An Executable Operational Semantics for SystemC using Abstract State Machines

Amjad Gawanmeh, Ali Habibi, and Sofiène Tahar

Department of Electrical and Computer Engineering,

Concordia University, Montreal, Canada

Email: {amjad, habibi, tahar}@ece.concordia.ca

Technical Report

March 2004

Abstract

In this work, we use Abstract State Machines (ASM) modeling language, AsmL, to define the semantics of the SystemC system level language. ASM provides an efficient methodology for formally specifying computing systems. The SystemC semantics we defined include the SystemC simulator and non–trivial SystemC components including FIFO channels, MUTEX channels, message queuing, request–grant protocol and SystemC FIFO hierarchical channels with handshake protocol. Also we present the semantics of design rules for SystemC channels including static and dynamic design rules checking. We used the executable language AsmL in order to build an abstract SystemC simulator, this allows the specification of SystemC designs in ASM and their execution using the provided SystemC ASM semantics. We believe that this work reduces the learning time and effort for understanding SystemC by providing an abstract executable simulator.

Contents

1 Introduction

Systems now are combinations of different types of components: microprocessors, DSPs, memories, software, etc. So there was a need for such language to fi ll the gap between hardware description languages (HDLs) and traditional programming languages, this language should combine together hardware and software modeling. SystemC was introduced as a new system level design and modeling language to overcome the problem of the growth in complexity and size of systems. SystemC comprises C++ class libraries and a simulation kernel used for creating behavioral and register transfer level (TRL) designs. SystemC can provide the common development environment needed to support software engineers working in C/C++ and hardware engineers working in HDLs such as Verilog or VHDL. The Abstract State Machines (ASM) methodology is a methodology for formally specifying computing systems (software, hardware, or mixed). ASM methodology is mathematically precise, yet general enough to be applicable to a wide variety of problem domains [4]. The ASM Thesis [5] asserts that any computing system can be described at its natural level of abstraction by an appropriate ASM. ASMs provides features to capture the behavioral semantics of programming and modeling languages, as a wide range of these languages were defi ned with this notion [6].

In this work, we introduce a formal and complete semantics of the main parts of SystemC language. It is complementary for the work of Müller *et. al.*[10] where an ASM based SystemC simulation semantics was introduced. We extend, however the defi nition to cover more complex components of SystemC and introduce a new defi nition for design rules of SystemC channels including static and dynamic design rules checking and we defi ne the semantics of SystemC simulator based on the defi nition of SystemC scheduler introduced in [10]. In the work presented before in this direction, tehre were fundamental and nontrivial parts of SystemC not well defi ned or still not defi ned at all, like, for example, *message queuing channel* and *request–grant protocol*. These SystemC components have methods that are necessary to be defi ned in order to understand the behavior of the these components during simulation, and to understand the operation of each one, and the way they communicate together. The simulation process was not faithfully defi ned, even though a defi nition of the scheduler part was presented. So defi ne SystemC simulator using ASM rules utilizing from the defi nition of SystemC scheduler, with minor modifi cation mentioned where it is.

We used the executable ASM specifi cation language AsmL [7] to model and execute SystemC simulator. The model can be executed on different tools that support AsmL language. Provided this model, we can defi ne our SystemC design on the model of the simulator and execute it. This will allow us to utilize what is availble the the tool that run AsmL specifi cation to test and verify our SystemC design[3].

The remainder of this paper is organized as follows. Section 2 is a description for the components of SystemC 2.0. In Section 3 we briefly describe Abstract State Machines. Section 4 gives an overview over related work on semantics of SystemC and the use of ASM in defi ning semantics of different languages. Section 5 provides the description of SystemC components under study, and the defi nition of its semantics in ASMs. Section 6 concludes this work.

2 SystemC

SystemC is a modeling language based on C++ that is intended to enable system level design in response to the need of a very fast executable specifi cations to validate and verify system concepts. It is a system−on−chip language representing functionality, communication, software and hardware at various system levels of abstraction. SystemC 2.0 release came to enable system-level modeling the RTL level of abstraction, including systems which might be implemented in software or hardware or some combination of the two [13].

SystemC 1.0 provides structural description features including modules and ports that can be used in systems design. In addition, there exist different data types to enable modeling hardware systems and processes to express concurrency. SystemC 2.0 introduces channels, interfaces, and events to enable communication and synchronization between modules or processes. An interface specifi es a set of access methods to be implemented within a channel where channels provides the implementation for these interfaces. An event is a flexible synchronization primitive that is used to construct other forms of synchronization. This will enable modeling hardware signals, queues, semaphores, memories and busses at the RTL level.

SystemC2.0 distinguishes two types of channels: *primitive channels* and *hierarchical channels*. the fi rst does not exhibit any visible structure, does not contain processes, and cannot access another primitive channel directly. Hierarchical channels, on the other hand, are modules that can have structure, contain other modules and processes, and access other channels.

Different channel types bring along different design rules. SystemC imposes rules on channels and the way they communicate. Design rules check should be carried out before simulation starts to ensure how many ports are connected and what the interface types are that these ports require. This is called *static* design rule checking. On the other hand, *dynamic* design rules checking is needed after the simulation has started to ensure that channels does not violate these rules during simulation time.

Most modeling languages, VHDL for example, use a simulation kernel. The purpose of the kernel is to ensure that parallel activities (concurrency) are modeled correctly. The behavior of the simulation should not depend on the order in which the processes are executed at each step in simulation time. The SystemC simulation kernel supports the concept of delta cycles. A delta cycle consists of an evaluation phase and an update phase. This is typically used for modelling primitive channels that cannot change instantaneously. By separating the two phases of evaluation and update, it is possible to guarantee deterministic behavior, because a primitive channel will not change value until the update phase occurs, it cannot change immediately during the evaluation phase.

SystemC simulation kernel contains a scheduler to determine the order of execution of processes within the design based on the event sensitivity of the processes and the event notifi cations which occur. It supports both software and hardware-oriented modeling. The semantics of this scheduler was completely defi ned using ASM rules [10] and denotational semantics [11]. Understanding the scheduler is necessary to understand the simulation process.

In a layered approach, the SystemC simulation kernel forms the base layer [2]. On top of the simulation kernel, SystemC implements dynamic sensitivity and events. This facilitates both the modeling at higher levels of abstraction as well as the creation of refi ned communication channels. The next layer contains the defi nition of channel types and interfaces and, ports. This layer implements the so-called interface method-call (IMC) scheme, which supports dynamic master/slave relationships between processes. Parts of SystemC 1.0 are implemented on top of the channels,

Table 1: SystemC Layered Approach

Property Remote Procedure Calls (RPC)
Signals
Channels, Interfaces and Ports
Events & Dynamic Sensitivity
SystemC Simulation Kernel

interfaces and ports layer. The IMC scheme is a generalization of the Remote Procedure Calls (RPC) scheme. So, the RPC scheme is implemented on top of the other four layers. Table 1 shows the approach [2].

3 Abstract State Machines

Sequential ASMs were introduced in [4] including distributed characteristics. We present here ASM features that are necessary to understand this work.

3.1 States

States are given as many–sorted first–order structures. A structure is given with respect to a signature which is a finite collection of function names, each of a fixed arity. The given structure fixes the syntax by naming sorts and functions. An algebra provides *domains* (i.e., carrier sets) for the sorts and a suitable symbol interpretation for the function symbols on these domains, which assigns a meaning to the signature. Therefore, a state is defined as an algebra of a given signature with *domains* and an interpretation for each function symbol.

A *vocabulary* is a finite collection of function names, each with a fixed arity. Every ASM vocabulary contains the following *logic symbols*: nullary function names *true, false, undef*, the equality sign, the names of the usual Boolean operations, and a unary function name *Bool*. Some function symbols (such as *Bool*) are tagged as *relations*.

A *state* S of vocabulary Γ is a non–empty set X (*the superuniverse of* S), together with interpretations of all function symbols in Γ over X. A function symbol f of arity r is interpreted as an r–ary operation over X; if $r = 0$, f is interpreted as an element of X. The interpretations of the function symbols *true*, *false*, and *undef* are distinct, and are operated upon by the Boolean operations in the usual way. The value *undef* is used to code functions whose value is outside the indicated domain.

3.2 Locations and Updates

A state transition into the next state occurs when dynamic functions change their evaluation. *Locations* and *updates* capture this notion.

A *location* of a state is a pair $loc = (f, \overline{a})$, where f is a dynamic function symbol and $\overline{a}$ is a tuple of elements in the domain of the function. The element $f(\overline{a})$ at a state is the *value* of the location $(f, \overline{a})$ in that state.

For changing values of locations the notion of an *update* is used. An *update* of a state is a pair $\alpha = (loc, val)$ where $loc = (f, \overline{a})$ is a location and val, the update value, is a value in the function domain. To fire an update at a state, the update value is set to the new value of the location. As a consequence, the overall dynamic function f is redefined to map the location onto the new value.

3.3 Transition Rules

Transition rules define the state transitions of an ASM. While terms denote values, transition rules denote *update sets*, which define the dynamic behavior of an ASM. At each state all update sets are fired simultaneously which causes a state change. All locations that are not referred to in the update sets remain unchanged.

ASM runs starting in a given initial state are determined by a closed transition rule declared to be the *program*. Basic transition rules are *skip*, *update*, *block*, and *conditional rules*.

The *skip* rule is the simplest transition rule. This rule specifies an "empty step". No function value is changed. It is denoted as

$$skip$$

The *update* rule is an atomic rule denoted as

$$f(t_1, t_2, \ldots, t_n) := t$$

It describes the change of interpretation of function f at the place given by $(t_1, t_2, \ldots, t_n)$ to the current value of term *t*.
A *block* rule is a group of transition rules. Sub-transitions rules within a block rule (R_1 and R_2 below) are fired simultaneously. All transition rules that specify the behavior of the ASM are grouped into a block, namely the *program*), indicating that all of them are fired simultaneously.

$$\begin{array}{l} block \\ \quad\quad R1 \\ \quad\quad R2 \\ endblock \end{array}$$

A *conditional rule* specifies a guarded execution.

$$\begin{array}{l} if\ guard\ then\ R1 \\ else\ R2 \\ endif \end{array}$$

Where $guard$ is a first order Boolean term. R_1 and R_2 denote arbitrary transition rules. The condition rule is fired in state S by evaluating the guard g in S, if it evaluates to *true* R_1 fires, otherwise R_2 fires.

The ASM executable language is called the Abstract state machine Language or AsmL. AsmL is integrated with Microsoft's software development environment including Visual Studio, Word, and Component Object Model (COM). AsmL effectively supports specification and rapid prototyping of different kinds of models.

4 Related work

In this section we first describe the previous work on formalizing semantics of SystemC, then we describe the use of ASM ito define semantics of different programming and modeling languages.

Salem [11] presented formal semantics of a synchronous subset of SystemC using denotational semantics. The delta cycle was formulated using function domains. Physical time was modeled on the clock period level. A description style based on defining two types of processes: Synchronous and Combinational was also proposed. The semantics of the SystemC methods and threads limited to this description style were defined. The evaluate and update phases of SystemC scheduler have been formulated for both timed and immediate notifications.

This work provides the description of the above parts only using general syntactic rules. It does not provide any specific definitions for basic SystemC components and processes; like *wait* or *notify*. The provided semantics, as all denotational semantics, were compact, precise, and providing a rigorous way to understand the SystemC as programming or modeling language. However, the introduced definitions were more abstract and less implementation–dependent than what a modeling language (C++ based) would require, because SystemC users care more for the behavior of the language, the implementation and interaction between different components, and how synchronization is maintained during simulation time.

Müller *et. al.* [8] first presented a semantics definition of SystemC.0 that covers method, thread, and clocked thread behavior. The semantics includes watching statements, signal assignment, and wait statements using the notion of Abstract State Machines. After that Müller [10] defined semantics of some parts of SystemC 2.0. These parts include signals updates for primitive channels with *write()* method only, *notify*, *notify_delay*, *cancel*, and *next_trigger* for SystemC events and dynamic sensitivity, *wait()* method for synchronization of threads, and finally SystemC scheduler.

This work nicely describes the above components using the theory of ASMs, specially SystemC scheduler, however, there are still fundamental parts of SystemC that were not defined, including some SystemC primitive and hierarchical channels, design rules for SystemC channels, and the semantics of SystemC simulator. There are nontrivial components introduced in SystemC 2.0 that their semantics should be defined in order to understand its behavior and how different components work and interact. Examples of these a *message queuing channel* and a *request–grant protocol*. These SystemC components implement methods that interact between each others and triggers other methods with dynamic or static sensitivity, they also have some synchronization to maintain the correct operation of these channels. The way of interaction and synchronization cannot be ignored when talking about operational semantics because it is necessary to understand the simulation process.

However, we consider our work complementary to the work presented before, since all together will provide a more comprehensive definition for SystemC components and the simulation semantics of SystemC. We present this work utilizing from many aspects of the definitions of some SystemC methods and SystemC scheduler as presented in [10].

ASMs have been extensively used in the definition of different modeling and hardware description languages (HDLs). Börger *et. al.* [1] presented a formal definition of the VHDL'93 simulator with ASMs. The specification was defined in terms of ASM transition rules modeling the relevant behavioral constructs of VHDL'93. Sasaki [12] provided a formal semantic analysis for Verilog–HDL and VHDL in order to give the simulation model especially focusing on signal scheduling and timing control mechanism. In another work for Müller *et. al.* [9] introduced a definition of the SpecC language semantics 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.

ASMs also have been used in describing the semantics of programming languages such as Java, C, C++, COBOL, SDL, Prolog, Process Description Language (PDL), SDL-2000, and SML. The complete references are available on ASM web site [6]

5 Semantics of SystemC

In this section we provide and ASM based semantics for SystemC channels, design rule checks, and SystemC simulator. Notification of events is achieved by calling the *notify()* method with no arguments which is called immediate notification. Processes sensitive to the event will run during the current evaluation phase. *notify()* can also be called with a zero time argument, which is called delta notification. Processes sensitive to the event will run during the evaluation phase of the next delta cycle. Finally *notify()* can be called with a non–zero time argument which is called timed notification. Processes sensitive to the event will run during the evaluation phase at some future simulation time. The *notify()* method cancels any pending notifications, and carries out various checks on the status of existing notifications. The SystemC simulation kernel does not impose any order on processes that are simultaneously ready–to–run. In our definition, we treate different kinds of notifications sperately. Immediate notifications are not shown in this definition since they are implied in the execution of a method, which we treate abstractly here. Timed and delta notifications are shown below within the simulator definition. Events notify type can be immediate ($NONE$), at specific simulation time ($TIMED$), or at the next SystemC delta cycle ($DELTA$). The vocabulary of our semantics includes τ which is the set of timed events, λ the set of delta events, δ SystemC zero time, and T_c the current simulation time.

The description of SystemC components is based on the Functional Specifications for SystemC [2] and SystemC website [14].

5.1 SystemC Channels

Channel accesses can be decomposed into evaluation step and update step. This will enable handling simultaneous actions in both simple and specialized primitive channels such as read and write operations in the case of a queue or simultaneous bus requests issued by several bus masters. The evaluate step is executed when an interface method of the channel is invoked. This interface method is executed in the context of the calling process during the evaluate phase of a delta cycle. The update step, if requested, is executed in a separate context during the update phase of a delta cycle. The update step is required in cases where: simultaneous actions have to be serialized, or any form of arbitration or resolution is required in order to make the channel access deterministic. The update step is carried out when necessary only.

5.1.1 SystemC signal

We provide the semantics of the *update* method only for this type. The method *read* is trivial, and method *write* was faithfully described in [10]. The *update* method as described in SystemC documentation ensures deterministic behavior in the case of simultaneous read and write actions. If the new value of the signal is equal to the current value, then no update is needed. If the value

changed event ($ValueChangedEvent$)type is $TIMED$ then we remove this event from the timed events set. After that, we add it to the pending events set, set its time to next SystemC delta cycle, and finally change its type to $DELTA$ event type.

$$
\begin{aligned}
&update() \equiv \\
&\quad if\ CurrentValue \neq NewValue\ then \\
&\quad\quad CurrentValue := NewValue \\
&\quad\quad if\ ValueChangedEvent.Type = TIMED\ then \\
&\quad\quad\quad \tau := \tau - \{ValueChangedEvent\} \\
&\quad\quad endif \\
&\quad\quad if\ ValueChangedEvent \notin \lambda\ then \\
&\quad\quad\quad\quad \lambda := \lambda \cup \{ValueChangedEvent\} \\
&\quad\quad endif \\
&\quad\quad time(ValueChangedEvent) := T_c + \delta \\
&\quad\quad ValueChangedEvent.Type := DELTA \\
&\quad endif
\end{aligned}
$$

5.1.2 SystemC FIFO

The SystemC FIFO primitive channel is part of SystemC 2.0. It provides methods for blocking and non–blocking I/O and some additional methods to query the status of the channel. The blocking methods ensure the dynamic sensitivity where we can not explicitly declare the calling process to be sensitive to the FIFO.

The *read* method checks first if the channel is empty, then it waits until a data writing event ($dataWrittenEvent$) takes place. Data is then popped out of channel, and the channel is scheduled for update at the next delta cycle. The flag $dataReadFlag$ is used to perform the update phase in the channel later, while the $updateRequested$ flag is used to check if the channel has already been added to the channels update array, it is reset to *false* after the update is performed in the primary channel base class. The semantics of *read* method is:

$$
\begin{aligned}
&read(data) \equiv \\
&\quad while\ EmptyFIFO\ wait\ (dataWrittenEvent) \\
&\quad data := head\ (FIFO) \\
&\quad FIFO := tail\ (FIFO) \\
&\quad if\ !\ updateRequestedthen \\
&\quad\quad channelsUpdateArray := channelsUpdateArray \cup \{self\} \\
&\quad endif \\
&\quad dataReadFlag := true \\
&\quad updateRequested := true
\end{aligned}
$$

Where the semantics of *wait*(sc_even) is faithfully defined in [10], and the semantics of $while$ statement can be defined as :

$$
\begin{aligned}
&while\ b\ do\ c \equiv \\
&\quad if\ b\ then \\
&\quad\quad execute\ c \\
&\quad\quad step := while\ b\ do\ c
\end{aligned}
$$

$else\ skip$

where b is a Boolean expression and c is an executable command.

The *write* method similarly behaves like *read* method. Its semantics of is defined as follows:

$$
\begin{aligned}
&write(data)\ \equiv \\
&\quad while\ FullFIFO\ wait\ (dataReadEvent) \\
&\quad FIFO := FIFO \cup\ data \\
&\quad if\ !\ updateRequestedthen \\
&\qquad channelsUpdateArray := channelsUpdateArray\ \cup\ \{self\} \\
&\quad endif \\
&\quad dataWrittenFlag := true \\
&\quad updateRequested := true
\end{aligned}
$$

Non–blocking read and write do wait not for a read or write event if the channel cannot perform the operation, instead, it just suspends the method, otherwise, if the operation is doable, data is popped out of channel in case of *read* or pushed into channel in case of *write*, and then the channel is scheduled for update in the next delta cycle. Below is the semantics of non–blocking read, non-blocking write can be defined similarly.

$$
\begin{aligned}
&nonBlockingRead(data)\ \equiv \\
&\quad if\ EmptyFIFO\ then\ skip\ \rightarrow\ false \\
&\quad else \\
&\quad\ block \\
&\qquad data := head\ (FIFO) \\
&\qquad FIFO := tail\ (FIFO) \\
&\quad if\ !\ updateRequestedthen \\
&\qquad channelsUpdateArray := channelsUpdateArray\ \cup\ \{self\} \\
&\quad endif \\
&\qquad updateRequested := true \\
&\quad\ endblock\ \rightarrow\ true \\
&\quad endif
\end{aligned}
$$

The notion $c \rightarrow\ v$ means: after executing the command c the expression evaluates to the value v.

The method *update* notifies the set of processes are that are suspended on a read or write even when any of these occurs, these processes will resume execution at the next delta cycle. The method checks first if data has been written to the channel, by checking $dataWrittenFlag$, then it removes the data written event ($dataWrittenEvent$) from timed events if it is there, add it to the set of delta events (λ), set its time to next delta cycle, and finally set the event type into $DELTA$ event. Similar steps take place if data has been read from the channel.

$$
\begin{aligned}
&update()\ \equiv \\
&\quad if\ dataWrittenFlagthen \\
&\quad\ if\ dataWrittenEvent.Type\ =\ TIMED\ then \\
&\qquad \tau := \tau\ -\{dataWrittenEvent\}
\end{aligned}
$$

$$
\begin{array}{l}
\quad endif \\
\quad if\ dataWrittenEvent \notin \lambda\ then \\
\quad\quad \lambda := \lambda \cup \{dataWrittenEvent\} \\
\quad endif \\
\quad time(dataWrittenEvent) := T_c + \delta \\
\quad dataWrittenEvent.Type := DELTA \\
endif \\
if\ dataReadFlag\ then \\
\quad if\ dataReadEvent.Type\ =\ TIMED\ then \\
\quad\quad \tau := \tau\ - \{dataReadEvent\} \\
\quad endif \\
\quad if\ dataReadEvent \notin \lambda\ then \\
\quad\quad \lambda := \lambda \cup\ \{dataReadEvent\} \\
\quad endif \\
\quad time(dataReadEvent) := T_c + \delta \\
\quad dataReadEvent.Type := DELTA \\
endif
\end{array}
$$

5.1.3 SystemC MUTEX

This channel is part of SystemC 2.0 only. It performs FIFO queuing of pending requests and issues a warning if multiple requests are issued during the same delta cycle. The MUTEX (mutual exclusion) channel is owned into only one process during any delta cycle at simulation time. If the channel is not locked, it is given to the first process that issues a request. Only the process that locked the MUTEX is allowed to unlock it. Dynamic sensitivity is used to suspend processes that request locking the channel when it is already locked, and and later resume them. The SystemC MUTEX primitive channel can be used to model shared variables, either through inheritance by deriving a shared variable channel from the MUTEX channel or by convention where access to a certain variable is protected by a MUTEX.

The *lock* method keeps trying to take ownership of the channel while it is in use by another, the process will wait on freeing MUTEX channel event ($MutexFreeEvent$), and then check if the target channel is still in use. When it is freed, the process takes ownership of the channel, and it can unlock it later. The method, *trylock* tries one time to take ownership of the channel, it either fails or succeeds. The *unlock* method frees the channel (if process is the current owner of the channel) and triggers other processes that are suspended on freeing channel event in the next delta cycle.

$$
\begin{array}{l}
lock() \equiv \\
\quad while\ InUse\ wait\ (MutexFreeEvent) \\
\quad MutexOwner := CurrentProcess
\end{array}
$$

$$
\begin{array}{l}
tryLock() \equiv \\
\quad if\ InUse\ then\ skip\ \rightarrow\ false \\
\quad else \\
\quad\quad MutexOwner := CurrentProcess\ \rightarrow true
\end{array}
$$

$$
\begin{array}{l}
\quad endif \\
\\
unlock() \equiv \\
\quad if\ MutexOwner \neq CurrentProcess\ then\ skip \rightarrow false \\
\quad else \\
\quad\quad block \\
\quad\quad\quad MutexOwner := NULL \\
\quad\quad\quad if\ MutexFreeEvent.Type = TIMED\ then \\
\quad\quad\quad\quad \tau := \tau - \{MutexFreeEvent\} \\
\quad\quad\quad endif \\
\quad\quad\quad if\ MutexFreeEvent \notin \lambda\ then \\
\quad\quad\quad\quad \lambda := \lambda \cup \{MutexFreeEvent\} \\
\quad\quad\quad endif \\
\quad\quad\quad time(MutexFreeEvent) := T_c + \delta \\
\quad\quad\quad MutexFreeEvent.Type := DELTA \\
\quad\quad endblock \rightarrow true \\
\quad endif
\end{array}
$$

5.1.4 Message Queue

This channel illustrates a channel which cannot be used without ports and supports per–port configuration information through channel attribute and in a specialized port. The message queue channel uses two port attributes: a priority and a non-blocking flag. The priority attribute is stored in the specialized port, while the non-blocking flag attribute is stored in the channel. Both attributes can be changed dynamically, i.e., methods are provided in the interface and the specialized port to read and write the attributes.

The message queue channel provides two basic methods: *send()* and *receive()*. In both methods, the priority port attribute is used as request priority, when multiple processes are waiting to queue a message or dequeue it. In the *send()* method, the priority attribute is also used as message priority for the underlying priority queue that stores the messages. The nonblocking flag port attribute is used to make the *send()* and *receive()* methods either blocking (default) or non–blocking.

The *send* method first checks if there is no space available in the messages priority queue ($messagesPQ$) or if there are any receive requests pending in the send request ports priority queue ($SendReqPortsPQ$). If this is the case, then the port sending the message is enqueued according to its priority and mapped into a newly created send acknowledgment event ($sendAckEvent$) in the send acknowledgments event array ($SendAckEventArray$). This map is necessary for the update phase later in order to notify channels that are sensitive to the event corresponding to the port sending the message. After that, the *send* method is suspended waiting on a send acknowledgment event ($sendAckEvent$). At this point the message priority queue is ready to enqueue the message and the channel is scheduled for update at the next delta cycle.

$$
\begin{array}{l}
send(port, msg, priority) \equiv \\
\quad if\ messagesPQ.SIZE \not< N \\
\quad or\ SendReqPortsPQ.SIZE \neq 0\ then
\end{array}
$$

$SendReqPortsPQ.insert(port, priority)$
$SendAckEventArray := SendAckEventArray \cup$
$\{(port, sendAckEvent)\}$
$wait(sendAckEvent)$
$endif$
$messagesPQ.insert(msg, priority)$
$channelsUpdateArray := channelsUpdateArray \cup \{self\}$

The *receive* method behaves in similar way; if there are no messages in the message priority queue ($messagesPQ$), or there are any receive requests already pending in the receive request ports priority queue ($ReceiveReqPortsPQ$), then the port receiving the message is enqueued according to its priority and then mapped into a newly created receive acknowledgment event ($receiveAckEvent$) in the receive acknowledgments event array ($ReceiveAckEventArray$). The *receive* method is then suspended waiting on a receive acknowledgment event ($receiveAckEvent$). At this point, the channel is ready to take the message with the highest priority in the messages priority queue, and the channel is scheduled for update at the next delta cycle.

$receive(port, msg, priority) \equiv$
$if\ messagesPQ.SIZE \not> 0$
$or\ ReceiveReqPortsPQ.SIZE \neq 0\ then$
$ReceiveReqPortsPQ.insert(port, priority)$
$ReceiveAckEventArray := ReceiveAckEventArray \cup$
$\{(port, receiveAckEvent)\}$
$wait(receiveAckEvent)$
$messagesPQ.extract(msg)$
$channelsUpdateArray := channelsUpdateArray \cup \{self\}$

The *update* method verifies that there is apace in the messages priority queue and there are send requests in send requests priority queue ($SendReqPortsPQ$), so that it can extract the message with the highest priority, remove the send acknowledgment event from these events array, and assign the corresponding event (mapped before to the port) to a newly created event ($sendAckEvent$). Then delta cycle notification is performed. The same procedure is done for *receive* operation.

$update() \equiv$
$if\ messagesPQ.SIZE < N$
$and\ SendReqPortsPQ.SIZE \neq 0\ then$
$SendReqPortsPQ.extract(port)$
$sendAckEvent := SendAckEventArray.remove(port)$
$if\ sendAckEvent.Type = TIMED\ then$
$\tau := \tau - \{sendAckEvent\}$
$endif$
$if\ sendAckEvent \notin \lambda\ then$
$\lambda := \lambda \cup \{sendAckEvent\}$
$endif$
$time(sendAckEvent) := T_c + \delta$

$$
\begin{aligned}
&\quad sendAckEvent.Type := DELTA \\
&endif \\
&if\ messagesPQ.SIZE > 0 \\
&and\ ReceiveReqPortsPQ.SIZE \neq 0\ then \\
&\quad ReceiveReqPortsPQ.extract(port) \\
&\quad receiveAckEvent := ReceiveAckEventArray.remove(port) \\
&\quad if\ receiveAckEvent.Type = TIMED\ then \\
&\quad\quad \tau := \tau - \{receiveAckEvent\} \\
&\quad endif \\
&\quad if\ receiveAckEvent \notin \lambda\ then \\
&\quad\quad \lambda := \lambda \cup \{receiveAckEvent\} \\
&\quad endif \\
&\quad time(receiveAckEvent) := T_c + \delta \\
&\quad receiveAckEvent.Type := DELTA \\
&endif
\end{aligned}
$$

5.1.5 Request Grant Protocol

This protocol deals with two SystemC channels, master and slave, that are sharing one port. Only one master and one slave can be connected to the port at one time. During a $writeMaster$ operation, the method verifies that the channel is not already requested, otherwise, it waits on the no–request event ($noRequestEvent$). Data then is stored in the an update variable ($projectedValue$), projected signals request ($projectedSignals.REQUEST$) is enabled (which means that a master channel has already written to this port), and finally channel is added to channels update array for future update.

$$
\begin{aligned}
&writeMaster(data) \equiv \\
&\quad if\ currentSignals.REQUEST\ then \\
&\quad\quad wait(noRequestEvent) \\
&\quad endif \\
&\quad projectedValue := data \\
&\quad projectedSignals.REQUEST := true \\
&\quad channelsUpdateArray := channelsUpdateArray \cup self
\end{aligned}
$$

To perform a ***read*** operation, the channel has to be granted to the master port. So it waits on a grant event ($grantEvent$) if the current signals grant ($currentSignals.GRANT$) is disabled. After that the port reads the current value, projected signals grant and request are disabled, and finally channel is scheduled for later update.

$$
\begin{aligned}
&readMaster(data) \equiv \\
&\quad if\ !\ currentSignals.GRANT\ then \\
&\quad\quad wait(grantEvent) \\
&\quad endif \\
&\quad data := currentValue \\
&\quad projectedSignals.REQUEST := false
\end{aligned}
$$

$projectedSignals.GRANT := false$
$channelsUpdateArray := channelsUpdateArray \cup self$

The methods *readSlave* and *writeSlave* behaves in synchronization with *readMaster* and *writeMaster* as shown below.

$writeSlave(data) \equiv$
$if\ !\ currentSignals.REQUEST\ then$
$wait(RequestEvent)$
$endif$
$projectedValue := data$
$projectedSignals.GRANT := true$
$channelsUpdateArray := channelsUpdateArray \cup self$

$readSlave(data) \equiv$
$if\ !\ currentSignals.REQUEST\ then$
$wait(requestEvent)$
$endif$
$data := currentValue$

After updating channel with new values, the *update* method notifies processes that are sensitive to request even ($requestEvent$) when there is one, and processes that are sensitive to no–request event ($noRequestEvent$) when there is no request. It also notifies processes that are sensitive to grant event ($grantEvent$) when there is one to be updated in the next delta cycle.

$update \equiv$
$currentSignals := projectedSignals$
$currentValue := projectedValue$
$if\ currentSignals.REQUEST\ then$
$if\ requestEvent \notin \lambda\ then$
$\lambda := \lambda \cup \{requestEvent\}$
$endif$
$time(requestEvent) := T_c + \delta$
$else$
$if\ noRequestEvent \notin \lambda\ then$
$\lambda := \lambda \cup \{noRequestEvent\}$
$endif$
$time(noRequestEvent) := T_c + \delta$
$endif$
$if\ currentSignals.GRANT\ then$
$if\ grantEvent \notin \lambda\ then$
$\lambda := \lambda \cup \{grantEvent\}$
$endif$
$time(grantEvent) := T_c + \delta$

$endif$

5.1.6 FIFO with handshake protocol

This hierarchical FIFO channel uses the same interface as the primitive FIFO channel. The channel contains a module that is clocked and uses a handshake protocol for reading and writing. The channel corresponds the handshake to read and write requests at a positive clock edge if there is such a request. Hierarchical channels are used when users would want to be able to explore the underlying structure. Also when channels contain processes or other channel.

We provide the semantics of the two basic methods of these channels types; *read* and *write* methods for the SystemC FIFO clocked channel. This channel is connected into read and write clocked handshake channels. The *write* method first checks if write is enabled ($writeEnableSignal$) in order to proceed, if not, the process is suspended on value chancing event ($ValueChangedEvent$) of the write enable signal. After that data is written to write handshake channel ($writeDataSignal$), write request signal ($writeRequest$) is enabled, and the process is suspended on a value change event of the write acknowledgment signal ($writeAckSignal$). When the event triggers this process, it disables the write request and terminates.

$$
\begin{aligned}
&write(data) \equiv \\
&\quad if\ !\ writeEnableSignal\ then \\
&\quad\quad wait(writeEnableSignal.ValueChangedEvent) \\
&\quad endif \\
&\quad writeDataSignal := data \\
&\quad writeRequest := true \\
&\quad wait(writeAckSignal.ValueChangedEvent) \\
&\quad writeRequest := false
\end{aligned}
$$

Similarly, the *read* method first checks if read is enabled ($readEnableSignal$) in order to proceed, if not, the process is suspended on a value changing event ($ValueChangedEvent$) of the read enable signal. Then, the read request signal ($readRequest$) is enabled, and the process is suspended on a value change event of the read acknowledgment signal ($readAckSignal$). When resumed, the process can read data from read handshake channel ($readDataSignal$), and the read request is disabled.

$$
\begin{aligned}
&read() \equiv \\
&\quad if\ !\ readEnableSignal\ then \\
&\quad\quad wait(readEnableSignal.ValueChangedEvent) \\
&\quad endif \\
&\quad writeRequest := true \\
&\quad wait(readAckSignal.ValueChangedEvent) \\
&\quad data := readDataSignal \\
&\quad readRequest := false
\end{aligned}
$$

The signals: $writeEnableSignal$, $writeDataSignal$, $writeRequest$ and $writeAckSignal$ are all used in the clocked handshake write channel when writing data to the FIFO channel, (and

some are updated in the methods implements the channel. Also the signals: $readEnableSignal$, $readDataSignal$, $readRequest$, and $readAckSignal$ are all used in the clocked handshake read channel. These variable will maintain synchronization between the read and write channels.

5.2 Design Rules

Each SystemC channel requires specific number of ports to be connected to it, or arbitrarily unlimited in some cases. When a channel is created, it has to pass design rules check in order to make sure that the number of ports connected is allowed. This is called *static* design rules checking. on the other hand, a channel may impose that only one process can perform an I/O operation at one time, so the channel has to pass design rules checking when processes access the channel. This is called *dynamic* design rules checking.

The sc_signal channel demonstrates how to do dynamic design rule checking in addition to static design rule checking. During static design rule checking, the channel makes sure that only one writer port is attached. If the type of the port to be connected ($port.TYPE$) is input port (IN) or input/output port ($INOUT$), then the channel asserts that no port is already connected to its output port ($outputPort$) and the port is connected. If, on the other hand, the port type is output port, then it can be connected without any check.

$$
\begin{aligned}
&SignalsStaticDesignRule(port) \equiv \\
&\quad if\ port.TYPE = IN \\
&\quad or\ port.TYPE = INOUT\ then \\
&\qquad if\ outputPort \neq NULL\ then \\
&\qquad\quad outputPort := port \\
&\qquad else\ ERROR \\
&\qquad endif \\
&\quad else \\
&\qquad skip \\
&\quad endif
\end{aligned}
$$

During dynamic design rule checking, the channel checks that there is only one driver process writing to the channel. If a process ($currentProcess$), is trying to write to the channel, then it should be verified that there is no driver process ($driverProcess$) accessing the channel, or it is the driver process that is trying to write to the channel.

$$
\begin{aligned}
&SignalsDynamicDesignRule() \equiv \\
&\quad if\ driverProcess = NULL\ then \\
&\qquad driverProcess := currentProcess \\
&\quad else \\
&\qquad if\ driverProcess \neq currentProcess\ then \\
&\qquad\quad ERROR \\
&\qquad endif \\
&\quad endif
\end{aligned}
$$

The MUTEX channel allows only one owner and it has to pass *dynamic* design rules checking when any process tries to take ownership of the channel, the method *trylock* previously defined

implies this rule checking.

SystemC FIFO channel and clocked handshake FIFO channel require that only one reader and one writer be connected to the channel, so only static design rules checking is needed. Similarly, in the request–grant protocol, only one master and one slave channel can be connected at anytime, so only static design rule checking is needed. We show below the static design rules for the request–grant protocol.

$$
\begin{aligned}
&RGProtocolDesignRule(port, type) \equiv \\
&\quad if\ type == MASTER\ then \\
&\qquad if masterPort \neq NULL\ then \\
&\qquad\quad ERROR \\
&\qquad else \\
&\qquad\quad masterPort := port \\
&\qquad endif \\
&\quad else \\
&\qquad if slavePort \neq NULL\ then \\
&\qquad\quad ERROR \\
&\qquad else \\
&\qquad\quad slavePort := port \\
&\qquad endif \\
&\quad endif
\end{aligned}
$$

The message queue channel allows any number of ports to be connected, so there are no design rules check needed. This design rule is simply defined with the $skip$ ASM rule.

$$
\begin{aligned}
&MessageQueueDesignRule \equiv \\
&\qquad skip
\end{aligned}
$$

5.3 SystemC Simulator

The SystemC simulation kernel uses the concept of delta cycles to modelling primitive channels that cannot change instantaneously and ensure the deterministic behavior of the components. In SystemC however, the language allows non–determinism. For hardware modelling, this is unacceptable. But for software modelling, this might represent a system where a shared variable is used, and where non–determinism is not important, what is necessary is to guarantee that two processes cannot simultaneously access the variable. So, in software modeling, it is useful to be able to cause a process to run without executing the update phase. This requires events to be notified immediately through immediate notification which may cause non–deterministic behaviour. Some concepts such as mutual exclusion (MUTEX) or semaphores to cope with such situations.

We define below SystemC simulator with ASM rules. A sequence of steps is used to clarify the definition, each step is defined and explained below. During simulation time, simulation status can be one of the following: SC_SIM_OK, SC_SIM_ERROR, or $SC\_SIM\_USER\_STOP$.

$$
\begin{aligned}
&Simulator \equiv \\
&\qquad initialize
\end{aligned}
$$

$toggleRunnableProcesses$
$Schedule$
$CheckSimStatus$
$processTimedEvents$
$terminate$

The initialize step first checks for a user stop request, then it sets to simulation end time ($SimulationEndTime$) according to the required simulation duration parameter. Then it prepares all method processes and threads processes for simulation ($P.prepareForSimulation()$) by creating a coroutine package and allocating necessary stack space in memory. The runnable processes table ($RunnableProcessesTable$) contains two queues of threads handlers (push queue and pop queue) and similarly two queses of method handlers. One queue is used to push scheduled processes at current delta cycle resulted from immediate notification, and the other one to hold processes currently being executed. The operation $RunnableProcessesTable.initialize()$ just resets all these queues to its initial state where they are assumed to be empty. The operation $toggleRunnableProcesses$ alternates between these two queues, in other words, it change the current push queue into a pop queue, and reset the current pop queue. This enables the simulator to distinguesh between processes currently executed and processes triggered by events with immediate notification.

After that, we pushs all methods into runnable methods queue and all threads into runnable threads queue, and finally we process delta notifications by triggering all sensitive processes for delta events.

$$
\begin{aligned}
initialize \equiv\ & \\
& CheckSimStatus \\
& if\ simStatus \neq SC\_SIM\_OK\ then \\
& \quad step := terminate \\
& endif \\
& SimulationEndTime := \\
& \quad if\ duration = \delta\ then\ MAX\_TIME \\
& \quad else\ T_c\ +\ duration \\
& endif \\
& \forall P \in THREADS \cup CTHREADS :\ P.prepareForSimulation() \\
& RunnableProcessesTable.initialize() \\
& updateChannels \\
& \forall M \in METHODS :\ RunnableProcessesTable.push(M) \\
& \forall Th \in THREADS :\ RunnableProcessesTable.push(Th) \\
& \forall e \in \lambda :\ e.trigger()
\end{aligned}
$$

The $CheckSimStatus$ step just checks for a simulation error, or a user stop request.

$$
\begin{aligned}
CheckSimStatus \equiv\ & \\
& if\ simStatus\ =\ SC\_SIM\_USER\_STOP \\
& \ or\ simStatus\ =\ SC\_SIM\_ERROR\ then \\
& \quad step := terminate \\
& endif
\end{aligned}
$$

In the $processTimedEvents$ step, we first update the current simulation time, if there are no more timed notifications, then T_c is set to δ, otherwise, it is the time of the event in τ smaller than all other events times. Then all timed events are processed, if there are any runnable processes and we did not exceed simulation time, then we resume executing runnable processes by toggeling runnable processes table. However, if, after processing timed events, there are no runnable processes, we advance simulation time, and process timed events again.

$$
\begin{aligned}
&processTimedEvents \equiv \\
&\quad T_c := \ if\ \tau\ = \emptyset\ then\ \delta\ else\ nextTime(\tau)\ endif \\
&\quad \forall e\ \in \tau\ : \\
&\qquad if\ (time(e)\ =\ T_c\ then \\
&\qquad\quad e.trigger() \\
&\qquad endif \\
&\quad if\ RunnableProcesses.SIZE\ =\ 0 \\
&\quad and\ T_c\ \neq\ SimulationEndTime\ then \\
&\qquad step := processTimedEvents \\
&\quad endif \\
&\quad if\ T_c\ \neq\ SimulationEndTime\ then \\
&\qquad step := toggleRunnableProcesses \\
&\qquad step := terminate \\
&\quad endif
\end{aligned}
$$

$$
\begin{aligned}
&nextTime(\tau)\ \equiv \\
&\quad time(event) :\ event\ \in\ \tau, \forall e\ \in\ \tau, time(event)\ \leq time(e)
\end{aligned}
$$

The schedular semantics, quoted from [10], is shown below to clarify the full simulation steps. The semantics of each step are also defined in the same reference. However, the last two step $CheckEvents$ and $AdvanceTime$ do not distinguish between timed and delta notifications, so we just define the delta events checks step ($ProcessDeltaEvents$) within the scheduler, as implemented in the source code, and we move the $AdvanceTime$ step and define it within the timed events check above.

$$
\begin{aligned}
&Schedule\ \equiv \\
&\quad ResumeProcess \\
&\quad CheckReadyToRun \\
&\quad UpdateChannels \\
&\quad CheckEvents \\
&\quad AdvanceTime
\end{aligned}
$$

So, to be consistant with the overall simulation process and be able to define delta and timed notifications we would define the scheduler as follows:

$$
\begin{aligned}
&Schedule\ \equiv \\
&\quad ResumeProcess \\
&\quad CheckReadyToRun \\
&\quad UpdateChannels
\end{aligned}
$$

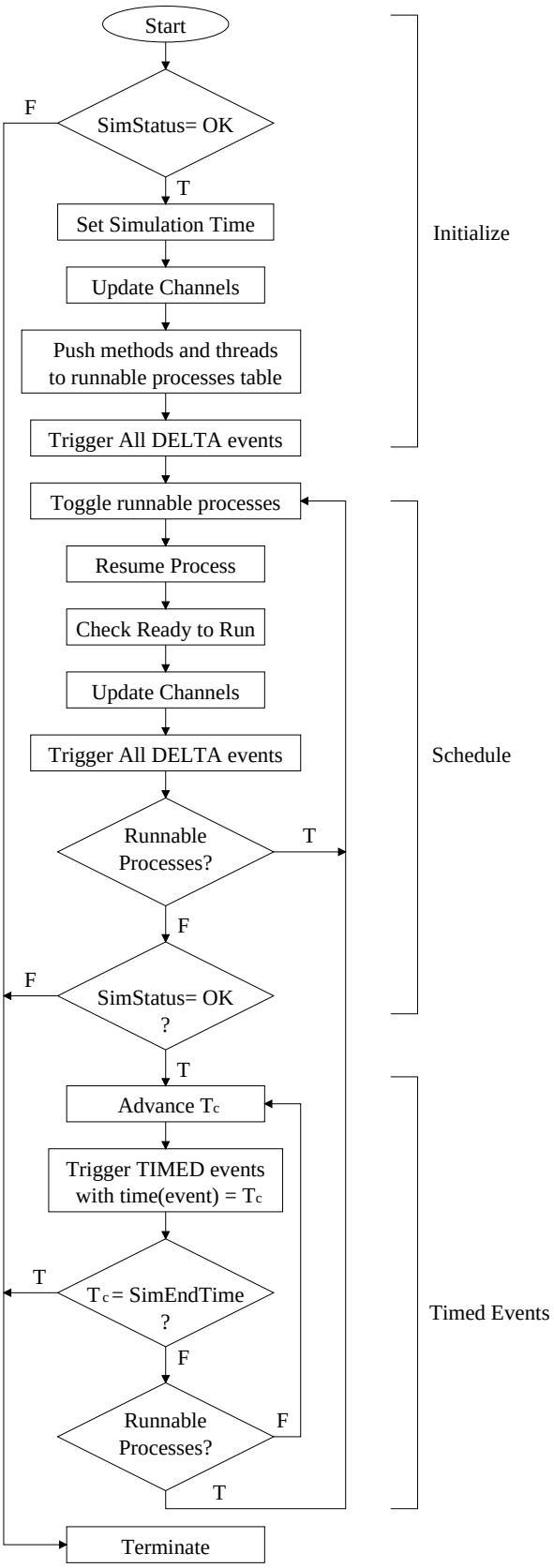

Figure 1: The SystemC Simulator

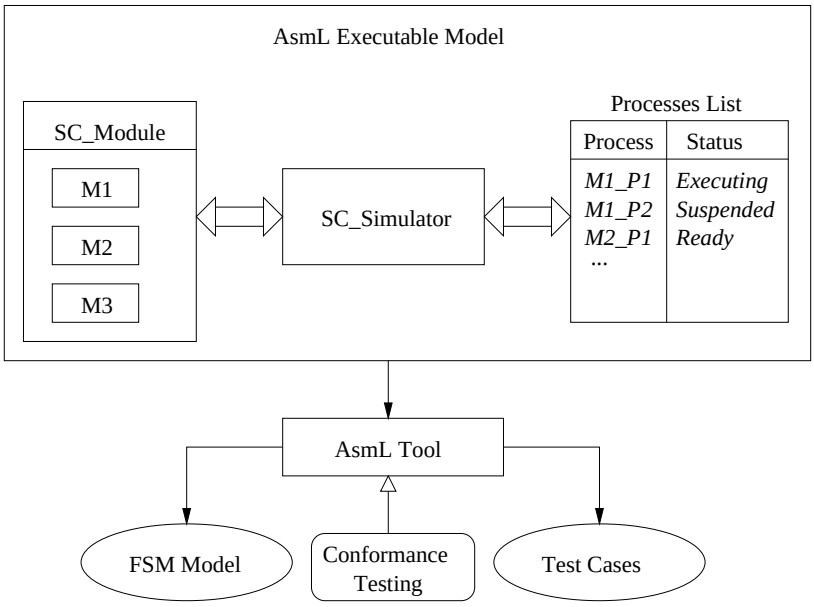

Figure 2: Executing an abstract AsmL model for the semantics of SystemC simulator

$$ProcessDeltaEvents$$

The $ProcessDeltaEvents$ steps triggers all event in λ and check if runnable processes table contains any processes to execute, it toggles queues and go to $ResumeProcess$ step, otherwise, it goes to $CheckSimStatus$ up before processing timed notifi cations.

$$\begin{aligned}
&processDeltaEvents \equiv \\
&\quad \forall e \in \lambda : \\
&\qquad e.trigger() \\
&\quad if\ RunnableProcesses.SIZE = 0\ then \\
&\qquad step := ResumeProcess \\
&\quad else \\
&\qquad step := CheckSimStatus \\
&\quad endif
\end{aligned}$$

Figure 1 shows a flow diagram for SystemC simulation process.

We chose the AsmL language to model the SystemC simulator in order to use it as an underlying engine to execute the abstract AsmL model of the design. The main purpose of this execution is to generate FSMs for the model under test. The .NET format can also be generated if the appropriate compiler is used (not provided by AsmL tester). We can generate test suites and conduct conformance tests on the basis of a model using the AsmL Tester tool. Figure 2 shows the general framework for semantics execution.

6 Conclusion

We have provided a faithful and common formal semantics for SystemC main parts including primitive and hierarchical channels, SystemC design rules, and SystemC simulator. The Abstract

State Machines formalism was used in order to define the semantics of these components. We choose ASMs because of the features they provide in order to define operational semantics of wide range of programming languages and hardware description languages.

We built our work upon previous work on definition of SystemC scheduler and basic methods for some SystemC components, however, we extended the definition to cover complex parts of the language, and we used the definition of scheduler in order to present a definition of SystemC simulator. We then defined an abstract SystemC simulator using ASM executable language AsmL, where SystemC design can be embedded to the model and executed. Then supporting tools for AsmL can be used for conformance testing, test case generation, or FSM generation for the design.

Our model helps the accurate understanding on the semantic interoperability for system level designers. It also reduces the learning time and effort by dealing with concise but intuitive and understandable transition rules. While the rules and explanations presented above are longer than what can be achieved with other semantics formalisms, the readability of these rules is substantially better, specially with a minimal amount of notational overhead. Such explanations and implementation details can provide a better basis for explaining and understanding a modeling language such as SystemC.

References

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

[2] Functional Specification for SystemC 2.0. Synopsis Inc. April 2002.

[3] W. Grieskamp, L. Nachmanson, N. Tillmann and M. Veanes. Test Case Generation from AsmL Specifications, In E. Borger, A. Gargantini and E. Recobene (eds.), Abstract State Machines - Advances in Theory and Applications, LNCS 2589, Springer Verlag, 2003, pp 413–414.

[4] Y. Gurevich. Evolving Algebras 1995: Lipari Guide. In E. Börger (ed.), Specification and Validation Methods, Oxford University Press, 1995.

[5] Y. Gurevich, Sequential Abstract State Machines Capture Sequential Algorithms. ACM Transactions on Computational Logic, vol. 1, no. 1, July 2000, pp. 77–111.

[6] J. Huggins. Abstract State Machines website. http://www.eecs.umich.edu/gasm/.

[7] AsmL for Microsoft .NET (version 2.1.5.7 or higher), Software Distribution. Containing Tools, Samples and Documentation. Downloadable at http://www.research.microsoft.com/foundations/asml.

[8] W. Müller, J. Ruf, D. Hofmann, J. Gerlach, Th. Kropf, Th., and W. Rosenstiel. The Simulation Semantics of SystemC. In Proc. of Design, Automation and Test in Europe (DATE'01), IEEE CS Press. Munich, Germany. 2001.

[9] W. Müeller, R. Dölemer, A. Gerstlauer. The Formal Execution Semantics of SpecC. Proc. of the International Symposium on Systems Sythasis (ISSS02), Oct 2002, Nagoya, Japan.

[10] W. Müller, J. Ruf, and W. Rosenstiel. SystemC Methodologies and Applications. Kluwer Academic Publishers. 2003.

[11] A. Salem. Formal Semantics of Synchronous SystemC.Proc. Design, Automation and Test in Europe Conference and Exposition (DATE'03), IEEE Computer Society. Munich, Germany. March 2003. pp. 10376-10381.

[12] H. Sasaki. A formal semantics for Verilog-VHDL simulation interoperability by abstract state machine. Proc. of the conference on Design, automation and Test in Europe. Munich, Germany. January 1999.

[13] S. Swan, An Introduction to System Level Modeling in SystemC 2.0. Open SystemC Initiative (OSCI), Cadence Design Systems, Inc. May 2001.

[14] SystemC website. http://www.systemc.org.