The Simulation Semantics of SystemC

Wolfgang Mueller
C-LAB/Paderborn University
Paderborn, Germany

Juergen Ruf, Dirk Hoffmann, Joachim Gerlach, Thomas Kropf, Wolfgang Rosenstiehl
University of Tuebingen
Tuebingen, Germany

Abstract

We present a rigorous but transparent semantics definition of SystemC that covers method, thread, and clocked thread behavior as well as their interaction with the simulation kernel process. The semantics includes watching statements, signal assignment, and wait statements as they are introduced in SystemC V1.0. We present our definition in form of distributed Abstract State Machines (ASMs) rules reflecting the view given in the SystemC User's Manual and the reference implementation. We mainly see our formal semantics as a concise, unambiguous, high-level specification for SystemC-based implementations and for standardization. Additionally, it can be used as a sound basis to investigate SystemC interoperability with Verilog and VHDL.

1. Introduction

SystemC is the emerging de-facto-standard for system-level modeling and design from the Open SystemC Initiative (OSCI) which is controlled by a steering group and backed by a growing community of currently over 50 charter member companies from the systems, semiconductor, IP, embedded software, and EDA industries. SystemC has received an extreme increase in acceptance over the last months for system specification and simulation. Although SystemC comes with a well-written User's Manual and a reference implementation of the simulator the documentation leaves some open question w.r.t. the precise meaning. However, a precise semantics of SystemC is mandatory for various applications in simulation, synthesis, and formal verification.

To our knowledge, this article is the first publication of a formal SystemC semantics. Our semantics description is intended to provide a concise definition of the complete SystemC V1.0 simulation semantics for potential standardization. This is an important step towards future SystemC compliant implementations and applications in various fields like formal verification. In the domain of system synthesis and simulation our formal semantics can be used as a sound basis for SyntemsC language extensions and to define interoperability with Verilog and VHDL models by their ASMs representation such as described in [11].

We present a concise and rigorous but yet intuitive semantic definition of SystemC V1.0 [9] in terms of Gurevich's *distributed Abstract State Machines* [6]. ASMs allows us to produce our specification following the terminology and the view given in the SystemC User's Manual and corresponding to the VHDL'93 semantics in [2]. We develop a mathematical definition of SystemC in terms of a *SystemC Algebra* considering methods, threads, and Cthreads with signal assignments, wait statements, and watching statements as well as the full computational model of interaction between the user defined processes and the simulation kernel process.

The remainder of this paper is organized as follows. Section 2 discusses related works. In Section 3, we briefly introduce what is needed from distributed ASMs. Section 4 gives an introduction to SystemC and its behavioral semantics in terms of a *SystemC Algebra*. Section 5 closes with a conclusion and outlook.

2. Related Works

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. [4] have covered the complete

1530-1591/01 $10.00 © 2001 IEEE

VHDL language. Their definitions were based on Colored Petri-Nets [7], Flow Graphs [10], and Abstract State Machines [2]. The latter covered VHDL'93 and was extended for VHDL-AMS in [12]. Other applications investigated VHDL-Verilog interoperability [11].

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, it is expected that the ITU standard SDL 2000 will be partly underlined by an ASM definition [5].

All these investigations demonstrated 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 SystemC simulator. In this article, we focus our investigations on SystemC V1.0 since it is currently the latest version which is supported by a reference implementation. However, as soon as other versions will be established we see no problem to scale our model to future SystemC extensions.

Our Algebra gives definitions of variable assignments, signal assignments, watching, and wait statements. The model is defined along the lines of the basic concepts of the VHDL'93 definition in [2] so that future work on interoperability with VHDL and Verilog are possible.

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 [6] for a formal introduction.

An ASM specification comes in form of guarded function updates (rules). 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. Rules are of form

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

At each step the guards evaluate to a set of function updates (block) each of 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:

$$\textbf{if } true \textbf{ then } A := B, B := A \textbf{ endif}$$

That definition gives an simultaneous update of the 0-ary functions A and B. Since both updates are simultaneously executed, 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 mentioning. The **var** rule constructor defines the simultaneous instantiation of a rule over a universe:

$$\textbf{var } v \textbf{ ranges over } Universe <Rules>$$

The constructor spawns and executes the rules for each element in $Universe$ simultaneously, i.e., the constructor spawns n rules where n is the (finite) 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 SystemC specification in the next section comes in the form of two modules: One for the kernel process and one for the processes.

4. SystemC

The SystemC language is basically a C++ language extension. SystemC comes as a freely available C++ library with header files describing the classes and a link library that contains the simulation kernel. A SystemC description can be compiled by any ANSI-compliant C++ compiler. The resulting executable specification realizes a cycle-based simulator that allows high speed simulation with integrated simulation control facilities. In the following, we focus on SystemC V1.0 [9] since that version is the latest one which is completely supported by the reference implementation. We first give a brief introduction to the structural SystemC aspects. Thereafter, we introduce the behavioral aspects by the means of an ASM specification.

4.1. Structure

In SystemC, the fundamental building block is a module (SC_MODULE). The main routine (sc_main) instantiates a

[1] In extension to [6] we use a comma in order to have an explicit separator between single updates.

set of hierarchically embedded submodules which are connected via in, out, and inout ports[2]. $PORTs$ can connect to $SIGNALs$ and other $PORTs$. $SIGNALs$ are data containers whose value changes generate events for the event-based simulator. $SIGNALs$ can create connections between module $PORTs$ establishing a direct communication link between modules. $PORTs$ can be seen as pointers to $SIGNALs$ so that each $PORT$ has to refer to exactly one $SIGNAL$. SystemC supports resolved and unresolved signals. Resolved signals can have more than one driver (busses) while unresolved signals can only have a single driver. Modules can contain definitions of SystemC functional units (processes) typically given in the constructor (SC_CTOR) of the module.

4.2. Formal Behavioral Semantics

The next paragraphs give the stepwise development of a formal SystemC V1.0 semantics starting with the basic behavioral concepts. Thereafter, we present a formal definition of the simulation kernel and the various SystemC specific statements, i.e., watching statements, signal assignment, and wait statements. Due to page limitation, we unfortunately cannot give detailed SystemC examples here and have to presume a basic knowledge of the SystemC syntax.

4.2.1. Basic Concepts

Derived from hierarchically organized modules, SystemC establishes a hierarchical network of a finite number of parallel communicating processes $p \in METHOD \cup THREAD \cup CTHREAD$[3] which, under the supervision of the distinguished simulation kernel process, concurrently update new values for given $SIGNALs$ and $VARIABLEs$. Signals do not change their values immediately. Their assignments become effective only in the next simulation cycle.

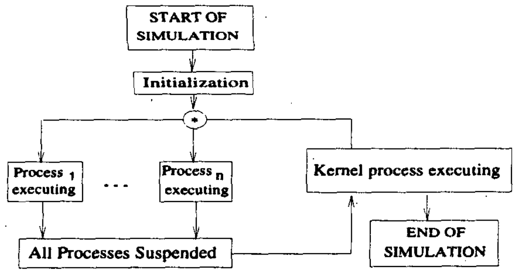

Figure 1. SystemC Simulation Cycles

The simulation is initiated by the *sc_start* command which first assigns initial values to signals. After the initial generation of events, there is a mutually exclusive execution of the simulation kernel process and the concurrently running (user defined) processes, i.e., the kernel process periodically starts its execution if all user defined processes are suspended and vice versa (cf. Figure 1). Each user defined process is *active* until it *suspends* upon reaching a wait statement or after executing the last process statement. Before getting *active* again, a process first checks its watching conditions and sets its program counter accordingly. Due to the life cycle of a process p we set $status(p) \in$ {*active, suspended, checkGlobalWatching, checkLocalWatching*}. After invocation, a method moves from *status suspended* to *active*. A thread has to check its global watching conditions before proceeding to *active*; Cthreads check their global and local watching conditions (see Fig. 2).

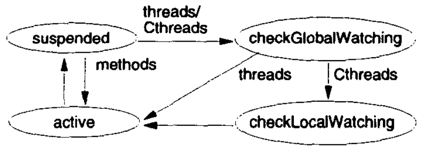

Figure 2. Life Cycle of a Process

Given the underlying discrete SystemC time model, the domain $TIME$ is linearly ordered and contains the distinguished element T_c for *current simulation time*. There are distinguished CLOCK objects. A $c \in CLOCK$ is represented as a list. Each list has the form $< c_1, ..., c_n >$ with $c_i \in TIME \times EDGE$. Lists are linearily ordered w.r.t. their time component $\in I\!R$. Elements of $EDGE$ are of value *neg* (negative) or *pos* (positive).

When all user defined processes are *suspended*, the kernel process goes through different phases and updates signals and clocks, invokes processes, and advances simulation time. Advancements of clocks and assignments to signals which are performed by user defined processes cause events which may trigger processes again to execute. SystemC processes are classified into methods, threads and *clocked* threads, i.e., $METHOD$, $THREAD$, $CTHREAD$. Clocked threads are executed only on request of time advancement, i.e., after all $METHODs$ and $THREADs$ require no further execution at T_c. Before resuming processes and executing them, the new signal values have to be assigned to current values which may lead to the generation of new events.

The rules in the following paragraphs constitute the program of ASM agents, one for the simulation kernel process and one for each user defined process. 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 $PROCESS\_Module$. For initialization we set

$Mod(a) := PROCESS\_Module$

$\forall a \in METHOD \cup THREAD \cup CTHREAD$ and

$Mod(k) := KERNEL\_Module$

for the simulation kernel process $k \in KERNEL$[4]. Modules of threads and clocked threads are set $undef$ after executing the last statement which disables these processes until the end of simulation (EOS). We suppose *phase = executeProcesses* and current

[2]Note here, that in contrast to VHDL, SystemC has no elaboration phase.

[3]clocked threads

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

time T_c to be set to 0.0. Events for all clocks with first elements at time 0.0 are set to true. Unless otherwise stated all functions are assumed to be *undef* and sets and lists to be empty.

The remainder of this document first defines the behavior of the simulation kernel. Thereafter, we give the semantics of the SystemC V1.0 specific statements.

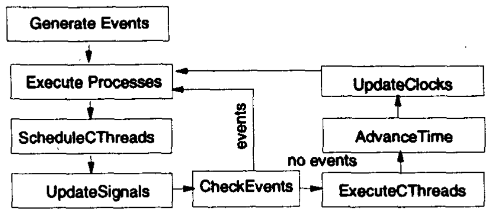

Figure 3. Phases of the Simulation Kernel

4.2.2. SystemC Simulation Kernel

The SystemC kernel is a separate process which is executed as soon as all user defined processes are suspended (cf. Figure 1). We abbreviate this by:

$$\begin{aligned}&AllProcessesSuspended \equiv\\ &\forall p \in METHOD \cup THREAD \cup CTHREAD :\\ &\qquad\qquad status(p) = suspended\end{aligned}$$

When all processes are suspended, the kernel goes through different states (see Fig. 3) by setting the function *phase* where *GenerateEvents* generates initial events for all *CLOCKs* which are active at T_c. Thereafter, *METHODs* and *THREADs* with events are executed until they suspend. *THREADs* are suspended after executing a wait statement. *METHODs* are suspended after their last statement. Thereafter, *CTHREADs* with events are scheduled for future execution. When no further events are generated, the scheduled *CTHREADs* are finally executed, the simulation time is advanced, and *CLOCKs* are updated w.r.t. the time of the next active clock. These phases are expressed by the following rules where we have used placeholders for the individual sequential phases *ExecuteProcesses*, *SCheduleCThreads* etc.[5]

```
if $AllProcessesSuspended$
then
  $ExecuteProcesses$
  $ScheduleCThreads$
  $UpdateSignals$
  $CheckEvents$
  $ExecuteCThreads$
  $AdvanceTime$
  $UpdateClocks$
endif
```

[5]Though we do not see any necessity to re-run *ScheduleCThreads* every cycle, we would like to keep it as given in the SystemC V1.0 User's Manual [9].

In details, *ExecuteProcesses* checks for events of all *SIGNALs* and *CLOCKs* in all *sensitivityLists*[6] of all *METHODs* and *THREADs*. In the case of an *event* each thread t is set to *status checkGlobalWatching* after which it will be activated. Methods are immediately set *active* since no watchings are permitted. *phase* is finally incremented.

```
$ExecuteProcesses \equiv$
var $m$ ranges over $METHOD$
var $t$ ranges over $THREAD$
var $s$ ranges over $SIGNAL \cup CLOCK$
if $phase = executeProcesses$
then
  if $event(s) \wedge s \in sensitivityList(t)$
  then $status(t) := checkGlobalWatching$ endif
  if $event(s) \wedge s \in sensitivityList(m)$
  then $status(m) := active$ endif
  $phase := scheduleCThreads$
endif
```

ScheduleCThreads checks for events on *CLOCKs* over all *sensitivityLists* of all *CTHREADs*. In the case of an *event* the corresponding *CTHREAD* p is added to the set of *scheduled CTHREADs*, clock events are reset, and the next phase is being assigned.

```
$ScheduleCThreads \equiv$
if $phase = scheduleCThreads$
then var $p$ ranges over $CTHREAD$
      var $c$ ranges over $CLOCK$
      if $event(c) \wedge c \in sensitivityList(p)$
      then $scheduled := scheduled \cup \{p\}$
      endif
      $event(c) := false,$
      $phase := updateSignals$
endif
```

After scheduling *CTHREADs* their outputs are updated if the new value of the output signal does not equal the old value. Each signal update generates an event. Other events of other signals are reset to *false*.

```
$UpdateSignals \equiv$
if $phase = updateSignals$
then var $s$ ranges over $SIGNAL$
      if $value(s) \neq newValue(s)$
      then $value(s) := newValue(s),$
            $event(s) := true$
      else  $event(s) := false$
      endif
      $phase := checkEvents$
endif
```

The next phase checks if any events have been generated on signals and sets the next phase either to *executeProcesses* or toexecute *CThreads*.

[6]For better readability we use $s \in sensitivityList(p)$ as an abbreviation for $head(s) \in sensitivityList(p)$ if $s \in CLOCK$.

```
$CheckEvents \equiv$
if $phase = checkEvents$
then if $\exists s \in SIGNAL : event(s) = true$
    then $phase := executeProcesses$,
    else $phase := executeCThreads$
    endif
endif
```

When no further events are generated at the current time T_c the set of postponed $CTHREADs$ are executed by setting their $status$ to $checkGlobalWatching$ which lateron proceeds to $active$. Additionally, the set of $scheduled$ processes has to be reset for the next cycle, and $phase$ proceeds to $advanceTime$.

```
$ExecuteCThreads \equiv$
var $p$ ranges over $scheduled$
if $phase = ExecuteCThreads$
then
  $status(p) := checkGlobalWatching$,
  $scheduled := \emptyset$,
  $phase := advanceTime$
endif
```

For advancing the time, we first have to check for the final end of simulation (EOS) at which the simulation kernel is deactivated to $undef$. Otherwise, the current time T_c is advanced to the next point in time T_n and clocks are updated accordingly.

```
$AdvanceTime \equiv$
if $phase = advanceTime$
then if $T_n = EOS$
    then $Mod(Self) := undef$
    else $T_c := T_n$,
        $phase := updateClocks$
    endif
endif
```

T_n is computed by considering the minimum of all first and second element of all clock waveforms $\geq T_c$. If the time of the first element equals the current time, the time of the second element has to be considered.

$$T_n = min\{t \mid t = time(c_{i,j})\}, where$$

$$j = \begin{cases} 1, & if\ time(c_{i,1})) > T_c \\ 2, & otherwise \end{cases}$$

$\forall c_{i,j} \in CLOCK$ denoting the j-th waveform element of clock i. Clocks which have an active edge at the new current time T_c are updated by removing the outdated first element, i.e., setting $c := tail(c)$ if the second element of the clock waveform c equals the current time T_c. Thereafter, an event is set for each of those signals and $phase$ proceeds to perform the execution of processes again.

```
$UpdateClocks \equiv$
if $phase = updateClocks$
then $phase := executeProcesses$,
    var $c$ ranges over $CLOCK$
    if $tail(c) \neq \emptyset \wedge time(c_2) = T_c$
    then $c := tail(c)$,
        $event(c) := true$
    endif
endif
```

4.2.3. Checking Watching Conditions

After invocation and before becoming $active$, each process first has to check for true global and local watching conditions (see Fig. 2)[7]. A global watching can be defined for clocked and unclocked threads. Local watchings are only allowable within a Cthread and can be nested. If a watching condition evaluates to true the continuation of the program either

(i) resets to the first statement of the thread/Cthread in the case of a global watching or

(ii) proceeds to the first statement of the escape subblock within the local watching block.

To model these continuations, we use the function $programCounter$ which gives an abstraction of the control flow when executing statements of a SystemC program. To model the global watching, we specify the set $globalWatch(p)$ for each process p with elements $\in CONDITION$ representing the global watching conditions $cond(Expr)$ of a process p. Conditions are derived from the expression $Expr$ of a watching statement. $globalWatch(p)$ represents one condition with disjunctively combined subconditions, all with same priority.

For modeling local watchings, we use a priority list $localWatch(p)$ with elements $\in CONDITION \times ESCAPE$ for each process p. Elements in that list are tuples with a condition and an abstract representation of an escape block. For $e \in ESCAPE$, the function $first(e)$ returns the first statement in that block and $succ(e)$ returns the successor statement of that block, i.e., the first statement after the W_END. Both functions are required to reset the program counter during the execution. The list $localWatch$ is ordered w.r.t. the priority of the watching statement. We assume that the list has one distinguished element with highest priority. When there are several watching conditions defined in the same watching block, their conditions are disjunctively joined within one list element. For nested local watching blocks the priority of the enclosing block is higher than priority of the enclosed block. Global watching conditions have higher priorities than local ones.

In status $checkGlobalWatching$ global watching conditions of processes $Self \in THREAD \cup CTHREAD$ are checked. Note here, that $globalWatch$ becomes true when one of its subconditions evaluate to true. In that case the $programCounter$ is adjusted accordingly, i.e., it is reset to the first statement of the thread. Thereafter, all local watchings are inactivated by setting

[7]For the syntax of the statements the reader is referred to the watching statements paragraph on the next page.

their list to $\emptyset$. This has to be done since the priority of global watching is higher than the one for the local watchings. Finally, $CTHREADs$ are set to $status$ $checkLocalWatching$; other threads directly proceed to $active$.

```
if $status(Self) := checkGlobalWatching$
then if $globalWatch(Self) = true$
     then
        $programCounter(Self) :=$
                    $jump(Self, firstStatement(Self))$,
        $localWatch(Self) := \emptyset$,
        $mode(Self) := global$
     endif
     if $Self \in CTHREAD$
     then $status(Self) = checkLocalWatching$
     else $status(Self) = active$
     endif
endif
```

In status $checkLocalWatching$ a $CTHREAD$ checks for a list element e of highest priority with true watching condition. If there exists one, that element is extracted from $localWatch$ by the function $findLocalCond$. We leave that function unspecified since its definition should be intuitively clear. If it does not exist, i.e., $e = \bot$, the process $Self$ proceeds to status active. Otherwise, the $programCounter$ is reset to the first statement of the escape block of e, i.e., $first(escape(e))$. Additionally, $trunc$ prunes the lower priority tail elements including e from $localWatch$.

```
if $status(Self) = checkLocalWatching$
then
  if $(e := findLocalCond(localWatch(Self))) \neq \bot$
  then
     $programCounter(Self) :=$
                    $jump(Self, first(escape(e)))$,
     $localWatch(Self) := trunc(e, localWatch(Self))$
  endif
  $status(Self) := active$
endif
```

4.2.4. SystemC Statements

We now can proceed with the semantics definition of the SystemC V1.0 specific statements given in the below table. The table shows their allowable usage within the different sorts of processes.

	Methods	Threads	CThreads
global watching	-	x	x
local watching	-	-	x
signal assignment	x	x	x
wait	-	x	x
wait until	-	-	x

We first discuss the role of the program pointer. Thereafter, we give the semantics of watching statements, signal assignment, and wait statements.

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

```
$Self$ $executes$ statement $\equiv$
$programCounter(Self) =$ statement $\wedge$ $status(Self) = active$
```

The semantics of the individual watching, signal assignment, and wait statements are given hereafter.

Watching Statements. We start with the semantics elaborating global and local watching statements. In order to decide if the current watching definition ($watching(Expr)$) appears in the context of a global watching or of the current local one we set $mode(p) \in \{local, global\}$. Since the program counter starts in the outer scope, we initially set all threads to mode $global$. Global watchings are typically defined in the constructor SC_CTOR of a SystemC module, e.g.:

```
SC_CTOR {
    SC_CTHREAD(cthread_fct, clk.pos());
    watching(reset.delayed() == true);
}
```

In contrast, local watchings are defined within the scope of a Cthread. Each local watching block is enclosed by W_BEGIN and W_END and has the form

```
W_BEGIN<...>W_DO<...>W_ESCAPE<...>W_END
```

where the list of watching conditions are specified after W_BEGIN. The control flow of a CThread continues after the W_DO if all watching conditions evaluate to $false$ and jumps to the first statement after W_ESCAPE as soon as one condition evaluates to $true$. Otherwise, that part is skipped and the program continues after W_END.

For global watching definitions, the condition given by an individual expression is added to $globalWatch$. Otherwise, in $mode$ $local$ the current condition is joined with the condition of the actual list element of the actual local watching block.

```
if $Self$ $executes$ (watching($Expr$))
then $programCounter(Self) := nextStmt(Self)$
  if $mode(Self) = global$
  then $globalWatch(Self) := globalWatch(Self)$
          $\cup\, cond(Expr)$
  else $localWatch(Self) :=$
          $addToActual(localWatch(Self), cond(Expr))$
  endif
endif
```

Upon reaching a W_BEGIN, watching conditions are decided to be local thereafter. Additionally, an new actual element for the local watching list is generated through the abstract function $addNewActual$. Its initial condition is set $false$. All local watching conditions inside the W_BEGIN W_DO block are disjunctively added to the actual $localWatch$ element by the function $addToActual$.

```
if $Self\ executes\ \langle \underline{\texttt{W\_BEGIN}} \rangle$
then $mode(Self) := local,$
     $localWatch(p) = addNewActual(localWatch(p),$
          $(false, ESC\_Block(programCounter(Self)))),$
     $programCounter(Self) := nextStmt(Self)$
endif
```

Upon reaching a W_DO, the conditions thereafter can be of type global again. Thus, we reset $mode$ to $global$.

```
if $Self\ executes\ \langle \underline{\texttt{W\_DO}} \rangle$
then $mode(Self) := global,$
     $programCounter(Self) := nextStmt(Self)$
endif
```

When executing W_ESCAPE (i.e., after having completed the W_DO block) the *programCounter* is set to the successor of the W_END statement, i.e., the successor of $escape$ block of the $actual$ watching block. Additionally, the currently executed local watching is removed from the priority list of local watchings by $removeActual$.

```
if $Self\ executes\ \langle \underline{\texttt{W\_ESCAPE}} \rangle$
then $programCounter(Self) :=$
          $succ(escape(actual(localWatch(Self)))),$
     $localWatch(Self) :=$
          $removeActual(localWatch(Self))$
endif
```

Signal Assignment. Right-hand-side values in signal assignments are not immediately assigned to the current value of signal S but to its potential $newValue$. Updating a current value with a new value generates an event if the values are different. The update is performed by the simulation kernel process. This operation is equivalent with the $write$ statement. Note here, that parallel write accesses to the $newValue$ are allowable. Corresponding to the semantics of global variable assignments the competing assignments to S are non-deterministically resolved.

```
if $Self\ executes\ \langle S\ =\ Expr \rangle$
then $newValue(S) :=$
          $resolve(competingNewValues(S)),$
     $programCounter(Self) := nextStmt(Self)$
endif
```

Wait Statements. On reaching a wait statement a process simply stops execution by setting its $status$ to $suspended$.

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

Due to the SystemC User's Manual[8], other variations can be derived from that basic definition. The variant $wait(n)$ in $CTHREADs$ may be expressed by a sequence of n $wait()$ statements and

$wait\_until(Expr) \equiv do\ \ wait();\ \ while(!Expr);$

5. Conclusion and Outlook

This article introduces the simulation semantics of complete SystemC V1.0 by the means of ASMs. We have limited our definitions to SystemC V1.0 since this is the latest version with a reference implementation at the time when this article was written. However, due to the scalability of ASM specifications our semantics definition should be easily scalable to future SystemC versions.

Our future investigations will focus on extensions of our ASM definition towards future versions of SystemC and on interoperabilities and equivalences between VHDL'93 and SystemC 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] 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.

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

[7] S. Olcoz. A Formal Model of VHDL Using Colored Petri-Nets. In C. Delgado Kloos and P. T. Breuer, editors, *Formal Semantics For VHDL*. Kluwer, Boston, 1995.

[8] Open SystemC Initiative, Synopsys Inc, CoWare Inc, Frontier Inc. *SYSTEM C Version 0.9 User's Guide*, 1999.

[9] Open SystemC Initiative, Synopsys Inc, CoWare Inc, Frontier Inc. *SYSTEM C Version 1.0 User's Guide*, 2000.

[10] R. Reetz and T. Kropf. Correct system level design with VHDL. In C. Delgado Kloos and P. T. Breuer, editors, *Formal Semantics For VHDL*. Kluwer, Boston, 1995.

[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.