SystemC Semantics in Fixpoint

Ali Habibi and Sofiène Tahar

Department of Electrical and Computer Engineering,

Concordia University, Montreal, Canada

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

Technical Report

December 2004

Abstract

SystemC is among a group of system level design languages proposed to raise the abstraction level for embedded system design and verification. Defining the formal semantics of SystemC is an important and mandatory step towards enabling the formal verification of SystemC. In this technical report, we present a sound and complete semantics of the main parts of SystemC in fixpoint. Such a semantics are a precise and non ambiguous definitions of the SystemC library. They are very useful to guarantee a unique implementation of the library and interpretation of its behaviour. Besides, they can used in conducting formal proofs for sound abstractions or even to construct syntactical transformers to other languages.

Contents

1 Introduction

SystemC [5] is an object-oriented system level language for embedded systems design and verification. It is expected to make a stronger effect in the area of architecture, co-design and integration of hardware and software. The SystemC library is composed from a set of classes and a simulation kernel extending C++ to enable the modeling of complex systems at a higher level of abstraction than state-of-the-art HDL (Hardware Description Languages). However, except for small models, the verification of SystemC designs is a serious bottleneck in the system design flow. Direct model checking of SystemC designs is not feasible due to the complexity of the SystemC library and its simulator.

In this report, we provide a formalization of the SystemC semantics in fixpoint. For instace, we will first enumerate the syntactical domains. Then, we will provide the semantics of whole program and the semantics of SystemC modules. Finally, we will provide the proofs for the completeness and soundness of the whole semantics. Our approach updates on the generic semantics provided by Logozzo in [2] by: (1) modifying the syntactical domains to support SystemC specific domains such as Modules and Signals; (2) upgrading the default environment and store to include the values of the signals in the previous, current and next simulation cycles; (3) adding to the generic class semantics (in particular to the class constructor) the information related to initializing the processes and threads involved in the simulation; and (4) re-establishing the soundness and completeness proofs considering the updates and modifications of points (1), (2) and (3).

Related work to ours concerns mainly writing SystemC semantics. For instance, several approaches have been used to write the SystemC semantics (e.g., using ASM is [3]). Denotational semantics [4] is found to be most effective since objects can be expressed as fixpoints on suitable domains. Salem in [6] proposed a denotational semantics for SystemC. However, the proposal in [6] was very shallow and does not relate the semantics of the whole SystemC program to the semantics of its classes. Besides, in all the previous works there was no proofs of soundness of the presented semantics.

The rest of this report is organized as follows: Section 2 presents the main SystemC syntactical domains. Section 3 presents the SystemC semantics in fixpoint. Finally, Section 4 concludes the report.

2 Syntactical Domains

SystemC have a large number of syntactical domains. However, they are all based on the single `SC_Module` domain. Hence, the minimum representation for a general SystemC program is as a set of modules.

Definition 2.1 *(SystemC Module:* `SC_Module`*)*
A SystemC Module is a set $\langle$`DMem`, `Ports`, `Chan`, `Mth`, `SC_Ctr`$\rangle$*, where* `DMem` *is a set of the module data members,* `Ports` *is a set of ports,* `Chan` *a set of SystemC Chan,* `Mth` *is a set of methods (functions) definition and* `SC_Ctr` *the module constructor.*

Definition 2.2 *(SystemC Port:* `SC_Port`*)*
A SystemC Port is a set $\langle$`IF`, `N`, `SC_In`, `SC_Out`, `SC_InOut`$\rangle$*, where* `IF` *is a set of the virtual methods declarations,* `N` *is the number of interfaces that may be connected to the port,* `SC_In` *is an input port (provides only a* `Read` *method),* `SC_Out` *is an output port (provides only a* `Write` *method) and* `SC_InOut` *is an input/output port (provides* `Read` *and* `Write` *functions).*

Definition 2.3 *(SystemC Channel:* `SC_Chan`*)*
A SystemC Channel is a set $\langle$`SigMeth`, `CurrVal`, `PrevVal`, `NewVal`, `SC_Mutex`, `SC_Semaph`$\rangle$*, where* `SigMeth` *is a set of basic channel virtual methods (including in particular the* `Update` *method),* `CurrVal` *the current value of the signal,* `PrevVal` *its previous value,* `NewVal` *its value in the next simulation cycle,* `Mutex` *is a mutex channel (including additional methods such as* `Lock` *and* `UnLock`*) and* `SC_Semaph` *is a semaphore interface (including in particular the number of concurrent accesses to the interface).*

In contrast to default class constructors for OO languages, the SystemC module constructor `SC_Ctr` contains the information about the processes and threads that will be executed during simulation, and their sensitivity lists, `SC_SL`, specifying which events can affect their states.

Definition 2.4 *(SystemC Constructor:* `SC_Ctr`*)*
A SystemC Constructor is a set $\langle$`Name`, `Init`, `SC_Pr`, `SC_SSt`$\rangle$*, where* `Name` *is a string specifying the module name,* `Init` *is a default class constructor,* `SC_Pr` *a set of processes and* `SC_SSt` *is a set of sensitivity statements (to set the process sensitivity list* `SC_SL`*).*

Definition 2.5 *(SystemC Process: $\texttt{SC\_Pr}$)*
A SystemC process is a set $\langle \texttt{PMth}, \texttt{PTh}, \texttt{PCTh} \rangle$*, where* $\texttt{PMth}$ *is a method process (defined as a set* $\langle \texttt{Mth}, \texttt{SC\_SL} \rangle$ *including the method and its sensitivity list),* $\texttt{PTh}$ *is a thread process (accepts a wait statement in comparison to the method process),* $\texttt{PTh}$ *is a clocked thread process (sensitive to the clock event).*

Definition 2.6 *(SystemC Process Sensitivity List: $\texttt{SC\_SL}$)*
A SystemC sensitivity list is a set $\langle \texttt{SL}_S, \texttt{SL}_D \rangle$*, where* $\texttt{SL}_S$ *is a static sensitivity list and* $\texttt{SL}_D$ *is a dynamic list. Both lists contain a set of events* $\texttt{SC\_Event}$ *but are different in the sense that one can be updated during the simulation while the other is not changeable.*

Definition 2.7 *(SystemC Event: $\texttt{SC\_Event}$)*
A SystemC event is a set $\langle \texttt{t}, \texttt{notify}, \texttt{cancel} \rangle$*, where* $\texttt{t}$ *specifies (in simulation cycles) when the notification is supposed to be sent,* $\texttt{notify}$ *the method used to notify the owner module and* $\texttt{cancel}$ *is the method used to cancel an event.*

Definition 2.8 *(SystemC Program: $\texttt{SC\_Pg}$)*
A SystemC program is a set $\langle \texttt{L}_{\texttt{SC\_Mod}}, \texttt{SC\_main} \rangle$*, where* $\texttt{L}_{\texttt{SC\_Mod}}$ *is a set of SystemC modules and* $\texttt{SC\_main}$ *is the main function in the program that performs the simulator initialization and contains the modules declarations.*

Note that restricting our model to modules does not affect the validity of the results since modules are the default syntactical domain for SystemC. All other domains are built on top of it.

3 Fixpoint Semantics

3.1 Semantic Domains

In this section, we define the semantics of the whole SystemC program, $\mathbb{W}\,[\![\texttt{SC\_Pg}]\!]$, and the SystemC module, $\mathbb{M}_{SC}[\![\texttt{m\_sc}]\!]$. Then, present the proofs (or proof sketches) of the soundness and completeness of $\mathbb{M}_{SC}[\![\texttt{m\_sc}]\!]$.

Definition 3.1 *(Delta Delay: δ_d)*
The SystemC simulator considers two phases evaluate *and* update. *The separation between these two phases is called* delta delay.

Definition 3.2 *(SystemC Environment: $\mathtt{SC\_Env}$)*
The SystemC environment is the summation of the default C++ environment ($\mathtt{Env}$) as defined in [2] and the signal environment ($\mathtt{Sig\_Store}$) specific to SystemC: $\mathtt{SC\_Env} = \mathtt{Env} + \mathtt{Sig\_Env} = [\mathtt{Var} \to \mathtt{Addr}] + [\mathtt{SC\_Sig} \to \mathtt{Addr},\mathtt{Addr}]$, where $\mathtt{Var}$ is a set of variables, $\mathtt{SC\_Sig}$ is a set of SystemC signals and $\mathtt{Addr} \subseteq \mathbb{N}$ is a set of addresses.

Definition 3.3 *(SystemC Store: $\mathtt{SC\_Store}$)*
The SystemC store is the summation of the default C++ store ($\mathtt{Store}$) as defined in [2] and the signal store ($\mathtt{Sig\_Store}$): $\mathtt{SC\_Store} = \mathtt{Store} + \mathtt{Sig\_Store} = [\mathtt{Addr} \to \mathtt{Val}] + [(\mathtt{Addr}, \mathtt{Addr}) \to (\mathtt{Val},\mathtt{Val})]$, where $\mathtt{Val}$ is a set of values such that $\mathtt{SC\_Env} \subseteq \mathtt{Val}$.

Let $\mathsf{R}_0 \in \mathcal{P}(\mathtt{SC\_Env}\times\mathtt{SC\_Store})$ be a set of initial states, $\mathtt{pc}_{\mathtt{in}}$ be the entry point of the main function $\mathtt{sc\_main}$ and $\to\subseteq$: $(\mathtt{SC\_Env} \times \mathtt{SC\_Store}) \times (\mathtt{SC\_Env}\times\mathtt{SC\_Store})$ be a transition relation.

3.2 Whole SystemC Program Semantics

The whole SystemC program semantics can be defined as the traces of the executions of the program starting from a set of initial states R_0. It can be expressed in fixpoint semantics as follows:

Definition 3.4 *(Whole SystemC Program Semantics: $\mathbb{W}[\![\mathtt{SC\_Pg}]\!]$)*
Let $\mathtt{SC\_Pg} = \langle \mathsf{L}_{\mathtt{SC\_Mod}}, \mathtt{SC\_main}\rangle$ be a SystemC program. Then, the semantics of $\mathtt{SC\_Pg}$, $\mathbb{W}[\![\mathtt{SC\_Pg}]\!] \in \mathcal{P}(\mathtt{SC\_Env}\times\mathtt{SC\_Store}) \to \mathcal{P}(\mathcal{T}(\mathtt{SC\_Env}\times \mathtt{SC\_Store}))$ is

$$\mathbb{W}[\![\mathtt{SC\_Pg}]\!](R_0) = \mathit{lfp}\,^{\subseteq}_{\emptyset}\lambda X. \quad (R_0) \cup \{\rho_0 \to \ \ldots \ \rho_n \to \rho_{n+1} | \ \rho_{n+1} \in (\mathtt{SC\_Env}\times \mathtt{SC\_Store}) \wedge \{\rho_0 \to \ \ldots \ \rho_n\} \in X \wedge \rho_n \to \rho_{n+1}\}$$

3.3 Module Semantics

A SystemC module is a particular C++ class where the constructor declares a set of processes and thread that will executed during simulation according to a set of events (timed or non-timed). The module semantics can be defined as the set of all its instances. While and object module semantics reflects the evolution of the object internal state.

3.3.1 Module Constructor Semantics

Definition 3.5 *(Process Declaration: $\mathbb{P}_R$ $[\![\texttt{SC\_Pr}]\!]$)*
Let $\texttt{SC\_Pr} = \langle \texttt{PMth}, \texttt{PTh}, \texttt{PCTh} \rangle$ *be a SystemC process. Then, the semantics of* $\mathbb{P}_R$ $[\![\texttt{SC\_Pr}]\!]) \in \mathcal{P}(\texttt{SC\_Env} \times \texttt{SC\_Store}) \to \mathcal{P}(\mathcal{T}(\texttt{SC\_Env} \times \texttt{SC\_Store}))$ *is*

$$\mathbb{P}_R\, [\![\texttt{SC\_Pr}]\!](R_0, M, SL) = \\ lfp^{\subseteq}_{\emptyset}\, \lambda X,\ m,\ sl.\ (R_0) \cup \{\rho_0 \to\ \ldots\ \rho_n \to \rho_{n+1} \mid \rho_{n+1} \in (\texttt{SC\_Env} \times \texttt{SC\_Store}) \wedge \{\rho_0 \to\ \ldots\ \rho_n\} \in X \wedge \rho_n \to \rho_{n+1} \wedge \rho_{n+1}(X) = (m, sl)\}$$

Definition 3.6 *(SystemC Module Constructor:* $\mathbb{P}_{Ctr}$ $[\![\texttt{SC\_Ctr}]\!]$*))*
Let $\texttt{SC\_Ctr} = \langle \texttt{Name}, \texttt{Init}, \texttt{SC\_Pr}, \texttt{SC\_SSt} \rangle$ *be a constructor of a SystemC module. Then, the semantics of* $\mathbb{P}_{Ctr}$ $[\![\texttt{SC\_Ctr}]\!] \in \mathcal{P}(\texttt{SC\_Env} \times \texttt{SC\_Store}) \to \mathcal{P}(\mathcal{T}(\texttt{SC\_Env} \times \texttt{SC\_Store}))$ *is*

$$\mathbb{P}_{Ctr}[\![\texttt{SC\_Ctr}]\!](L_{P,M,SL}) = lfp^{\subseteq}_{\emptyset}\quad \lambda\, \{(p_1, m_1, sl_1), \ldots, (p_i, m_i, sl_i), \ldots, (p_n, m_n, sl_n)\}. \\ \cup_{(p_i, m_i, sl_i) \in L_{p,m,sl}}\, \mathbb{P}_R\, [\![\texttt{SC\_Pr}]\!](R_0, M, SL)\}$$

3.3.2 Module Object Semantics

In a general OO context, such as C++, an object can be defined a set of states including a first (initial) state representing the object just after its creation and a set of states resulting from the interaction of the object with its context [2]. In this case, the interaction can happen in two ways: (1) the context invokes an object's method, or (2) the context modifies a memory location reachable from the object's environment. In [2], this interaction was very well defined using two functions $\texttt{next}_{\texttt{d}}$, for direct interactions, and $\texttt{next}_{\texttt{ind}}$ for indirect interactions and the object semantics, $\mathbb{O}\,[\![\texttt{o}]\!]$, was defined as:

$$\mathbb{O}[\![\texttt{o}]\!](\texttt{v},\texttt{s}) = \mathrm{lfp}^{\subseteq}_{\emptyset}\quad \lambda T.\ \texttt{S}_{\texttt{0}}\langle v, s\rangle \cup \{\sigma_0 \xrightarrow{l_0} \ldots \xrightarrow{l_{n-1}} \sigma_n \xrightarrow{l_n} \sigma' \mid \\ \{\sigma_0 \xrightarrow{l_0} \ldots \xrightarrow{l_{n-1}} \sigma_n \in T, \texttt{next}(\sigma_n)\ \ni\ \langle \sigma', l' \rangle\}$$

where next(σ) = $\texttt{next}_{\texttt{d}}(\sigma) \cup \texttt{next}_{\texttt{ind}}(\sigma)$, l is a transition label and $\texttt{S}_{\texttt{0}}\langle v, s\rangle$ is a set of initial states.

In addition to the semantics definition of an OO object in [2], a SystemC method can be activated by the SystemC simulator through the sensitivity list of the process. This interaction is a hybrid direct/inderect interaction because the SystemC simulator will, according the state of the program events (that maybe external to the module), invoke directly the concerned methods. First, we will define the interaction states, then, we will provide the complete definition for the direct, indirect and SystemC simulator based interaction functions.

Definition 3.7 *(Interaction States)*
The set of interaction states is $\sum = \mathtt{SC\_Env} \times \mathtt{SC\_Store} \times \mathtt{D}_{out} \times \mathcal{P}(\mathtt{Addr})$

After the creation of the module object, the reached states represent the initial states defined as follows:

Definition 3.8 *(Initial States* $\mathtt{S}_0\langle v_{sc}, s_{sc}\rangle$*)*
Let $v_{sc} \in \mathtt{D}_{in}$ *be a SystemC object input value,* $s_{sc} \in \mathtt{SC\_Store}$ *a store at object creation time and* $\mathtt{SC\_Obj}$ *a SystemC module object. The set of initial states of* $\mathtt{SC\_Obj}$ *is:*

$$\mathtt{S}_0\langle v_{sc}, s_{sc}\rangle = \{\langle e'_{sc}, s'_{sc}, \phi, \emptyset\rangle \mid \mathbb{P}_{Ctr}[\![\mathtt{SC\_Ctr}]\!](\mathtt{L}_{P,M,SL}) \ni \langle e'_{sc}, s'_{sc}\rangle\}$$

where: $\mathtt{L}_{P,M,SL} = \{(p_1, m_1, sl_1), \ldots, (p_i, m_i, sl_i), \ldots, (p_n, m_n, sl_n)\}$ *is a list of the all the module processes, methods, and sensitivity lists.*

In the previous definition ϕ is a void value ($\in \mathtt{D}_{\mathtt{out}}$) meaning that the constructor does not return any value and therefore does not expose any address to the context.

Definition 3.9 *(Transition Labels:* $\mathtt{Label\_SC}$*)*
The set of transition labels is $\mathtt{Label\_SC} = (\mathtt{Mth} \times \mathtt{D}_{in}) \cup (\mathtt{SC\_Pr} \times \mathtt{D}_{in}) \cup \{k\}$.

In previous definition we distinguish three type of interactions corresponding respectively to: (1) invoking a C++ method (direct interaction); (2) invoking a SystemC process (interaction through the SystemC simulator); and (3) modifying the memory location that is reachable from the the object environment (indirect interaction). The transition function $\mathtt{next\_SC}$ is made up of three functions: $\mathtt{next\_SC}_{\mathtt{dir}}$, $\mathtt{next\_SC}_{\mathtt{pr}}$ and $\mathtt{next\_SC}_{\mathtt{ind}}$.

Definition 3.10 *(Direct interactions:* $\mathtt{next\_SC}_{dir}$*)*
Let $\langle e_{sc}, s_{sc}, v_{sc}, \mathtt{Esc}\rangle \in \sum$ *an interaction state. Then, the direct interaction function* $\mathtt{next\_SC}_{dir} \in [\sum \to \mathcal{P}(\sum \times \mathtt{Label\_SC})]$ *is defined as:*

$$\begin{aligned}&\mathtt{next\_SC}_{dir}(\langle e_{sc}, s_{sc}, v_{sc}, \mathtt{Esc}\rangle) = \\ &\quad \{\langle\langle e'_{sc}, s'_{sc}, v'_{sc}, \mathtt{Esc'}\rangle, \langle mth, v_{in}\rangle\rangle \mid mth \in \mathtt{Mth}, v_{in} \in \mathtt{Din}, \\ &\quad \mathbb{M}[\![\mathtt{mth}]\!](v_{in}, e_{sc}, s_{sc}) \ni (v'_{sc}, e'_{sc}, s'_{sc}), \mathtt{Esc'} = \mathtt{Esc} \cup \text{reachable}(v'_{sc}, e'_{sc})\}.\end{aligned}$$

where $\mathbb{M}[\![\mathtt{mth}]\!]$ *is the semantics of generic OO method as defined in [2].*

The function *reachable* is updated helper function of the one defined in [2]. For instance, given an address v_{sc} and a store s_{sc}, *reachable* determines all the addresses that are reachable from v_{sc}. In the SystemC context, this function acts only on the data members of the module according to the following recursive definition:

Definition 3.11 *(The function* reachable*)*
The function reachable $\in [\mathtt{D}_{out} \times \mathtt{SC\_Store}] \to \mathcal{P}(\mathtt{Addr})$ *is defined as follows:*
reachable(v_{sc},s_{sc}) =
if $v_{sc} \in \mathtt{Addr}$ *then*
$\{\mathtt{Addr}\} \cup \{$reachable($e'_{sc}(d_{mem})$,$s'_{sc}$) $|$
$\exists$ $\mathtt{sc\_module}$ =
$\langle \mathtt{DMem}, \mathtt{Ports}, \mathtt{Chan}, \mathtt{Mth}, \mathtt{SC\_Ctr}\rangle$,
$d_{mem} \in \mathtt{DMem}$, $s_{sc}(v_{sc})$
is an instance of $\mathtt{sc\_module}$, $s_{sc}(s_{sc}(v_{sc}))$ = $e'_{sc}\}$
else $\emptyset$.

In the case of interactions related to changing the sensitivity list of a processor, the function $\mathtt{next\_SC_{pr}}$ considers the method that was affected to the process in the module constructor. Then, the invocation of the method is similar to the direct interaction.

Definition 3.12 *(Process interactions:* $\mathtt{next\_SC}_{pr}$*)*
Let $\langle e_{sc}, s_{sc}, v_{sc}, \mathtt{Esc}\rangle \in \sum$ *an interaction state. Then, the process interaction function* $\mathtt{next\_SC}_{pr} \in [\sum \to \mathcal{P}(\sum \times \mathtt{Label\_SC})]$ *is defined as:*
$\mathtt{next\_SC}_{pr}(\langle e_{sc}, s_{sc}, v_{sc}, \mathtt{Esc}\rangle)$ =
$\{\langle\langle e''_{sc}, s''_{sc}, v''_{sc}, \mathtt{Esc}''\rangle, \langle pr, m, sl, v_{in}\rangle\rangle \mid pr \in \mathtt{SC\_Pr}$, $v_{in} \in \mathtt{Din}$,
$\mathbb{P}_{Ctr}[\![\mathtt{SC\_Ctr}]\!](pr, m, sl) \ni (v'_{sc}, e'_{sc}, s'_{sc})$,
$\mathbb{M}[\![\mathtt{m}]\!](v_{in}, e'_{sc}, s'_{sc}) \ni (v''_{sc}, e''_{sc}, s''_{sc})$,
$\mathtt{Esc}''$ = $\mathtt{Esc} \cup$ reachable(v''_{sc}, e''_{sc})$\}$.

The third possible interaction corresponds to indirect interaction which may happen when an address escapes from an object. In that case, the context can modify the content of this address with any value. The function $\mathtt{next\_SC_{ind}}$ defines this type of interaction:

Definition 3.13 *(Indirect interactions:* $\mathtt{next\_SC}_{ind}$*)*
Let $\langle e_{sc}, s_{sc}, v_{sc}, \mathtt{Esc}\rangle \in \sum$ *an interaction state. Then, the indirect interaction function* $\mathtt{next\_SC}_{ind} \in [\sum \to \mathcal{P}(\sum \times \mathtt{Label\_SC})]$ *is defined as:*
$\mathtt{next\_SC}_{ind}(\langle e_{sc}, s_{sc}, v_{sc}, \mathtt{Esc}\rangle)$ =
$\{\langle\langle e_{sc}, s'_{sc}, \phi, \mathtt{Esc}\rangle, k\rangle \mid \exists\, \alpha \in \mathtt{Esc}.\ s'_{sc} \in$ update_sc(α, s'_{sc})$\}$.

The *update_sc* function is an extension of the *update* function defined in [2] in the sense that it considers SystemC signals in addition to C++ variables. It is defined as follows:

Definition 3.14 *(The function* update_sc*)*
The function update_sc $\in [\texttt{Addr} \times \texttt{SC\_Store} \rightarrow \mathcal{P}(\texttt{SC\_Store})]$ *is defined as follows:*

update_sc(α, s_{sc}) = $\{s'_{sc} \mid \exists\, v \in \texttt{Val}.\ s'_{sc} = s_{sc}[\alpha \mapsto v]\ \}$.

update_sc returns all the possible stores where $s_{sc}(\alpha)$ takes all the possible values in values domain Val.

Using the definitions of $\texttt{next\_SC}_{\texttt{dir}}$, $\texttt{next\_SC}_{\texttt{pr}}$ and $\texttt{next\_SC}_{\texttt{ind}}$, we define the global transition function next_SC as:

Definition 3.15 *(Transition function:* next_SC*)*
Let $\texttt{st} = \langle e_{sc}, s_{sc}, v_{sc}, \texttt{Esc}\rangle \in \sum$ *be an interaction state. Then, the transition function* next_SC $\in [\sum \rightarrow \mathcal{P}(\sum \times \texttt{Label\_SC})]$ *is defined as:*

$\texttt{next\_SC(st)} = \texttt{next\_SC}_{dir}\texttt{(st)} \cup \texttt{next\_SC}_{pr}\texttt{(st)} \cup \texttt{next\_SC}_{ind}\texttt{(st)}$

Using the transition function, a SystemC module object semantics is defined as follows:

Definition 3.16 *(SystemC Module Object:* $\mathbb{O}_{SC}[\![\texttt{o\_sc}]\!]$*)*
Let $v_{sc} \in \texttt{Val}$ *be a SystemC object input value and* $s_{sc} \in \texttt{SC\_Store}$ *a store at object creation time. Then the SystemC object semantics,* $\mathbb{O}_{SC}[\![\texttt{o\_sc}]\!] \in [D_{in} \times \texttt{Store}] \rightarrow \mathcal{P}(\mathcal{T}(\Sigma))$ *is defined as:*

$$\mathbb{O}_{SC}[\![\texttt{o\_sc}]\!](v_{sc}, s_{sc}) = \quad lfp^{\subseteq}_{\emptyset}\ \lambda T.\ S_0\langle v, s\rangle \cup \{\sigma_0 \xrightarrow{l_0} \ldots \xrightarrow{l_{n-1}} \sigma_n \xrightarrow{l_n} \sigma' \mid \{\sigma_0 \xrightarrow{l_0} \ldots \xrightarrow{l_{n-1}} \sigma_n \in T,\ \texttt{next\_SC}(\sigma_n) \ni \langle \sigma', l'\rangle\}\}$$

where $\sum = \texttt{SC\_Env} \times \texttt{SC\_Store} \times D_{out} \times \mathcal{P}(\texttt{Addr})$ *is a set of interaction states,* D_{in} *and* D_{out} *are respectively the semantic domains for the input and output values.*

Theorem 3.1 *Let*

$$F_{sc} = \lambda T.\quad S_0\langle v, s\rangle \cup \{\sigma_0 \xrightarrow{l_0} \ldots \xrightarrow{l_{n-1}} \sigma_n \xrightarrow{l_n} \sigma' \mid \{\sigma_0 \xrightarrow{l_0} \ldots \xrightarrow{l_{n-1}} \sigma_n \in T,\ \texttt{next\_SC}(\sigma_n) \ni \langle \sigma', l'\rangle\}$$

Then $\mathbb{O}_{SC}[\![\texttt{o\_sc}]\!](v_{sc}, s_{sc}) = \cup_{n=0}^{\omega} {F_{sc}}^{n}(\emptyset)$

Proof 1 *The proof is immediate from the fixpoint theorem in [1].*

Definition 3.17 *(SystemC Module Semantics:* $\mathbb{M}_{SC}[\![\texttt{m\_sc}]\!]$*))*
Let $\texttt{m\_sc} = \langle \texttt{DMem}, \texttt{Ports}, \texttt{Chan}, \texttt{Mth}, \texttt{SC\_Ctr}\rangle$ *be a SystemC module, then its semantics* $\mathbb{M}_{SC}[\![\texttt{m\_sc}]\!] \in \mathcal{P}(\mathcal{T}(\Sigma))$ *is:*

$$\mathbb{M}_{SC}[\![\texttt{m\_sc}]\!] = \quad \{\mathbb{O}_{SC}[\![\texttt{o\_sc}]\!](v_{sc}, s_{sc}) \mid \texttt{o\_sc} \text{ is an instance of } \texttt{m\_sc},\ \texttt{v\_sc} \in \texttt{D\_in},\ \texttt{s\_sc} \in \texttt{SC\_Store}\}$$

Theorem 3.2 *(SystemC Module semantics in fixpoint) Let*

$$G_{sc}\langle S\rangle = \lambda T. \quad \{\mathtt{S_0}\langle v,s\rangle \mid \langle v,s\rangle \in S\} \cup \{\sigma_0 \xrightarrow{l_0} \ldots \xrightarrow{l_{n-1}} \sigma_n \xrightarrow{l'} \sigma' \mid \{\sigma_0 \xrightarrow{l_0} \ldots \xrightarrow{l_{n-1}} \sigma_n \in T, \mathtt{next_{sc}}(\sigma_n) \ni \langle \sigma', l'\rangle\}$$

Then $\mathbb{M}_{SC}[\![\mathtt{m\_sc}]\!](v_{sc}, s_{sc}) = lfp^{\subseteq}_{\emptyset}\, G_{sc}\langle\, \mathtt{D}_{in} \times \mathtt{Store}\rangle$

Proof 2 *Although the SystemC model presents some additional functionalities on top of C++, the proof of this theorem is similar to the proof of Theorem 3.2 in [2]. For instance, considering the definition of $\mathbb{M}_{SC}$ and applying in order Definition 3.16, Theorem 3.1 and the fixpoint theorem in [1], the proof is straightforward.*

3.4 Soundness and Correctness of the Module Semantics

The last step in the SystemC fixpoint semantics is to relate the module semantics to the whole SystemC program semantics. For this purpose, we consider updated versions of the functions *split* ($\alpha^{\circ}_{\times}$), *project* ($\alpha^{\circ}_{\uparrow}$) and *abstract* (α°) as defined in [2]. The new functions are upgraded to support the SystemC simulation semantics, environment and store. For example, *split_SC* ($\alpha\_\mathtt{SC}^{\circ}_{\times}$) can drop the memory reached by the environment for a method that was previously executed in the current simulation cycle because a method cannot be executed again until the next cycle starts.

The basic concept behind defining the Module object semantics is to cut all the instances not involving the object. For this purpose, two helper functions are required: (1) $\alpha\_\mathtt{SC}^{\circ}_{\times}$ cuts all the traces involving the object instances; and (2) $\alpha\_\mathtt{SC}^{\circ}_{\uparrow}$ maps all the cut instances to interaction states.

Lets first define the helper function `split_SC` that given a trace τ and an object `o_sc` returns a pair consisting of the last state of the prefix of τ made up of the last state of the execution of a method or process of `o_sc` and the remaining suffix of prefix of τ. In contrast to the general case of OO programs, for SystemC, we consider both parts defaults C++ methods and processes according to the following definition:

Definition 3.18 *(The split helper function `split_SC`)*
Let `o_sc` be a SystemC module object, $\tau \in \mathcal{T}(\mathtt{SC\_Env} \times \mathtt{SC\_Store})$, `CurProcess` $\in$ `SC_Pr`, `CurMethod` $\in$ `Mth` and $\mathtt{pc}_{exit}$ be the exit point of $\tau(0)(\mathtt{CurMethod})$. Then `split_SC` $\in [(\mathcal{T}(\mathtt{SC\_Env} \times \mathtt{SC\_Store}) \rightarrow (\mathtt{SC\_Env} \times \mathtt{SC\_Store}) \times \mathcal{T}(\mathtt{SC\_Env} \times \mathtt{SC\_Store})]$ is defined as:

$$\texttt{split\_SC}(\tau) = \textit{let } n = \min\{i \in \mathbb{N} \mid \tau(i)(\texttt{CurProcess}) = \langle \texttt{CurMethod}, \texttt{SL} \rangle \wedge \tau(i)(\texttt{SL}) = \texttt{true} \wedge \tau(i)(\texttt{pc}) = \texttt{pc}_{exit} \wedge \tau(i)(\texttt{this}) = \texttt{o\_sc} \wedge \tau(i)(\texttt{StackHeight}) = \tau(0)(\texttt{StackHeight})\}$$
$$\textit{in } \langle \tau(n), \tau(n+1) \to \ldots \to \tau(\texttt{Len}(\tau) - 1) \rangle$$

The cut function $\alpha\_\mathsf{SC}^{\circ}_{\times}$ considers 4 different cases:

1. for empty trace, ϵ, it returns an empty trace.
2. if trace is part of the object trace then we split it recursively keeping only the last state of the execution of a method or process. The rest of the trace is are removed.
3. If this is not the current object and the store is not changed, then, we continue with the rest of the trace.
4. If this is not the current object and the store is changed, then, we keep the current trace and we continue with the rest of the traces.

Definition 3.19 *(Cut function: $\alpha\_\mathit{SC}^{\circ}_{\times}$)*
Let $\texttt{o\_sc}$ be a SystemC module object, $\tau \in \mathcal{T}(\texttt{SC\_Env} \times \texttt{SC\_Store})$. Then $\alpha\_\mathit{SC}^{\circ}_{\times} \in [(\mathcal{T}(\texttt{SC\_Env} \times \texttt{SC\_Store}) \times \texttt{SC\_Store}) \to \mathcal{T}(\texttt{SC\_Env} \times \texttt{SC\_Store})]$ is defined as:

$$\alpha\_\mathit{SC}^{\circ}_{\times} = \lambda(\tau, \texttt{S}_{\texttt{last}}).$$

$$\begin{cases} \epsilon & \texttt{if } \tau = \epsilon \\ \texttt{let } \langle \rho', \tau' \rangle = \texttt{split\_SC}(\tau) \\ \texttt{in let } \langle e'_{sc}, s'_{sc} \rangle = \rho' & \texttt{if } \tau = \langle e_{sc}, s_{sc} \rangle \to \tau'', e_{sc}(\texttt{this}) = \texttt{o\_sc} \\ \texttt{in } \rho' \to \alpha\_\mathit{SC}^{\circ}_{\times}(\tau', s'_{sc}) \\ \alpha\_\mathit{SC}^{\circ}_{\times}(\tau'', \texttt{S}_{\texttt{last}}) & \texttt{if } \tau = \langle e_{sc}, s_{sc} \rangle \to \tau'', e_{sc}(\texttt{this}) \neq \texttt{o\_sc}, \texttt{S}_{\texttt{/S(o\_sc)}} = \texttt{S}_{\texttt{last/S(o\_sc)}} \\ \langle e_{sc}, s_{sc} \rangle \to \alpha\_\mathit{SC}^{\circ}_{\times}(\tau'', \texttt{S}) & \texttt{if } \tau = \langle e_{sc}, s_{sc} \rangle \to \tau'', e_{sc}(\texttt{this}) \neq \texttt{o\_sc}, \texttt{S}_{\texttt{/S(o\_sc)}} \neq \texttt{S}_{\texttt{last/S(o\_sc)}} \end{cases}$$

The second part of the abstraction includes the $\alpha\_\mathsf{SC}^{\circ}_{\uparrow}$ function which maps the states of a trace to interaction states.

Definition 3.20 *(Map function: $\alpha\_\mathit{SC}^{\circ}_{\uparrow}$)*
Let $\mathtt{o\_sc}$ be a SystemC module object, $\tau \in \mathcal{T}(\mathtt{SC\_Env} \times \mathtt{SC\_Store})$. Then $\alpha\_\mathit{SC}^{\circ}_{\uparrow} \in [(\mathcal{T}(\mathtt{SC\_Env} \times \mathtt{SC\_Store}) \times \mathcal{P}(\mathtt{Addr})) \to \mathcal{T}(\sum)]$ is defined as:
$\alpha\_\mathit{SC}^{\circ}_{\uparrow} = \lambda(\tau, \mathtt{Esc})$.

$$
\begin{cases}
\epsilon & \mathtt{if}\ \tau = \epsilon \\
\begin{array}{l} \mathtt{let}\ \langle e_{sc}, s_{sc}\rangle = \rho \\ \mathtt{in}\ \mathtt{let}\ \mathtt{Esc}' = \mathtt{Esc} \cup \\ \quad \mathtt{reachable\_SC}(\rho(\mathtt{retVal}), s_{sc}) \\ \mathtt{in}\ \langle\langle e_{sc}, s_{sc}, \rho(\mathtt{retVal}), \mathtt{Esc}\rangle, \\ \langle \rho(\mathtt{curMethod}), \rho(\mathtt{inVal})\rangle\rangle \\ \to \alpha\_\mathit{SC}^{\circ}_{\uparrow}(\tau', \mathtt{Esc}') \end{array} & \mathtt{if}\ \tau = \rho \to \tau', e_{sc}(\mathtt{this}) = \mathtt{o\_sc} \\
\begin{array}{l} \mathtt{let}\ \langle e_{sc}, s_{sc}\rangle = \rho \\ \mathtt{in}\ \langle\langle e_{sc}, s_{sc}, \phi, \mathtt{Esc}\rangle, k\rangle \\ \to \alpha\_\mathit{SC}^{\circ}_{\uparrow}(\tau', \mathtt{Esc}) \end{array} & \mathtt{if}\ \tau = \rho \to \tau', e_{sc}(\mathtt{this}) \neq \mathtt{o\_sc}
\end{cases}
$$

The abstraction function $\alpha\_\mathtt{SC}^{\circ}$ projects from the traces of an execution the set of relevant states to a specific object.

Definition 3.21 *(Abstract function: $\alpha\_\mathit{SC}^{\circ}$)*
Let $\mathtt{o\_sc}$ be a SystemC module object, $\mathtt{T} \subseteq \mathcal{T}(\mathtt{SC\_Env} \times \mathtt{SC\_Store})$ a set of execution traces and $\mathtt{s}_{\emptyset}$ the empty store. The the abstraction function $\alpha\_\mathit{SC}^{\circ} \in [(\mathcal{T}(\mathtt{SC\_Env} \times \mathtt{SC\_Store}) \to \mathcal{P}(\mathcal{T}(\sum)]$ is defined as:
$\alpha\_\mathit{SC}^{\circ}(\mathtt{T}) = \{\alpha\_\mathit{SC}^{\circ}_{\uparrow}(\alpha\_\mathit{SC}^{\circ}_{\times}(\tau, \mathtt{s}_{\emptyset}), \emptyset) \mid \tau \in \mathtt{T}\}$

Theorem 3.3 *(Soundness of $\mathbb{M}_{SC}[\![\mathtt{m\_sc}]\!]$) Let $\mathtt{M}_{SC}$ be a whole SystemC program and let $\mathtt{m}_{SC} \in \mathtt{M}_{SC}$. Then*
$\forall\, R_0 \in \mathtt{SC\_Env} \times \mathtt{SC\_Store}.\ \forall\, \tau \in \mathcal{T}(\mathtt{SC\_Env} \times \mathtt{SC\_Store})$.

$$\tau \in \mathbb{W}[\![\mathtt{SC\_Pg}]\!](R_0) : \exists \tau' \in \mathbb{M}_{SC}[\![\mathtt{m}_{SC}]\!].\ \alpha\_\mathit{SC}^{\circ}(\{\tau\}) = \{\tau'\}$$

Proof 3 *(Sketch) We have to consider both cases when τ contains an object $\mathtt{o}_{SC}$, instantiation of $\mathtt{m}_{SC}$, and when it does not include any $\mathtt{o}_{SC}$. For the second situation, the proof of the theorem is trivial considering that τ will be an empty trace. In the first case, the trace is not empty (let it be τ''). Since SystemC modules are initialized in the main program $\mathtt{sc\_main}$ before the simulation starts, there exist an initial environment, store and set of variables that define the initial trace $\sigma_0 \in \tau''$. The rest of the traces in τ'' are interaction states of $\mathtt{o}_{SC}$ because they are obtained by applying $\alpha\_\mathit{SC}^{\circ}$ on τ. Therefore, $\tau'' \in \mathbb{M}_{SC}[\![\mathtt{m}_{SC}]\!]$.*

Theorem 3.4 *(Completeness of $\mathbb{M}_{SC}[\![]\!]$) Let $\mathtt{m}_{SC}$ be a SystemC module. Then*

$$\forall \tau \in \mathcal{T}(\Sigma). \quad \tau \in \mathbb{M}_{SC}[\![\mathtt{m}_{SC}]\!]: \exists\, \mathtt{SC\_P} \in \langle \mathtt{L}_{\mathtt{SC\_Pg}} \rangle.\ \exists \rho_0 \in \mathtt{SC\_Env} \times \mathtt{SC\_Store}.\ \exists\, \mathtt{o}_{SC} \text{ instance of } \mathtt{m}_{SC}.\ exists\ \tau' \in \mathcal{T}(\mathtt{SC\_Env} \times \mathtt{SC\_Store}).\ \tau' \in \mathbb{W}[\![\rho_0]\!] \wedge \alpha\_\mathtt{SC}^{\circ}(\{\tau'\}) = \{\tau\}$$

Proof 4 *(Sketch) A SystemC program satisfying the previous theorem can be constructed by creating and instance of $\mathtt{m}_{SC}$ in the `sc_main` function, the initial state corresponds to the state when the module's constructor, `SC_Ctr`, was executed. An execution of a method of $\mathtt{m}_{SC}$ corresponds to executing a method thread (setting of the events in its sensitivity list to* Active*) and a change of a port corresponds to updating its internal signal by the new values. Hence, it is always possible to construct both $\mathtt{SC\_P}$ and ρ_0. For instance, there exist many other possible constructions involving SystemC threads, clocked threads, etc.*

4 Conclusion

In this report, we presented the fixpoint semantics of the SystemC library. Then, we proved the soundness and the correctness of the the semantics of the SystemC basic class SC_Module w.r.t. to a trace semantics of a the whole SystemC program. Such a result presents a first step towards applying formal methods to SystemC. In particular, the concrete semantics, we defined, can be used to construct sound abstract semantics allowing static code analysis or model checking of SystemC programs.

References

[1] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, TX, USA, 1979.

[2] F. Logozzo. *Anhalyse Statique Modulaire de Langages a Objets*. PhD thesis, Ecole Polytechnique, Paris, France, June 2004.

[3] W. Müller, J. Ruf, and W. Rosenstiel. *SystemC Methodologies and Applications*. Kluwer Academic Pub., 2003.

[4] P. D. Mosses. *Denotational semantics*, volume B of *Handbook of Theoretical Computer Science*, chapter 11, pages 575–631. Elsevier Science B.V., 1990.

[5] Open SystemC Initiative. http://www.systemc.org, 2004.

[6] A. Salem. Formal semantics of synchronous SystemC. In *Proc. Design, Automation and Test in Europe Conference*, pages 376–381, Munich, Germany, March 2003.