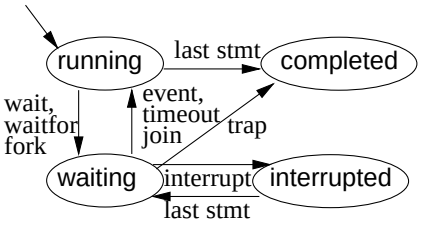

Figure 2. Life Cycle of a Behavior

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. Agents are instantiations of ASM modules. We first define rules for the *KERNEL_Module*. Thereafter, we define the semantics of distinguished statements executed in instantiations of the *BEHAVIOR_Module*. For initialization, we set

$Mod(b) := BEHAVIOR\_Module$
$\forall b \in BEHAVIOR$ and
$Mod(k) := KERNEL\_Module$
for the kernel process $k \in KERNEL$ [2].

Also, we 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.

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, $b_i \in descendant(b_i)$, 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

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

keeping all concurrent assignments. Finally, *resolve* selects non-deterministically one of these values.

if $Self\ executes\ \langle v\ =\ Expr\rangle$
then $value(v) := resolve(competingValues(value(Expr)))$,
$programCounter(Self) := nextStmt(Self)$
endif

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.

if $Self\ executes\ \langle \underline{\texttt{wait}}(EventList)\rangle$
then $status(Self) := waiting$,
$sensitivity(Self) := EventList$,
$programCounter(Self) := nextStmt(Self)$
endif

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.

if $Self\ executes\ \langle \underline{\texttt{waitfor}}(Delay)\rangle$
then $status(Self) := waiting$,
$timeout(Self) := T_c + Delay$,
$programCounter(Self) := nextStmt(Self)$
endif

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.

if $Self\ executes\ \langle \underline{\texttt{notify}}(EventList)\rangle$
then $\forall e \in EventList : notified(e) := true$,
$programCounter(Self) := nextStmt(Self)$
endif

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.

if $Self\ executes\ \langle \underline{\texttt{notifyone}}(EventList)\rangle$
then $notifiedonelist := notifiedonelist + Eventlist$,
$programCounter(Self) := nextStmt(Self)$
endif

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

if $Self\ executes\ \langle \underline{\texttt{par}}\{b_1;...;b_M\}\rangle$
then $SPAWN(\{b_1;...;b_M\}, Self)$,
$programCounter(Self) := nextStmt(Self)\}$
endif

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

$$SPAWN(Blist, Self) \equiv \{FORK(Blist, Self, waiting); JOIN(Blist, Self, running)\}$$

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[3]. When all children are completed, it is reset to *running*.

$FORK(Blist, Self, Status) \equiv$
$\forall b \in Blist$:
extend *BEHAVIOR* **with** *b*
$status(b) = running,\quad parent(b) := Self$
endextend ,
$children(Self) := Blist$,
$status(Self) := Status$

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.

$JOIN(Blist, Self, Status) \equiv$
if $\forall b \in Blist : status(b) = completed \wedge$
$phase := ExecuteBehaviors$
then $\forall b \in Blist$:
reduce *BEHAVIOR* **by** *b* **endreduce** ,
$children(Self) := \emptyset$,
$status(Self) := Status$
endif

[3]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*.

Pipe Statement. Similar to the `par` statement, the `pipe` statement[4] also spawns a set of children. In addition, the `pipe` statement consists of five phases, *init*, $filling$,*running*, $flushing$, $finished$, which reflect the actual behavior of a pipeline.

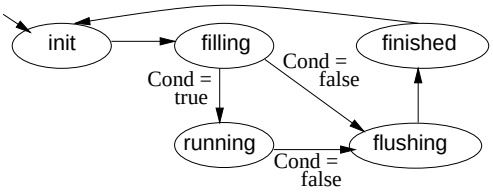

Figure 3. Different States of a Pipe

Comparable to a `for` loop in C, the SpecC `pipe` statement can be seen as an iterator with an initial statement *Init* before filling the pipe, an incremental statement *Incr* which executes after each iteration, and a condition *Cond* which determines when the pipeline starts flushing.

As an example, let us consider a pipeline with 4 behaviors. Let us further consider that after spawning the second behavior in the second loop, the condition becomes false. Then, in the next 3 loops the pipe flushes spawning $\{b_2, b_3\}$, $\{b_3, b_4\}$, and $\{b_4\}$ before terminating in status $finished$ (see Figure 4).

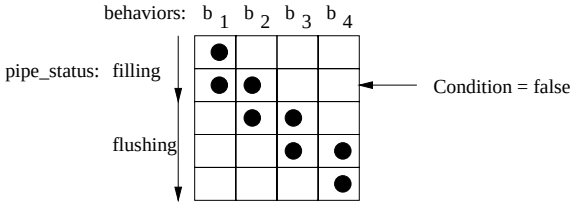

Figure 4. Pipeline Example

The following rule implements the main state transitions as given in Figure 3 by setting *pipe_status*.

if $Self executes\langle$`pipe` $(Init; Cond; Incr)\ \{b_1; ...; b_M\}\rangle$
then
  if $pipe\_status(Self) := init$
  then
    $pipe\_status(Self) := filling,$
    $EXECUTE(\langle Init\rangle)$
  endif
  if $pipe\_status(Self) := filling$
  then $fillPipe$ **endif**
  if $pipe\_status(Self) := running$
  then *runPipe* **endif**
  if $pipe\_status(Self) := flushing$
  then $flushPipe$ **endif**
  if $pipe\_status(Self) := finished$
  then
    $pipe\_status(self) := init,$
    $programCounter(Self) := nextStmt(Self)$
  endif
endif

Here, initially the *Init* statement is executed and the pipe starts $filling$. During $filling$, a loop starts to spawn an incremental number of behaviors. As soon as the pipe condition evaluates to false, the pipe proceeds to $flushing$, otherwise to *running*[5].

$fillPipe \equiv$
$\{FILL\_LOOP(\{b_1, ..., b_M\}, Self);$
if $value(\langle Cond\rangle) = false$
then $pipe\_status(self) := flushing$
else $pipe\_status(Self) := running$
endif $\}$

After having completely filled, the pipe continuously spawns all behaviors, after which the *Incr*-statement is executed, and the *Cond* is checked. If it evaluates to false, the pipe proceeds to $flushing$.

$runPipe \equiv$
$\{SPAWN(\{b_1, ..., b_M\}, Self);$
$EXECUTE(\langle Incr\rangle);$
if $value(\langle Cond\rangle) = false$
then $pipe\_status(self) := flushing$
endif $\}$

When *pipe_status* is $flushing$, we simply execute the $FLUSH\_LOOP$ macro defined below.

$flushPipe \equiv$
$\{FLUSH\_LOOP(\{b_1, ..., b_M\}, Self);$
$pipe\_status(Self) := finished\}$

[4] Without loss of generality, we only discuss the `pipe` statement with termination arguments here. The `pipe` statement without arguments is just a special case that never terminates.

[5] $FILL\_LOOP$ defines a for loop over the given behaviors which is given as a macro after the definitions of the states.

Loops for filling and flushing are given by the following two macros. We use a C-like description since we found that this execution can be better given in programming language–like constructs. Their transformations to ASMs should be rather intuitive and left to the reader. The first definition gives a `for`-loop incrementing the number of spawned behaviors from b_1 to b_M for filling the pipe. In each interaction, after all behaviors have joined, the *Incr*–statement is executed. The loop immediately exists when the condition *Cond* of the pipe evaluates to false.

$$FILL\_LOOP(\{b_1,...,b_M\},Self) =$$
for $(last = 1, first = 1; (last < M)\&\&(\langle Cond\rangle);$
$last++)$
$\{SPAWN(\{b_{first},...,b_{last}\},Self); EXECUTE(\langle Incr\rangle)\}$

The loop for flushing takes the currently spawned behaviors $\{b_{first},...,b_{last}\}$[6] and repeats until $first$ finally reaches $last = M$.

$$FLUSH\_LOOP(\{b_1,...,b_M\},Self) =$$
while $(first < last)\{$
$SPAWN(\{b_{first},...,b_{last}\},Self);$
if $(last < M) last++;$
$first++\ \}$

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.

if $Self$ *executes* $\langle \underline{\texttt{try}}\{b;\}Excp_1...Excp_M\rangle$
then $status(Self) := waiting,$
$excpSensitivity(Self) :=$
$eventlist(Excp_1) \cup ... \cup eventlist(Excp_M),$
$exceptions(Self) := Excp_1 + ... + Excp_M,$
$programCounter(Self) := firstStmt(b)$
endif

[6]Note here, that $first$ and $last$ keep their values from filling the pipe. After completely filling, $first = 1$ and $last = M$. When not completely filled, $last < M$.

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:

$$BehaviorsActive \equiv$$
$$\exists b \in BEHAVIOR : status(b) = running \vee$$
$$\forall c \in children(b) : status(c) = completed$$

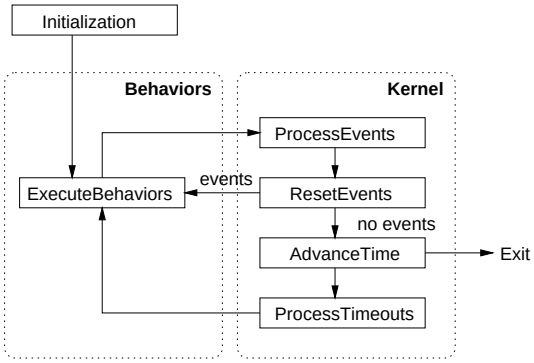

Figure 5. Phases of the SpecC Kernel

When no behavior is running, the kernel goes through different sequential states (see Figure 5) 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*,

if $\neg BehaviorsActive$
then $phase := ProcessEvents$ **endif**

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

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 $notified(e) = true \wedge e \in excpSensitivity(b)$
  then $HandleException$
  endif
  if $notified(e) = true \wedge e \in sensitivity(b)$
  then $status(b) := running$
    $sensitivity(b) := \emptyset$
  endif
  if $notifiedonelist \neq \emptyset$
  then $\forall nl \in notifiedonelist$:
    choose e_1 **in** nl
    satisfying $(\exists b_1 : e_1 \in sensitivity(b_1))$
      $status(b_1) := running$
      $sensitivity(b_1) := \emptyset$
    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[7] 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.

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

$HandleExceptions \equiv$
if $type(minException) = trap$
then if $status(b) = waiting \wedge topmost(b) = true$
  then $\forall i \in descendant(b) : status(i) := completed$
    $FORK(behavior(minException), Self, completed)$
  endif
elseif $type(minException) = interrupt$
  then if $status(b) = waiting \wedge topmost(b) = true$
    then
      $\forall i \in descendant(b) : status(i) := interrupted$
      $FORK(behavior(minException), Self, interrupted)$
  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 ,
  $notifiedonelist := \emptyset$,
  if $\exists b \in BEHAVIOR : status(b) = running$
  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 $\forall b \in BEHAVIOR$:
    $timeout(b) = undef$
  then $EXIT$
  else
    $T_c := min\{timeout(b) \mid b \in BEHAVIOR \wedge$
      $timeout(b) \neq undef\}$,
    $phase := ProcessTimeouts$
  endif
endif

In the final phase, we simply set the staus 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 $timeout(b) = T_c$
  then $status(b) := running$,
    $timeout(b) := undef$,
  endif
  endvar ,
  $phase := ExecuteBehaviors$
endif

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 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 and 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
$\forall$	loop/matching algorithm
$\exists$	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 [7], the language reference manual [5] 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 already exists – 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.

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] E. Börger and W. Schulte. Defining the Java Virtual Machine as Platform for Provably Correct Java Compilation. In L. Brim, J. Gruska, and J. Zlatuska, editors, *Mathematical Foundations of Computer Science, MFCS 98*, Lecture Notes in Computer Science. Springer, 1998.

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

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

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

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

[8] 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, *Proceedings of the 9th SDL Forum* . Elsevier Science B.V., 1999.

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

[10] 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.

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

[12] 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.

[13] C. Wallace. The Semantics of the C++ Programming Language. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.