Enabling SystemC Verification 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

May 2004

Abstract

SystemC is a system level language recently proposed to raise the abstraction level for embedded systems design and verification. We propose a verification methodology for SystemC designs based on a combination of static code analysis and SystemC semantics described with abstract state machines (ASM). We abstract the source SystemC design into hypergraphs to keep an abstract (simplified) view of the design including only processes status, activation conditions and order of execution. This latter is then modeled with ASMs and compiled with the AsmL tool in order to generate a finite state machine that can be used for formal verification by external tools linked to ASM, such as model checkers or theorem provers. We can also generate a .NET representation of the abstracted SystemC model to guide test vectors generation and perform coverage analysis within the .NET framework. Our approach solves a number of problems classical verification techniques do face when used with SystemC through the efficient handling of the object–oriented aspect of SystemC and the complexity of its simulation environment.

1 Introduction

Today's systems combine different types of components, including microprocessors, DSPs, memories, embedded software, etc. For some time, there was a need for system level languages to fill the gap between hardware description languages (HDLs) and traditional software programming languages. Such language should combine together hardware and software modeling and verification aspects. SystemC [13] was introduced as a new system level 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 (RTL) 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. This advance in technology and upgrade in languages capabilities had the drawback of increasing the complexity of the verification process. In fact, both the object-oriented (OO) aspect of SystemC and its hybrid software/hardware structure make it impossible to completely verify using pure simulation. In the same time, other formal verification approaches suffer from problems related to state-explosion (model checking) or complexity of the verification process (theorem provers).

In this work, we present a verification approach that takes advantage of both the SystemC programs (also called designs) structure and existing verification tools. For instance, most of the System-on-a-Chip (SoC) design methodologies consider that an SoC is composed of multiple Intellectual Properties (IPs). Assuming individual IPs have been verified, the verification process of the SoC properties are mostly related

to transactions (processes order of execution, triggering events, etc.) which can be represented efficiently as state machines (in particular Abstract State Machines: ASM [7]).

The ASM methodology is mathematically precise, yet general enough to be applicable to a wide variety of problem domains [7]. The ASM thesis asserts that any computing system can be described at its natural level of abstraction by an appropriate ASM. ASMs provide features to capture the behavioral semantics of programming and modeling languages, as a wide range of these languages were defined with this notion [8]. There are many languages that have been developed for ASMs, the recent one is AsmL [9] which was developed at Microsoft. We chose this language to model the SystemC simulator in order to execute the complete design. SystemC modeling in ASM has several advantages, in particular it is possible to interface this representation to existing formal verification tools such as model checkers (SMV [15] and MDG [5]), and theorem provers (PVS [6]). At the same time new tools, in particular AsmL tester [9], generates a finite state machine representation of the ASM model and can even output testing scenarios that are useful to perform better coverage of the system's possible states.

In order to make our verification methodology more efficient when dealing with SystemC designs we considered first to abstract the concrete SystemC design in order to extract a representation that only includes the process related information (execution and activation events). To do so, we used an approach based on abstract interpretation [1] that was first introduced by [14] for C++ programs. We added to the work of [14] the support of the SystemC simulator and library components. In addition to that we defined completely the SystemC simulator semantics and SystemC components in ASM while adapting definitions presented in [11]. This way, we will get a one-to-one mapping between the abstracted model and the ASM model. The complete formal semantics of the main parts of SystemC language presented in this paper is complementary to the work of Müller *et. al.*[11], where an ASM based SystemC semantics was introduced. We extend, however the definitions in [11] to cover more complex components of SystemC and introduce a new semantics definition for design rules of SystemC channels including static and dynamic design rules checking, and define the semantics of the SystemC simulator based on the definition of the SystemC scheduler introduced in [11].

The rest of the report is organized as follows: Section 2 overviews the related work. Sections 3 and 4 describe SystemC library and the abstract state machines, respectively. Section 5 describes our verification methodology. Section 6 presents a case study to illustrate our methodology. Section 7 concludes the paper with future work hints.

2 Related Work

Related projects to ours concern mainly defining SystemC semantics and verifying SystemC designs. For instance, Salem [12] presented the formal semantics of a synchronous subset of SystemC using denotational semantics. The delta cycle was formulated using function domains. Physical time was modeled at 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 also defined. The evaluate and update phases of SystemC scheduler have been formulated for both timed and immediate notifications. The work in [12], however, 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* or SystemC simulator.

Müller *et. al.* [10] first presented a semantics definition of SystemC 1.0 that covers method, thread, and clocked thread behavior. Later, Müller *et. al.* [11] defined parts of the semantics of SystemC 2.0, including 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.

The work in [11] nicely describes the above components using the theory of ASMs, however, there are still fundamental parts of SystemC 2.0 that were not defined, including some SystemC primitive and

hierarchical channels, design rules for SystemC channels, and most importantly the semantics of SystemC simulator. There are nontrivial components introduced in SystemC 2.0 such as mutual exclusive channels and request–grant protocol, which semantics should be defined in order to understand its behavior and how different components work and interact. Our work provides a comprehensive definition for the SystemC simulation semantics. It is different in the sense that we provide an abstract simulator which can be used as an underlying engine to execute abstract models of the designs. We also focus our definition towards solving the verification problem of SystemC by abstracting SystemC models and represent them in ASM and then generate a state machine out of the design. Probably this work is one of the very few ones that have executed the semantics defined in ASM on the supporting tools for ASM.

State machine representations were also used in verification of SystemC designs either when applying model checking or guiding functional verification. For instance, Drechsler *et al.* [2] proposed to use reachability analysis to verify certain properties of a SystemC design. Nevertheless, they restricted SystemC to a Verilog like language. Ferrandi *et al.* [3] proposed to use state machines to perform efficient functional verification of SystemC designs. They constructed an FSM directly from the code then use it to guide the test generation. In that work, the FSM generation was briefly described and does not consider the semantics of the SystemC simulator. Besides, we believe that, considering the complexity of the data structures of SystemC and its OO aspect, it is more suitable to represent SystemC designs as an ASM rather than a concrete FSM.

3 SystemC

SystemC [13] 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 specification to validate and verify system concepts. It is an SoC language representing functionality, communication, software and hardware at various system levels of abstraction.

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 specifies a set of access methods to be implemented within a channel, where channels provide the implementation for these interfaces. An event is a flexible synchronization primitive that is used to construct other forms of synchronization.

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 do not violate these rules during simulation time.

Most HDLs, 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 modeling primitive channels that cannot change instantaneously. By separating the two phases of evaluation and update, it is possible to guarantee deterministic behaviors.

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 notifications which occur. It supports both software and hardware-oriented modeling. The semantics of this scheduler was defined using ASM rules [11] and denotational semantics [12]. Understanding the scheduler is necessary to understand the simulation process.

In a layered approach, the SystemC simulation kernel forms the base layer. 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 refined communication channels. The next layer contains the definition 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[4].

4 Abstract State Machines

States in ASM are given as many–sorted first–order structures [7]. 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.

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. 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 *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 *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.

AsmL [9] is integrated with Microsoft's software development environment including Visual Studio, MS Word, and Component Object Model (COM), where it can be compiled and connected to the .NET framework. AsmL effectively supports specification and rapid prototyping of different kinds of models. The AsmL tester can also be used for FSM generation or test case generation.

5 SystemC Verification Methodology

Our goal is to represent a concrete SystemC design as an ASM model that can be used to verify some design properties, where we preserve the behavior of the original model with regard to the execution of the processes and the activation of the events. Figure 1 gives an overview of our methodology, including three steps as follows:

1. *Hypergraph Compiler*: In order to extract the information related to processes and their activation conditions we apply a static analysis technique based on abstract interpretation [1]. The output of this step is a graph, called *hypergraph* [14], that will include a representation of both the program's environment and the process's environment.

2. *ASM Model extractor*: From the previous hypergraph representation, we extract the list of processes present in the design and their activation events. we also associate with every process a list of events that will be activated when this process becomes active. The output is represented as an ASM model.

3. *AsmL Compiler*: The ASM model representing the system combined with the SystemC library semantics (also in ASM) are combined to form the final AsmL code. Once compiled, this latter can be used for model checking (SMV and MDG), theorem Proving (PVS) or used to guide tests generation (using the AsmL tester/compiler itself).

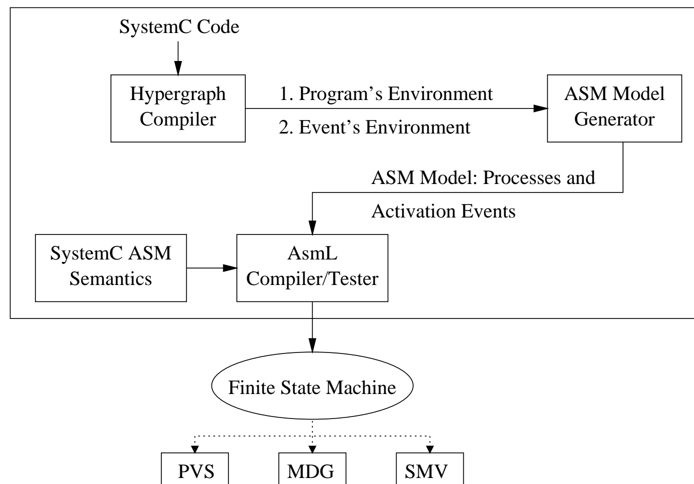

Figure 1: ASM-based SystemC verification methodology.

We defined an *abstract* model for the SystemC simulator along with the primary classes that are necessary to model SystemC designs and execute them. Then we are able to write our design within this AsmL and run it on the defined simulator.

5.1 SystemC Design Abstraction

In order to analyze statically SystemC designs and extract the required information to generate the ASM representation we considered an approach based on abstract interpretation [1]. Abstract interpretation is a formal technique that has proven to be efficient with object-oriented languages and large programs. The approach consists of:

1. Construct collecting semantics: define statically the future domains that will serve for the analysis and their specific manipulations.

2. Construct abstract semantics: map a property to a finite representation of the property more suitable for the analysis.

3. Define analysis techniques: analyze the abstract representation of the system in order to extract properties and/or to reduce the program size.

At the end of the analysis, the program will be represented as a hypergraph, which can be interpreted as a general automata connecting its states by branches (also called hyper-branches). Theses branches can be seen as an extension to Binary Decision Diagrams (BDDs) more adapted to programs representation. They offer a higher level of abstraction and flexibility by introducing the notion of confined hypergraph. This encapsulation property of the hypergraphs is very suitable to SoC where a system is a connection of modules using its input and output ports. This abstraction approach was first introduced by [14] to analyze C++ code. We augmented this work by the support for the SystemC simulator and specific classes which is mandatory to extract information related to SystemC processes and events from the design.

5.2 Semantics of SystemC

In this section we provide and ASM based semantics for a SystemC componenets including SystemC signal, MUTEX channel, a SystemC protocol, SystemC design rule checks, and SystemC simulator. The SystemC simulation kernel does not impose any order on processes that are simultaneously ready–to–run. In our definition, we treat different kinds of notifications separately. Immediate notifications are not shown in this definition since they are implied in the execution of a method, which we treat abstractly here. Timed and delta notifications are shown below within the simulator definition. The type of events notification can be immediate ($NONE$), at specific simulation time ($TIMED$), or at the next SystemC delta cycle ($DELTA$). The vocabulary of our semantics includes the set of timed events τ, the set of delta events λ, the SystemC zero time δ, and the current simulation time T_c.

We defined the semantics of the main functions of the following components: SystemC signal, SystemC FIFO, SystemC mutual exclusive (MUTEX) channel, message queue, request–grant protocol, FIFO with handshake protocol (Note that some functions were already defined in [11]). The description of SystemC components is based on the functional specifications of SystemC [4]. In following, we show the request–grant protocol semantics for illustration purposes.

5.2.1 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 [11]. 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.

$$
\begin{array}{l}
update() \equiv \\
\quad if\ CurrentValue \neq NewValue\ then \\
\quad\quad CurrentValue := NewValue \\
\quad\quad if\ ValueChangedEvent.Type\ =\ TIMED\ then \\
\quad\quad\quad \tau := \tau\ - \{ValueChangedEvent\} \\
\quad\quad endif \\
\quad\quad if\ ValueChangedEvent \notin\ \lambda\ then \\
\quad\quad\quad\quad \lambda := \lambda \cup \{ValueChangedEvent\} \\
\quad\quad endif
\end{array}
$$

$$time(ValueChangedEvent) := T_c + \delta$$
$$ValueChangedEvent.Type := DELTA$$
$$endif$$

5.2.2 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.

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

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

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

5.2.3 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 is then stored in an update variable ($projectedValue$), the projected signals request ($projectedSignals.REQUEST$) is enabled (which means that a master channel has already written to this port), and finally the channel is added to the channels update array for future updates.

$$
\begin{array}{l}
WriteMaster(data) \ \equiv \\
\qquad if\ currentSignals.REQUEST\ then \\
\qquad\qquad wait(noRequestEvent) \\
\qquad endif \\
\qquad projectedValue := data \\
\qquad projectedSignals.REQUEST := true \\
\qquad channelsUpdateArray := channelsUpdateArray\ \cup\ self
\end{array}
$$

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. Thereafter, the port reads the current value, projected signals grant and request are disabled, and finally the channel is scheduled for later updates. The semantics of the $readMaster$ is defined similarly to the $writeMaster$.

After updating the channels with new values, the *update* method notifies processes that are sensitive to request event ($requestEvent$) 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.

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

5.2.4 Design Rules

Each SystemC channel requires a specific number of ports to be connected to it, or arbitrarily unlimited in some cases. When a channel is created, it has to pass a 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 SystemC 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.

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

During dynamic design rule checking, the channel checks that there is only one driver process writing to the channel. If a process, is trying to write to the channel, then it should be verifi ed that there is no driver process accessing the channel, or it is the driver process that is trying to write to the channel. The MUTEX channel allows only one owner, and therefor, it has to pass *dynamic* design rules checking.

5.2.5 SystemC Simulator

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. The semantics of this scheduler was defi ned using ASM rules [11] and denotational semantics [12]. It is necessary to understand the scheduler in order to understand how the simulator works, however, it is not enough. The simulation process is affected by initialization process, processes execution and their order, events activation, and errors encountered during the simulation. Figure 2 shows a flow diagram of the SystemC simulation process. We defi ne the SystemC simulator semantics with ASM rules utilizing the scheduler defi nition presented in [11], where we highlight the steps that are already defi ned in [11] with dashed lines in the fi gure.

$$
\begin{aligned}
& Simulator \equiv \\
& \quad initialize \\
& \quad Schedule \\
& \quad CheckSimStatus \\
& \quad processTimedEvents \\
& \quad terminate
\end{aligned}
$$

The initialize step fi rst checks for a user stop request, then it sets to simulation end time ($SimulationEndTime$) according to the required simulation duration parameter. Thereafter, it prepares all methods processes and threads processes for simulation
($P.prepareForSimulation()$) by creating a coroutine package and allocating necessary stack space in memory. After that, we push all methods and threads into runnable methods and threads queue, respectively. Finally we process delta notifi cations by triggering all sensitive processes for delta events. The initialize semantics is given as where the $CheckSimStatus$ step just checks for a simulation error, or a user stop request.

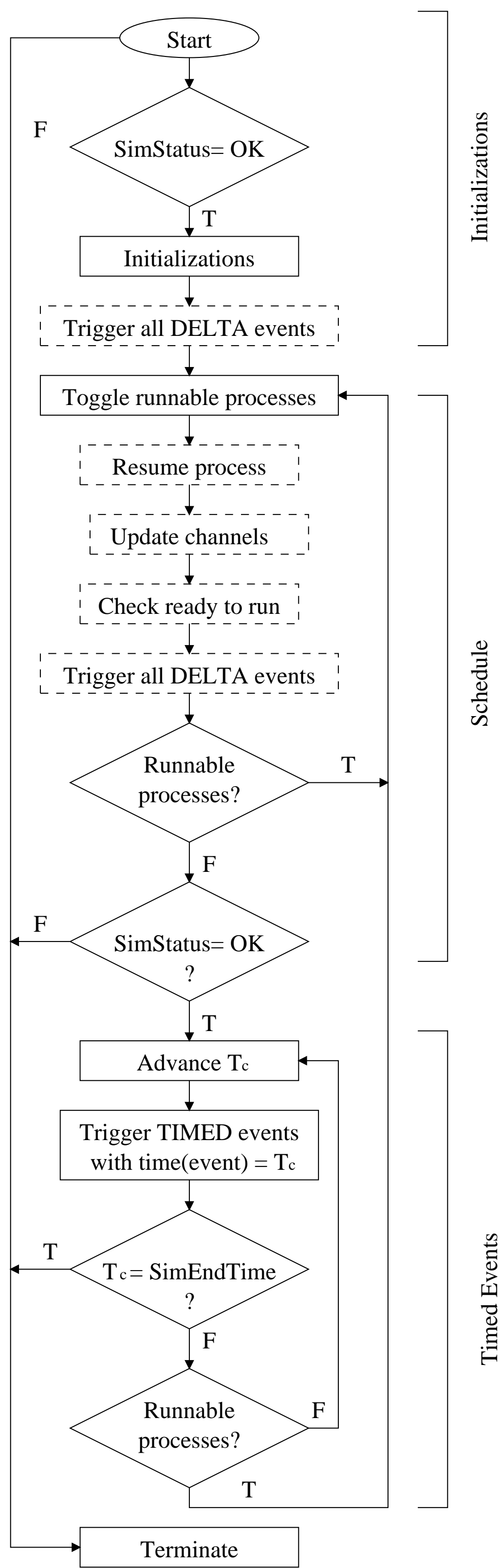

Figure 2: The SystemC Simulator

$$
\begin{array}{l}
Initialize \equiv \\
\quad if\ simStatus \neq SC_SIM_OK\ then \\
\quad\quad step := terminate \\
\quad endif \\
\quad SimulationEndTime := \\
\quad\quad if\ duration = \delta\ then\ MAX_TIME \\
\quad\quad else\ T_c + duration \\
\quad endif \\
\quad \forall P \in THREADS \cup CTHREADS :\ P.prepareForSimulation() \\
\quad updateChannels \\
\quad \forall M \in METHODS :\ RunnableProcessesTable.push(M) \\
\quad \forall Th \in THREADS :\ RunnableProcessesTable.push(Th) \\
\quad \forall e \in \lambda :\ e.trigger()
\end{array}
$$

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. All timed events are then processed. If there are more runnable processes and we did not exceed the simulation time, then we resume executing them by toggling the runnable processes table. However, if, after processing the timed events, there are no runnable processes, we advance the simulation time, and process timed events again.

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

$$
nextTime(\tau) \equiv time(event) :\ event \in \tau, \forall e \in \tau, time(event) \leq time(e)
$$

Therefore, the scheduler semantics, as defined in [11], does not distinguish between timed and delta notifications, so we just define the delta events check step ($ProcessDeltaEvents$) within the scheduler, as implemented in the source code, and move the $AdvanceTime$ step to define it within the timed events check above. Hence, to be consistent with the overall simulation process and to be able to define delta and timed notifications we would redefine the scheduler as follows (originally defined in [11] including a faithful definition of each step):

$$
\begin{array}{l}
Schedule \equiv \\
\quad\quad ResumeProcess \\
\quad\quad CheckReadyToRun \\
\quad\quad UpdateChannels \\
\quad\quad ProcessDeltaEvents
\end{array}
$$

The $ProcessDeltaEvents$ steps triggers all events in λ and checks if there are any more processes to execute. If this is the case, it toggles queues and goes to the $ResumeProcess$ step, otherwise, it goes back to the $CheckSimStatus$ step before processing timed notifications.

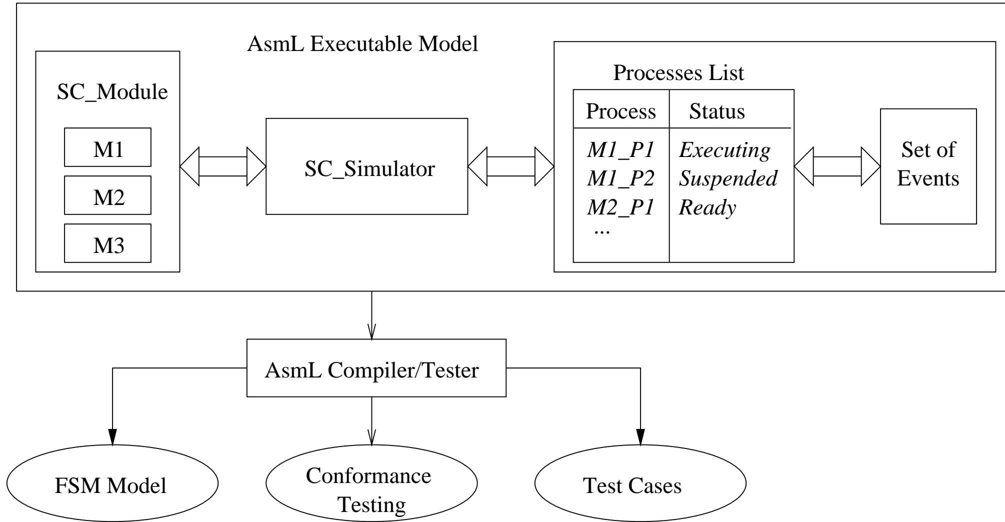

Figure 3: Executing an abstract AsmL model for the semantics of SystemC simulator

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

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 3 shows the general framework for semantics execution.

6 Case Study: Bus Structure Model

To illustrate the proposed SystemC verification methodology , we consider in this section a simple bus structure model [13].

6.1 Bus Description

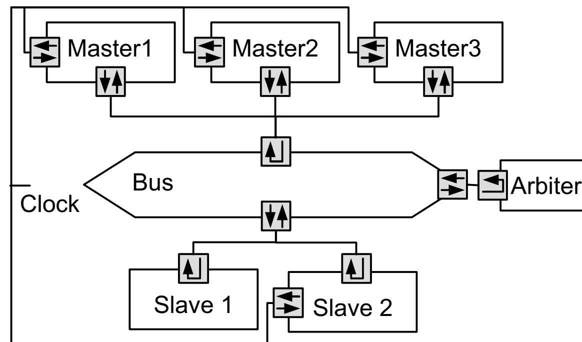

Figure 4: Simple bus structure.

Figure 4 shows the bus structure. It uses an overall form of synchronization, where modules attached to the bus execute on the rising clock edge, and the bus itself executes on a falling clock edge. Multiple masters can be connected to the bus. Each master is identified by a unique priority, that is represented by an unsigned integer number. The lower this priority number is, the more important the master is. Each master communicates with the bus via an interface, which describes the communication between masters and the bus; Three modes are possible: (1) *Blocking Mode* where data is moved through the bus in a burst–mode. Hence, the transaction cannot be interrupted by a request with a higher priority. (2) *Non-Blocking Mode* where the master read or write a single data word. After the transaction is completed, the caller must take care of checking the status of the last request, which can be issued and placed on the queue (BUS_REQUEST), served but is not completed (BUS_WAIT), completed without errors (BUS_OK), or finally did not complete due to an error (BUS_ERROR). (3) *Direct Mode*, where the direct interface functions perform the data transfer through the bus, but without using the bus protocol. They are usually used to debug the state of the memory.

The slave interface describes the communication between the bus and the slaves. Multiple slaves can be connected to the bus. Each slave models some kind of memory that can be accessed through the slave interface. Two modes are possible: (1) *Direct interface* where it can perform immediate read or writing of data without using the bus protocol. (2) *Indirect interface* where the slave can read or write a single data element. The functions return instantaneously and the caller must check the status of the transfer.

The arbiter is responsible for choosing the appropriate master when there is more than one connected to the bus. The arbiter performs the selection according to the following rules: (1) if the current request is a locked burst request, then it is always selected, (2) if the last request had its lock flag set and is again "requested", then it is selected from the collection queue and returned, otherwise (3) the request with the highest priority is selected from the collection queue and returned.

This structure includes several SystemC components and nicely makes use of the principles of using SystemC at the transactional level. Besides some of the sample properties, e.g. liveness and safety, cannot be verified using simulation. They require the usage of formal techniques such as model checking.

6.2 State Machine Generation

A partial representation of the bas hypergraph is given in Figure 5. It shows the first hypergraph generated from the bus code. It includes an events' environment containing several processes: masters, slaves, clocks, arbiter, etc. In parallel with the program environment, the events environment includes the list of all the system processes and their status.

The simulation manager is presented as a box connected to the entries of the program hypergraph. It can be seen as a procedure that determines the structure of the system according to the list of active processes. For example, if the Master 1 is active, then, only its correspondent code is analyzed. Each small box from the program environment, (e.g., arbiter()) presents a confined hypergraph that includes the correspondent object members and methods.

After applying reductions tactics on the hypergraph of Figure 4, the generated reduced hypergraph is concretized into an ASM model. Using the AsmL Tester, this model is executed on top of the abstract simulator we defined in section 5. In addition to the possibility of interfacing the ASM code to several model checkers and theorem provers, this AsmL Tester offers a variety of options that can serve to generate an FSM representation of the design. Figure 6 presents a simplified FSM including only 4 clock steps and taking into consideration the macro events corresponding to the masters sending and slaves receiving processes.

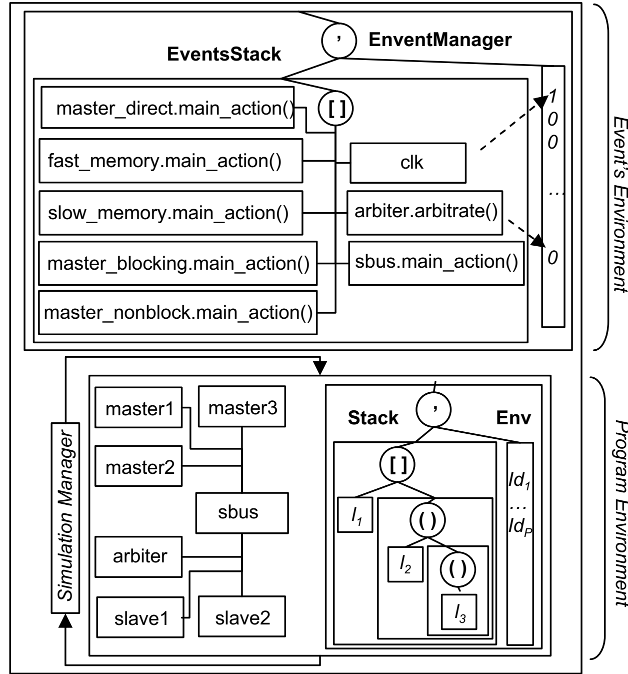

Figure 5: Hypergraph of the simple bus structure.

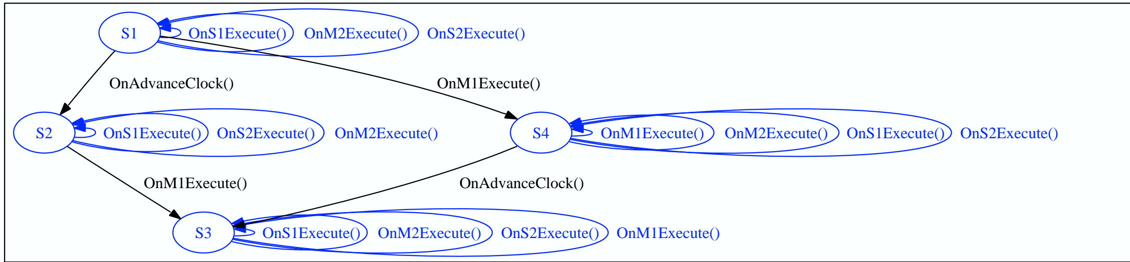

Figure 6: Generated state machine for the bus structure.

7 Conclusion

In this work we presented a methodology to verify SystemC designs based on a definition for SystemC semantics using Abstract State Machines (ASMs). First we defined SystemC nontrivial components, design rules checking, and SystemC simulator using Abstract State Machines transition rules. Then we used these semantics to construct an abstract SystemC simulator in AsmL. Using this simulator we can define and execute an abstract model of the source SystemC design. The abstraction methodology is based on abstract interpretation, where we represent our design as a hypergraph. Hypergraphs capture an abstract model based on SystemC processes and events for the source program. This abstract model can be executed on top of our abstract simulator, where it is possible to generate test cases or finite state machines for the model. Conformance testing can also be conducted using the AsmL Tester tool. In summary, the proposed approach will translate the verification problem of SystemC designs from the C++ level code into a higher level of abstraction, namely ASM level.

We applied our approach on the a case study of a bus structure model. We considered a SystemC model of the bus structure and generated its hypergraph, then extracted an events–based model, which is implemented in AsmL language and executed on top of the simulator. We generated an FSM of the model

taking into consideration the macro events corresponding to the sending and receiving processes. We believe that our work achieved two goals: generating events–based FSM of complex SystemC designs and reduce the learning time and effort for understanding SystemC by providing an abstract executable simulator.

As a future work, we intend to test the approach with a larger case study and apply verifi cation techniques such as assertion based verifi cation, property checking, or test cases generating. We also need to explore the possibility of applying this approach on other SoC modeling languages like SuperVerilog.

References

[1] P. Cousot and R. Cousot. Abstract interpretation frameworks. Journal of Logic and Computation, 2(4):511-547, August 1992.

[2] R. Drechsler and D. Groe. Reachability analysis for formal verification of SystemC, In Euromicro Symposium on Digital System Design (DSD), Dortmund, Germany. PP 337–340. 2002.

[3] F. Ferrandi, M. Rendine and D. Sciuto. Functional Verification for SystemC Descriptions Using Constraint Solving, Design, Automation and Test in Europe Conference and Exhibition (DATE). Paris, France. PP 744–751. 2002.

[4] Functional Specification for SystemC 2.0.1. Synopsys Inc. 2003.

[5] A. Gawanmeh, S. Tahar, K. Winter, Formal Verification of ASM Designs using the MDG Tool. In A. Cerone, P. Lindsay, editors, Int. Conf. on Software Engineering and Formal Methods (SEFM 2003), IEEE Computer Society, PP 210–219. 2003.

[6] A. Gargantini and E. Riccobene, Encoding Abstract State Machines in PVS. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele (eds.), International Workshop on Abstract State Machines, Monte Verita, Switzerland, Local Proceedings, PP 152–173. March 2000.

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

[8] J. Huggins. Abstract State Machines website. http://www.eecs.umich.edu/gasm/. 2003.

[9] 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. 2003.

[10] W. M¨uller, 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), Munich, Germany. pp. 64–70. 2001.

[11] W. M¨uller, J. Ruf, and W. Rosenstiel. SystemC Methodologies and Applications. Kluwer Academic Pub. 2003.

[12] A. Salem. Formal Semantics of Synchronous SystemC.Proc. Design, Automation and Test in Europe Conference and Exposition (DATE), Munich, Germany. pp. 10376–10381. 2003.

[13] SystemC website. http://www.systemc.org, 2004.

[14] F. Vederine. Analyses totales de programmes par interpretation abstraite. PhD thesis, Ecole Polytechnique, Paris, France, 2000.

[15] K. Winter. Model Checking Abstract State Machines, Ph.D. thesis, Tech. University of Berlin, Germany, 2001.