A Mathematical Characterization of System Design and Modeling

Xibin Zhao1,∗Anping He2,3, Jinzhao Wu2,4, Guowu Yang5, Yi Yang3, Ning Zhou3, Shihan Yang2, Lian Li2

1Key Laboratory for Information System Security of Ministry of Education, School of Software, Tsinghua University, China.
2Guangxi Key laboratory of Hybrid Computational and IC Design Analysis, Naning, China.
3School of Information Science and Engineering, Lanzhou University, China.
4School of Computer and Information Technology, Beijing Jiaotong University, China.
5School of computer Science and Software Engineering, University of Electronic Science and Technology, China.

Published online: 1 May 2012

Abstract: SystemC is an emerging standard hardware description language for system-level modeling and design. Formal semantics could give its meaning in a mathematically rigorous and unambiguous way. In this paper, we formalize SystemC both from processes and simulation environment by process algebra, which enables that deduction and verification of the whole simulation procedure are in an integrated and convenient manner. Meanwhile, in order to accommodate event-driven and concurrent properties of SystemC, we adopt event structure characterized by true concurrency, as semantic model of both SystemC and its simulation environment for denotational semantics and operational semantics, as well as demonstrate the correspondence between them. As a result, we propose a unified framework for formalizing SystemC.

Keywords: Denotational semantics, Operational semantics, Bi-simulation, Process algebra, Timed stable event structure.

1. Introduction

In hardware design procedures, hardware description languages (HDLs) are typically used to express designs at various levels of abstraction. HDLs are high level programming languages, whose programming constructs usually contain of assignments, conditions, iterations and appropriate extensions for real-time, concurrency and data structures being suitable for modeling hardware. SystemC HDL, developed by Open SystemC initiative (OSCI), is a modeling language based on C++. It models the system at different levels of abstraction, fills the gap between hardware and software design, provides a unifying language for hardware/software specifications, and becomes an IEEE standard [3] for system level modeling and design. SystemC contains C++ libraries and simulation kernel for creating behavioral and register-transfer level designs and provides a common development environment needed to support software engineers working in C/C++ and hardware engineers working in HDLs such as VHDL, Verilog, etc.

Although SystemC comes with a well-written Language Reference Manual [3] and a reference implementation of the simulator, the documentation leaves some open questions w.r.t. the precise and rigorous meaning. However, a precise formal semantic of SystemC is mandatory for various applications in simulation, synthesis and formal verification, provides a completely unambiguous specification of the language, and contributes significantly to the sharing, portability and integration of various applications. On the other hand, compared with other HDLs, e.g., Verilog and VHDL, there exist some new and interesting programming features of SystemC: processes trigger events actively instead of generating by the changes of states (e.g., Verilog), events represent some general conditions during the execution of program, and can be notified and canceled on many separate occasions. There are three kinds of event notifications: immediate event notification, delta-cycle delayed notification and timed notification. Timed notifications can be canceled via ‘cancel’ statements before they are triggered. Delayed notifications on the same event override each other and only one delayed notification

∗Corresponding author: e-mail: hapetis@gmail.com
survives. All of these new features make it worthwhile to give a precise semantics of the language. Since SystemC is essentially an event-driven HDL, it is convenient and natural that the semantic model could reflect these interesting features, fortunately, event structure [4, 5] answers precisely to our need. Event structure is capable of specifying causality, concurrency and choice between events. It has shown its strength in the construction of parallel and distributed systems, as well as in the initial steps of asynchronous circuit design. It was used successfully for the analysis and synthesis of digital systems [6]. Moreover, it is easy to do action refinement operation [7] on event structures, which is congruent with the top-down design of SystemC. Meanwhile, partial order reduction over it also facilitates the verification.

Our research focuses on a unified way to study SystemC and its simulation environment for two reasons. Firstly, according to the point of formal specification, processes and attached environment could not be split (as mentioned in [8]). Process is the cornerstone of semantics of a language. Environment shows the dynamic occasion of processes. Therefore it is a prerequisite to investigate both processes and the environment. For SystemC, the unified way makes the simulation exact, simple and optimized.

Secondly, compared to some well-known model checking successes like the JAVA-PATHFINDER, why does programming language, like C and C++ are more difficult to be verified? Because Java programming language is compiled to the same byte code for Java virtual machines running on all kinds of operation systems. Java essentially encapsulates both the executable codes and information of environment, which simplifies the model-checking process. However, the executable codes for C and C++ depend on their compilers and operating systems.

In this paper, we study SystemC to precisely obtain a unified semantic of both SystemC and its simulation environment with a well-defined semantic model, which is the first step to find an effective and applied way of verification of SystemC.

1.1. Related work

Although there are some works on formal semantics [9] and algebraic laws for the hardware modeling languages [10] by far, to the best of our knowledge, there is no publication concerning the unification of processes and simulation environment of SystemC.

The simulation semantics of SystemC in the form of distributed Abstract State Machine (ASM) specifications and the denotational semantics for a synchronous subset of SystemC were studied in [11] and [12] respectively. In [13], the SystemC Process State Machines were described as a variation of the UML method state machines. In [10], the event-driven properties, operational semantics and algebraic law of SystemC were well studied, but the environment was not involved.

Moreover, previously, we have studied the SystemC by a well-defined event based sub-set of SystemC language, e.g., SystemC\textsuperscript{\textsc{pa}} [14], which is similar to K.L. Man’s SystemC\textsuperscript{\textsc{pl}} [9, 15, 16], but Man’s article focuses on SystemC\textsuperscript{\textsc{pl}} but not the environment of running processes and event-driven model.

1.2. Notations

In this article we denote an action with lower Greek letter, and the corresponding event is $e$ with a suffix of this letter. In order to reflect the essentials of the user defined actions, including Assign, Test, Change, Notify, Clock$_7$ and Clock$_x$ are also used. Moreover, the capital letters $P$, $Q$ and $B$ are used to denote the names of processes or behaviors and $G$, $E$ and $X$ for set. We use letters with math style, e.g., $D$ and $A$, to denote the special set, mapping and so on.

1.3. Organization

The paper is organized as follows: In Section 2, we show some basic notations and definitions of timed stable event structure, process algebra and its semantics. In Section 3, we depict the syntax of SystemC in detail and propose a unified modeling framework for both processes and simulation environment. We conduct a case study in Section 4 and conclude in Section 5.

2. Real-Timed Extension of Stable Event Structures and Process Algebra

Event structure is model of processes as events constrained by relations of consistency and enabling [4], which allow modeling of systems by specifying branching structure, causal ordering and concurrent running. An event structure is a set of events together with relations of partial-order and conflict. The partial-order relation models causality, whereas the conflict relation expresses alternative choices between events. Two events which are neither causally dependent nor in conflict may occur concurrently. In this sense, event structures provide explicit and separate representations of causality, choice and concurrency [17].

Stable event structures were introduced by Winskel to overcome the unique enabling problem of prime event structures [4]. This kind of event structures has an enabling relation, denoted $\vdash$, relating a (usually finite) set of events to a single event. The interpretation of $X \vdash e$ for a set $X$ of events and an event $e$ is that $e$ is enabled if all events in $X$ have occurred. Events are the occurrence of actions, then a event structure are always involved a well-defined action set, let $\mathcal{A}$ be the set.

An event structure is defined as a tuple $E = (E; \sharp, \vdash )$, where $E$ is a countable set of events; $\sharp \subseteq E \times E$ is
a symmetric and irreflexive relation, the conflict relation; \(\vdash\subseteq\mathsf{Con}\times E\) is an enabling relation such that \(X\vdash e\land X\subseteq Y\in\mathsf{Con}\Rightarrow Y\vdash e\) (here \(\mathsf{Con}\) is the set of finite conflict-free subsets of \(E\), i.e. those finite subsets \(X\subseteq E\) for which holds: \(\forall e,e'\in X: \neg(e\equiv e')\)). An event structure is called stable if \(X\vdash e\land Y\vdash e\land X\cup Y\subseteq\mathsf{Con}\Rightarrow X\cap Y\vdash e\). The set \(\mathsf{Con}(\mathsf{E})\) of configurations of \(\mathsf{E}\) consists of those \(C\subseteq E\) which are secured (\(\forall e\in C:\exists\{e_0,\ldots,e_n\}\subseteq C\) s.t. \((e_n=e)\land\forall i\leq n:\{e_0,\ldots,e_{i-1}\}\vdash e_i\) and conflict-free (\(\forall e,e'\in C: \neg(e\equiv e')\)).

Since the time-related properties, e.g., time-delay, is essential during SystemC simulation, we will extend the stable event structure with a real-time extension. Assuming a global clock [5] which approximates the time of every snapshot by a natural number. In our model, events are attached with time delays and the causalities. This kind of extension can be interpreted as: an event with a delay \(t\) will not be bundled by the time constraint at the time \(t\) since the start of the system and may happen at any time from \(t\) (if enabled) and a causality with \(t\) delays enabled event \(t\) time units. Note: the occurrences of events themselves take no time, i.e. events happen ‘instantaneously’ [5].

Let \(\mathbb{N}\) be the set of natural numbers, then time domain can be denoted by \(\mathbb{N}=\mathbb{N}\cup\{\infty\}\), where \(\infty>\) \(n\) for any \(n\in\mathbb{N}\). We use \(\mathsf{CF}(S)\) to denote that all the events in set \(S\) is conflict-free.

\[
\mathsf{CF}(S) = \forall e,e' \in S: (e \neq e') \Rightarrow \neg(e\equiv e')
\]

and \(\mathsf{CF}(S)\) to the set of events that conflict with \(S\):

\[
\mathsf{CF}(S) = \{e : e \in E \land e' \in S \land \neg\mathsf{CF}(\{e,e'\})\}
\]

**Definition 1.** A timed event structure is a septuple \(\mathsf{TE} = (E,\preceq,\vdash,\mathcal{T},D)\) with:

- \(E\), a set of events;
- \(\preceq\subseteq E \times E\), the (irreflexive and symmetric) conflict relation;
- \(\vdash\subseteq\mathcal{P}(E)\times E\), the enabling relation;
- \(\mathcal{T}:E\rightarrow\mathbb{N}\), the action-labeling function;
- \(D:E\rightarrow\mathbb{N}\), the event delay function;
- \(\mathcal{T}:E\rightarrow\mathbb{N}\), the causality delay function.

such that

1. \(\forall X \subseteq E, e \in E : X\vdash e \Rightarrow \mathsf{CF}(X\cup\{e\})\);
2. \(\forall X,Y \subseteq E, e \in E : (X\vdash e \land Y\vdash e) \Rightarrow (\neg\mathsf{CF}(X\cup Y) \lor Y = X)\).

In the following, we will consider only timed stable event structure and call it simply event structure.

The event structure \(\mathsf{TE} = (E,\preceq,\vdash,\mathcal{T},D)\) has correct timing iff for any \(e\in E\) and \(X\vdash e\), \(\mathcal{T}(X) \leq D(e)\). In what follows, only timed event structures with correct timing are considered.

Moreover, event structure can be depicted by figures, in which events are denoted by closed dots, conflict relations by a dotted line, enabling relations by drawing an arrow from each event in \(X\) to \(e\) and connecting all arrows by a small line, action labels are given near the dot, events and causality delays are depicted near the corresponding dots and lines respectively, and zero time unit delay is usually omitted. So the event structure pictured in Figure 4(a) can be described as \((E,\preceq,\vdash,\mathcal{T},D)\) with \(E = \{e_0,e_3,e_4\}\), \(\preceq = \{(e_3,e_4)\}\), \(\vdash = \{(e_0,e_3),(e_0,e_2),(e_2,e_0),(e_2,e_3),(e_2,e_4),(e_4,e_2),(e_4,e_3),(e_4,e_0)\}\), \(\mathcal{T} = \{(e_0,\alpha),(e_2,\beta),(e_4,\gamma),(e_4,\eta)\}\), \(D = \{(e_0,3),(e_3,14),(e_4,11)\}\) and \(T = \emptyset\).

Behaviors of system specified by a \(\mathsf{TE}\) are described by explaining which subsets of events constitute possible (partial) runs of the represented system. These subsets are modeled by sequences of timed events termed traces. Traces are sequences of timed events \(\sigma = (e_1,\ldots,e_n)\), such that \(\forall i : e_i\in\mathsf{TE}\wedge|\{e_i\}|\leq \mathcal{T}(\{e_i\})\) and \(\forall i < j : e_i\preceq e_j\). Now, \(e\) is enabled after \(\sigma\) if there exist certain events that occurred earlier with the flow relation pointing to \(e\), and meanwhile \(e\) should not in conflict with other earlier events in \(\sigma\), that is: \(\forall \{e_i\} \subseteq \mathsf{TE} : (e \in \mathcal{P}(\{e_i\}) \land e \notin \mathsf{CF}(\{e_i\}) \land \exists \mathcal{X} \subseteq [\sigma], X\vdash e)\). Let \(\mathsf{time}(e,\sigma)\) denote the minimal time instant from which \(e\) can occur, two aspects should be considered: (i) \(e\)’s absolute delay \(D(e)\), (ii) the time relative to all \(e\)’s causality, so \(\mathsf{time}(e,\sigma) = \mathsf{Max}(\{\mathcal{T}(e)\} \cup H)\) with \(H = \{t_i : \exists X \subseteq [\sigma], X\vdash e, t_i \leq \mathcal{T}(e)\}\). Let \(\sigma_1 = e_1\cdots e_{i-1} = (i-1)\)-th prefix of \([\sigma]\), so the definition of trace is as following:

**Definition 2.** A trace of timed event structure \(\mathsf{TE} = (E,\preceq,\vdash,\mathcal{T},D)\) is a sequence \(\sigma\) of timed events \((e_1,t_1)\cdots(e_n,t_n)\) with \(e_i\in\mathsf{TE}\) and \(t_i\in\mathbb{N}\) and \(i\in\mathbb{N}\), satisfying

- \(e_1\cdots e_n\) with \(e_i\in\mathsf{en}(\{e_{i-1}\})\);
- \(\forall i : e_i\in\mathsf{en}([\sigma_{i-1}])\);
- \(\forall i,j : (i < j \Rightarrow (t_i \leq t_j))\).

A state of an execution of a timed event structure \(\mathsf{TE}\) is called a timed configuration \(TC = \sigma\). Let \(\mathsf{Conf}(\mathsf{TE})\) denote the set of timed configurations of \(\mathsf{TE}\). Say that there is a transition from a timed configuration \(TC\) to a timed configuration \(TC'\) and write \(TC\rightarrow TC'\) iff \(TC\subseteq TC'\). It is easy to see that \(\rightarrow\) includes a partial order on \(\mathsf{Conf}(\mathsf{TE})\). Moreover, the class of this structure is denoted by \(\mathsf{SES}\) for convenience.

### 2.1. Process Algebra

We use the process algebra as the language to express behaviors of SystemC and its environments. The process algebra in this article is a time extension of LOTOS[18], which is a standard language without data types and value passing. Two timed features are attached, e.g., delay function restricting the occurrence time of atomic actions and transition between events, this two delay constraints are the most important to main features of SystemC, including notifications, notification cancelation, notification overriding, event waiting, time delaying, sensitivity lists, concurrent processses and delta-cycle. The syntax of (timed)
2.2. The Denotational Semantics

The causality-based denotational semantics relates the syntax of process algebra with event structure directly. This type of semantics can give a rigorous and unambiguous interpretation, and then show the true-concurrency character essentially.

Let \( \text{init}(\mathcal{T}E) \) denote the set of initial events of \( \mathcal{T}E \), \( \text{exit}(\mathcal{T}E) \) the set of successful termination events, and \( \text{pos}(\mathcal{T}E) \) that events with non-zero delay: \( \text{init}(\mathcal{T}E) \equiv \{ e \in E \mid \neg(\exists X \subseteq E : X \ni e) \} \), \( \text{exit}(\mathcal{T}E) \equiv \{ e \in E \mid l(e) = \delta \} \) and \( \text{pos}(\mathcal{T}E) \equiv \{ e \in E \mid D(e) \neq \emptyset \} \). Let \( \text{pin}(\mathcal{T}E) \equiv \text{pos}(\mathcal{T}E) \cup \text{init}(\mathcal{T}E) \). Function \( \mathcal{E}[] \) (also adopted in [5]) associates each process term \( P \in \mathcal{P} \) with an element of \( \text{SES} \). The causality based denotational semantics can be defined with \( \mathcal{E}[] \) respectively. Firstly, let \( \mathcal{E}[P] = \mathcal{T}E_i = \langle E_i, \xi_i, \tau_i, l_i, D_i, T_i \rangle \), for \( i = 1, 2 \) with \( E_1 \cap E_2 = \emptyset \). We show the denotational semantics of syntax of PA step by step:

1. The semantics of deadlock (0), successful termination (\( \tau \)) and relabeling (\( H \)) are self-explanatory:

\[
\begin{align*}
\mathcal{E}[0] & \equiv (0, 0, 0, 0, 0, 0), \\
\mathcal{E}[\tau] & \equiv (E, 0, 0, l, D, T), \\
\mathcal{E}[^{H}E] & \equiv (E, 0, 0, l, D, T)
\end{align*}
\]

\( ^{\circ} \) is operator for composition of functions

2. The semantics of action-prefix \((t)\alpha \) of action \( \alpha \), for an enabling relation is introduced from a new event set \( e_{\alpha} \) to all initial events in \( \mathcal{T}E_1 \):

\[
\begin{align*}
\mathcal{E}[(t)\alpha P_1] & \equiv \langle E \cup \{ e_{\alpha} \}, l_1, D \cup \{ (e_{\alpha}, l_1) \}, D, T \rangle
\end{align*}
\]

3. The semantics of Choice (+)

\[
\begin{align*}
\mathcal{E}[P_1 + P_2] & \equiv \langle E_1 \cup E_2, l_1 \cup l_2, D_1 \cup D_2, T_1 \cup T_2 \rangle
\end{align*}
\]
1. Successful terminated
\[ \sqrt{\xi \gamma \eta \zeta \alpha \beta} \]
2. Action prefix
\[ (t) \eta \xi \gamma \eta \zeta \alpha \beta \]
3. Alternative
\[ P_1 \parallel \gamma \eta \zeta \alpha \beta = P_1' \parallel \gamma \eta \zeta \alpha \beta \]
4. Sequential
\[ P_1 \parallel \gamma \eta \zeta \alpha \beta \parallel P_2 \]
5. Parallelism
\[ P_1 \parallel \gamma \eta \zeta \alpha \beta \parallel P_2 \parallel \gamma \eta \zeta \alpha \beta \]
6. Relabeling
\[ (\xi, \gamma \eta \zeta \alpha \beta) \rightarrow P' \parallel \gamma \eta \zeta \alpha \beta \]
7. Process instantiation
\[ \phi(B) \parallel \gamma \eta \zeta \alpha \beta \]

**Figure 1** Event based operational semantics for Process Algebra

6. The Semantics of Process Instantiation

The semantics of the process instantiation operator is much more complex. It involves fix-point semantics according to a seminal work of Winskel [17], which shows that categories of prime and stable event structures can be related to a category of Scott domains by adjunctions, e.g., for process instantiation, \( P := B \), we shall look for event structures satisfying equations of the form \( T \mathcal{E} = \mathcal{F}_B(T \mathcal{E}) \). Let us see partial order relation on event structures, which is a time extension version of the one from Winskel [4]:

**Definition 3.** Let \( T \mathcal{E}_i = (E_i, \tau^i, \tau^i, \rho^i, \rho^i, \delta^i, \delta^i) \), then \( T \mathcal{E} \preceq T \mathcal{E}_2 \) if

\[ E_1 \subseteq E_2, \]
\[ \tau_1 = \tau_2 \cap (E_1 \times E_1), \]
\[ \rho_1 = \{ (X \cap E_1, e) \mid e \in E_1 \cap \tau_2 e \}, \]
\[ \rho_1 = \rho_2 \]
\[ \delta_1 = \rho_2 \cap E_1, \]
\[ \delta_1 = \delta_2 \cap E_1, \]
\[ \forall e \in E_1, T \mathcal{E}_2(X, e) = T \mathcal{E}_1(X \cap E_1, e). \]

Let \( T \mathcal{E}_1 \preceq T \mathcal{E}_2 \preceq \cdots \preceq T \mathcal{E}_n \) be a partial order of \( (SES, \preceq) \), and \( T \mathcal{E}_1, T \mathcal{E}_2, \ldots \in SES \) are upper bounds of the order. We can construct \( T \mathcal{E} = (E, \tau, \rho, \delta, T) \) to be the l.u.b. (least upper bound) of the chain \( T \mathcal{E} \preceq T \mathcal{E}_2 \preceq \cdots \preceq T \mathcal{E}_n \), e.g.,

\[ E = (E_1 \cap E_2 \cdots) \cup E_n, \]
\[ \tau = (\tau_1 \cap (E \times E_1)) \cup \cdots, \]
\[ \rho = (\rho_1 \cap E) \cup \cdots, \]
\[ \delta = (\delta_1 \cap E) \cup \cdots, \]
\[ T = (T_1 \cap E) \cup \cdots. \]

According to the definition 3, \( T \mathcal{E} \preceq T \mathcal{E}_1, \ldots \) and \( T \mathcal{E} \preceq T \mathcal{E}_n \). So \( (SES, \preceq) \) is a c.p.o (complete partial order). Moreover, it is easy to know that 0, e.g. \( (\emptyset, \emptyset, 0, 0, \emptyset) \), is the least element under \( \preceq \). So this partial order is a pointed c.p.o. [19].

Then let \( \sqsubseteq \) \( T \mathcal{E}_1 \) be the least upper bound under this partial order, \( T \mathcal{E}_1 \preceq T \mathcal{E}_2 \preceq \cdots \preceq T \mathcal{E}_n \) be a chain, the least upper bound (under \( \preceq \)) is as following:

\[ \sqsubseteq \bigcup \mathcal{F}_B(T \mathcal{E}_i) \]

**Figure 3** Examples of semantics for parallel and relabeling.
2.3. Correspondence With Semantics

It is necessary to prove the consistency between the denotational semantics over event structures and its operational semantics deduced by the inference rules. The general way is to get the coherence between the two semantics, which means these two transition systems must be bi-simulative.

Let $Ev$ be the set of events and $A^{\phi}$ be a set of actions subscripted with a Greek letter. Then we can define the transition between the event structure, e.g., event transition system:

**Definition 6.** An event transition system is a quadruple $(Conf(TE), \rightarrow, L)$, that

$$P \xrightarrow{(e,\alpha,t)} P'$$

with $e \in Ev \land P, P' \in Conf(TE) \land \alpha \in A \land t \in \mathbb{N}$, a transition relation,

$$L = \{(e, t) \mid \exists \alpha \in A \land P, P' \in Conf(TE) \land P \xrightarrow{(e,\alpha,t)} P'\},$$

a set of labels.

Let $P$ be a SystemC behavior, $TS(P)$ is denoted as the transition system obtained by applying the inference rules of Figure 1 to $P$. On the other hand, the event structure of $P$ is $ES[P]$. According to definition 6, an event based transition system from $ES[P]$ could be constructed, e.g., $ETS(ES[P])$. These two systems are bisimular:

**Lemma 2.** $\forall B \in System^{PA}, TS(B)$ and $ETS(ES[B])$ are bisimilar.

We have the detailed definitions and proof, but omit them in this paper.

3. Formalization of SystemC

Our work is based on the IEEE Standard 1666-2005 version of SystemC [3] and to investigate formal semantics of SystemC, which provides a complete and unambiguous specification and contributes significantly to the sharing, portability and integration of various applications including simulation, synthesis and especially, formal verification.

In order to do the formal analysis and verification, it is necessary to rewrite the SystemC statements by a formal language. Previously, we study the SystemC by a well-defined event based sub-set, e.g., SystemC$^{PA}$ [14], which is similar to K.L. Man’s SystemC$^{\varphi}$ [9,15,16], but Man’s article does not include the environment of running processes and an event-driven model.

However, the current SystemC IEEE Standard [3] does not include the ‘Watching statements’, e.g., local watching and global watching, have been deprecated. Then it is not necessary to express the SystemC statements by a specific operator of process, e.g., the process algebra with timed extension of equation 1 is enough to rewrite all the statements of SystemC and powerful to cover all main features of SystemC.

**Lemma 1.** $\alpha \in FTS$, $\downarrow, \lceil \alpha \rceil, \llbracket \phi \rrbracket$, and $\sigma$ are continuous on $(SES, \leq)$.

Winskel has proven that $\llbracket \alpha \rceil, \lceil \alpha \rceil, \llbracket \phi \rrbracket$ continue on untimed event structure in [4]. Therefore, we only need to do some work to extend them with time, which is similar to [5]. The proof is omitted without bothering readers.

Let $F_B$ be a function over SES with all process operators introduced before (like $\llbracket \alpha \rceil, \lceil \alpha \rceil, \llbracket \phi \rrbracket$), then the semantics of process instantiation is as following:

**Definition 5.** The semantics of $P := B$ is $ES[P] \triangleq \cup_i F_B(\downarrow_i)$. In Figure 4, (a) shows the event structures for $F_B(\downarrow)$. (b) depicts $ES[B]$.

\[ F_{B}(\downarrow_{i}) \]

\[ E_{S}(B) \]

\[ B \]

\[ (a) \] Figure 4 An example of semantics for instantiation

with:

$$\uparrow = \{ \bigcup X_j, e \mid \exists k : \forall j \geq k : X_j \uparrow e \land X_{j+1} \cap E_j = X_j \}$$

$$T = \{ \bigcup X_j, e, t \mid \exists k : \forall j \geq k : T_j(X_j, e) = t \land X_{j+1} \cap E_j = X_j \}$$

Let $F$ be a function over SES, see the definition of continuity of $F$:

**Definition 4.** Let $(SES, \leq)$ be a pointed c.p.o. and $F : SES \rightarrow SES$. $F$ is continuous iff $F$ is monotonic and for any chain $TE \preceq T\varepsilon_2 \preceq \ldots$, we have $E(F(\sqcup_i T\varepsilon_i)) \subseteq E(\sqcup_i F(T\varepsilon_i))$.

Then let us investigate the process operators $\ldots, +, [], \llbracket \cdot \rrbracket$. Firstly, we distinguish all action prefix and $\sqrt{\cdot}$ occurrences by subscription with a Greek letter, let the operator of subscription be $\phi$. For instance, $P := (t)\alpha.P + \beta.P$ becomes $\phi(P) := (t)\phi(\alpha).P_\phi + \beta.P_\phi$. And secondly, we denote for operator $op \in PA$ the corresponding counterpart on event structures by $op$.
The syntax described in equation 1 assumes a given set of actions, containing four kinds abstracted from SystemC: (1) a silent or internal action denoted by \( \tau \), performing the invisible action doing nothing; (2) non-user defined actions, denoted by \( A_{NU} \), including Assign, Test, Change, Notify, \( \Delta \), \( \text{Clock} \), and \( \text{Clock} \); (3) user defined actions, denoted by \( A_U \), including all actions enabling the user defined events; and (4) a special action \( \delta \) indicates the successful termination of a process. Let \( A = A_{NU} \cup A_U \).

The precedences of the composition operators are, in descending order of their binding strength: \{\ldots, +, ||, []\}. The operators inside the braces have equal binding strength. In addition, operators of equal binding strength associate to the left, and parentheses are omitted when they do not introduce ambiguities.

The above algebraic syntax follows the principle of compositionality, as a result, the interpretation of each composite behavior expression can be defined as a function of the interpretation of its constituents, e.g., a complex SystemC behavior could be gained from composed simple ones.

For convenience, we use the special predicate \( \text{same}_v \) to indicate that the events (Assign, Test or Change) deal with the same variables:

\[
\text{same}_v(\alpha_1, \alpha_2) = \begin{cases} 
\text{true} & \alpha_1, \alpha_2 \text{ for same variable;} \\
\text{false} & \text{others.}
\end{cases}
\]

with \( \alpha_1, \alpha_2 \in \{\text{Assign}, \text{Test}, \text{Change}\} \).

3.1. Actions

3.1.1. User Defined Actions and Non-user Defined Actions

User defined actions model the user defined events in SystemC statements, once these events happen the events occur. And the names of these actions are the names of the events with a prefix ‘ud.’. For example, the user defined events write and read (see bellow) correspond to the actions ud.write and ud.read.

\[
\ldots \text{sc_event write, read;}
\]

Non-user defined actions are much more complicated: the Assign action models the assignment statement, happens while assigning the value of some expression to a variable (the simple variables and signals); the Change action will be enabled while some signals are being changed, monitoring the sensitivity list and system clock (for clock thread); the Test action occurs while one condition is satisfied, defined for modeling choice, like the ‘if-then’ statements; the Notify action formalizes the notification statements, happens once this statement occurs; the \( \Delta \) action occurs while one delta-cycle finish; and the \( \text{Clock} \) and \( \text{Clock} \) are used to model the system clock, representing the clock positive and negative which may also be included in the sensitivity list (see Figure 5 line 11 and Figure 8 line 10).

According to SystemC, Assign, Change, Test, Clock and Clock are relevant: for a concrete variable, once one Assign happens, the corresponding Test is enabled, occurrence of two different Test actions lead to a Change be enabled (for monitoring sensitivity list). Moreover, Notify is relevant to \( \Delta \) (see section 3.2).

For example, in Figure 9, Assign.load=true (line 5) and Assign.load=false (line 8) do not only enable the actions Test.load=true and Test.load=false respectively, which are used for determining the condition in Figure 6 line 4, but also the action Change.load (although this action is useless).

3.1.2. \( \delta \) and \( \tau \)

For the efficiency and clarity of studying systems, researchers always reside in a specific abstraction level. In fact, it is impossible to model a system without any abstraction. Invisible action, \( \tau \), is from a lower abstraction of system, which is silent and invisible to the higher researched level. In the lower level, \( \tau \) may be plentiful, but system will always perform some kinds of invisible actions before halted. In this article, \( \tau \) is only a placeholder in most cases if not specially described. Moreover, for the clarity and vividness, we sometimes add a suffix for \( \tau \), for example \( \tau_{\infty} \) which represents \( \tau \) being enabled after infinite time delay.

Similar to [5], \( \delta \) is a special action, indicating the successful termination of a process.

3.2. Processes

Of course, the processes defined in the kernel subset of SystemC contain all kinds of processes in SystemC and cover most of its statements. In this section, we introduce the unified way of formalizing the processes of SystemC and the simulation environment by process algebra theory.

The processes of a system usually do not run alone, but adheres to their surrounding environment. We believe the study of formalization of some concrete systems must involve both the processes and their adjacent environment. Here, the environment of processes of SystemC is essentially the simulation environment, which includes system clock and other things needed, for example, during a running of a process, Assign happens, the corresponding Test triggered and maybe the Change for the same variable.
also enabled, all of which would be useful for the remaining behaviors of the processes or even other parallel processes. Although Test and Change are not the real actions abstracted from SystemC, but actually needed for reacting with the simulation environment.

Let us continue from action prefix and system clock, the basis for constructing process, whose timing properties are well represented by suffix. For an action α and a behavior B, action prefix (t)α.B denotes, after t time units since the start of the system, a behavior which may engage in α and after which it behaves like B. Then the simplest process may be (t)α.√ and all complex ones are constructed from the simple with process operators in a compositional way. Then the system clock is denoted as P_{clock} with P_{clock} := Clock.\_Clock.\_P_{clock}.

Like other hardware description languages, such as Verilog and VHDL, parallel, sequentiality, branch (choice) and loop are the basic structures for a process of SystemC. Let us depict these keys gradually.

SystemC deals with the variables and their values essentially, of which the assignments and judgements are often difficult to process for event structure based models. Unlike value passing in CCS [20] or other related languages, we add no individual and specific mechanism to process algebra for this problem, but only several related actions and the environment. For simplicity, we only consider the value expression of an assignment statement, which could be vividly formalized as Assign_Asgn_exp (only Assign for simplicity here). Now assume an assignment statement was met after P_{1} was performed, this behavior could be formalized as P_{1}(Assign.√), but its simulation environment is much more complicated. Once Assign occurs, Test for this assignment is triggered (section 3.1.1), and in order to ensure two different Tests could enable the Change for the same variable, relabeling operator should be applied, so a complete form is P := P_{1}(Assign.√) and env = τ \{ Clock.\_Clock \} P_{clock} \{ Assign.\_Test.\_\{H\} \}, that H(\text{Test}) = Change \& A(\text{Test.}0) ≠ A(\text{Test.}0) ∧ same_{0}(\text{Test.} Test') ∧ (Test' ∈ \text{Env}) and H(\text{Test}) = \tau ∧ A(\text{Test.}0) = A(\text{Test'.}0) and same_{0}(\text{Test.} Test'), (Test' ∈ \text{Env}.

After investigating the example above, we say the formalized process contributes to both the process itself and the environment, as illustrated in the following snippet of SystemC code:

\begin{verbatim}
P_{1} ... if (load) { 
    count_val = din;
} else { 
    count_val = count_val +1;
} 
end = count_val;
P_{3} ...
\end{verbatim}

Formally, the above code show that an assignment occurs after a choice, which could be modeled with sequential processes as a whole. The non-deterministic choice operator here reacts with the environment. The formal representation of the behavior of the above code snippet could be: P := P_{1}(η.α_{1}.0 + α_{2}.0); α_{3}.0; P_{2} and env = τ \{ Clock.\_Clock \} P_{clock} \{ env_{p1} \{ (α_{1}.γ_{1}.0[H]) \} \{ (α_{2}.γ_{2}.0[H]) \} \{ (α_{3}.γ_{3}.0[H]) \} \{ (η.ζ.η.0) \} \}, that H_{i}(\text{Test}) = Change_{count{0}} \& A(\text{Test}.0) ≠ A(\text{Test}.0) ∧ same_{0}(γ_{1}.Test) ∧ Test ∈ \text{Env} with i = 1, 2, H_{3}(\text{Test}) = Change_{out} \& A(\text{Test}.0) ≠ A(\text{Test}.0) ∧ same_{0}(γ_{3}.Test) ∧ Test ∈ \text{Env}

In general, the loop repeats some behaviors until a specific condition is not satisfied, for which Process instantiation would be used. Because the P_{i} is behind a loop whose condition will be always true, it should be ignored for simplicity. Now the behavior of the whole codes above will be P = P_{1}; P' with P' = ε_{1}.0; ε_{2}.0; P_{2}; ε_{3}.0; P_{3}; P' and the environment, env = τ \{ Clock.\_Clock \} P_{clock} \{ env_{p1} \{ (ε_{1}.ε_{1}.0[H]) \} \{ (ε_{2}.ε_{2}.0[H]) \} \{ (ε_{3}.ε_{3}.0[H]) \} \}, which will collect the Test and Change actions generated by P', and

\begin{verbatim}
H_{i}(\text{Test}) = Change_{count{0}} \& A(ε_{i}.0) ≠ A(\text{Test}.0) ∧ same_{0}(ε_{i}.Test) ∧ Test ∈ \text{Env} with i = 1, 2,
H_{3}(\text{Test}) = Change_{out} \& A(ε_{3}.0) ≠ A(\text{Test}.0) ∧ same_{0}(ε_{3}.Test) ∧ Test ∈ \text{Env} and
H_{i}(ε_{i}) = \tau \& A(ε_{i}.0) = A(\text{Test}.0) ∧ same_{0}(ε_{i}.Test) ∧ (Test ∈ \text{Env}) with i = 1, 2, 3.
\end{verbatim}

Although action prefix, parallel, sequentiality, branch (choice) and loop consist of the cornerstones for constructing the processes of SystemC, most advantages, conveniences and plentiful properties would be missed if we stop here. Let us show the specifics.

The first thing is how to deal with delta-cycle, which involves the simulation procedure of the SystemC. The basic simulation model is (P_{1} \| ... \| P_{n}) \| env, and regard-

© 2012 NSP
Natural Sciences Publishing Cor.
less of $env$, if all processes are successfully terminated except for some delta-cycle notifications, a new delta-cycle will be advanced, e.g., $P_S$ are synchronous on action $\Delta$ and Notify enables state of $\Delta$ in $P_i$. So our method is to concatenate every $P_i$ with $\Delta, \triangledown$, as well as make Notify enable the $\Delta$ in $P_i$. Now, the simulation model would be $(P_1; \Delta, \triangledown \parallel \cdots \parallel (\Delta) P_n; \Delta, \triangledown) || env$.

Sensitivity is a very powerful and interesting thing in SystemC. Dealing with the static sensitivity list of method process is different from thread process (clock thread process), both of which involve the user defined events, Change action, system clock and a process. As for method process, when triggered, it executes from beginning to end, then returns control to the kernel, and cannot be terminated, e.g., once a sensitive event occurs, it monitors the method process instance is triggered, and then executes from beginning to end. Intuitively, if event $e_\alpha$ is in the sensitivity list of method process $P$, $\alpha$ will trigger $P$ once $\alpha$ is enabled, however, different occurrences of $e_\alpha$ may activate $P$ many times, so the way of formalizing this property of SystemC involves the process instantiation as well. Let us show the form of $P' := (t_\alpha, P_1; \tau_{\alpha, P_1}, \tau_{\alpha, \text{end}}, -P_1)$. $P'$ (with $env = \tau ((\text{Clock} \_X \_Y \_Z); P_{\text{clock}} \parallel \text{notify} (\alpha, P_0))$) represents the process $P$ with a sensitive action $\alpha$, and the conjunction of relabeling function and process instantiation makes $P$ be triggered one by one. So assume a method process $P_1$ with the sensitivity list $\{e_1, e_2, e_{\text{Change}}\}$ (section 3.1.1), say once $ud_{e_1}$ or Change happened, the process $P_1$ runs, so $P = (t_{ud_{e_1}}, 0) \parallel (t_{e_{\text{Change}}}, 0)$; $P_1; \tau_{\text{end}}, \triangledown || (t_{ud_{e_2}, \tau_{\text{end}}}, 0) || (t_{\tau_{\text{end}}}; P_1; (t_{\text{Clock} \_X \_Y \_Z}; P_{\text{clock}} \parallel \text{notify} (\alpha, P_0))$ represents the process $P$ with a sensitive action $\alpha$, and the conjunction of relabeling function and process instantiation makes $P$ be triggered one by one. So assume a method process $P_1$ with the sensitivity list $\{e_1, e_2, e_{\text{Change}}\}$ (section 3.1.1), say once $ud_{e_1}$ or Change happened, the process $P_1$ runs, so $P = (t_{ud_{e_1}}, 0) \parallel (t_{e_{\text{Change}}}, 0)$; $P_1; \tau_{\text{end}}, \triangledown || (t_{ud_{e_2}, \tau_{\text{end}}}, 0) || (t_{\tau_{\text{end}}}; P_1; (t_{\text{Clock} \_X \_Y \_Z}; P_{\text{clock}} \parallel \text{notify} (\alpha, P_0))$.

Notification, waiting and cancel statements are all involved in the user defined events, formalizations of which are shown below.

3.2.1. Notification Statements

There are three kinds of ‘notification statements’ used to enable user defined events:
1. event notify();
2. event notify(time);
3. event notify(SC_ZERO_TIME);

The above SystemC statements are related to Notify actions. Each of them enables one of the user defined actions. Line 1 shows event will be enabled immediately, line 2 shows during some time units, the event will be enabled and line 3 the event will be enabled in the next delta-cycle, so they can be modeled as:

1. $P := P_1; \{\text{Notify}.0\}; P_2$ with
   $env = \tau ((\text{Clock} \_X \_Y \_Z); P_{\text{clock}} \parallel \text{notify} (\alpha, P_0))$
2. $P := P_1; \{\text{Notify}.0\}; P_2$ with
   $env = \tau ((\text{Clock} \_X \_Y \_Z); P_{\text{clock}} \parallel \text{notify} (\alpha, P_0)$
3. $P := P_1; \{\text{Notify}.0\}; P_2$ with
   $env = \tau ((\text{Clock} \_X \_Y \_Z); P_{\text{clock}} \parallel \text{notify} (\alpha, P_0)$

with $P_1, P_2$ be the behaviors before and after the notification statement.

3.2.2. Cancel Statement

There is only one kind of ‘cancel statement’ used to reject the notification.

event cancel();

The cancel statement can be formalized as: $P := P_1; \{\tau_{\text{end}}, 0\}; P_2$

$env = \tau ((\text{Clock} \_X \_Y \_Z); P_{\text{clock}} \parallel \text{notify} (\alpha, P_0); (\text{Notify}.ud_{\alpha})$.

Note: only the timed notification (line 2 in previous figure) can be canceled.

3.2.3. Wait Statements

Formalizing the ‘wait statements’ involves Change action and user defined actions. Let us show the syntax of ‘wait statements’.

1. wait(time);
2. wait(event);
3. wait(event1 & event2);
4. wait(event1 & event2);
5. wait(time, event1) event2);
6. wait(time, event1 & event2);
7. wait();

‘Wait statements’ can be easily formalized. Intuitively, in line 5, if one of the events occurs during the time units since the start of the system or after this time unit elapses, the process would be resumed; likewise, while all of the events occur or time elapse, process is resumed in line 6. Line 1, 2, 3, 4, are similar without time constraints. The statement in line 7, represents ‘the process shall be resumed on the static sensitivity, in the absence of static sensitivity for this particular process, the process shall not be resumed again during the current simulation’ [3]. Some Change actions monitor the sensitivity list of a process, the relation between these actions is disjunct, so once any Change is enabled the process resumed. Supposing $P_1$ and $P_2$ be the formalism of statements before and after a wait statement, then each of the statements listed in above box can be formally expressed by the followings:

1. $P := P_1; \{l(t), 0\}; P_2$
2. $P := P_1; \{\tau_{ud_{\alpha}} + \text{tau}_{\text{end}}, 0\}; P_2$ with
   $env = \tau ((\text{Clock} \_X \_Y \_Z); P_{\text{clock}} \parallel \text{notify} (\alpha, P_0)$
3. $P_1: (\tau_{ud}\cdot 0 + \tau_{ud}\cdot 2.0); P_2$ with $\text{env} = \tau \circ \langle \text{Clock}_0, \text{Clock}_1 \rangle \circ \text{P}_{\text{clock}} \circ \langle \tau_{ud} \cdot 1 \rangle$ $\text{ud}_{\cdot 1}. \tau_{ud} \cdot 1.0 \circ \langle \tau_{ud} \cdot 2 \rangle \circ \text{ud}_{\cdot 2}. \tau_{ud} \cdot 2.0$

4. $P_1: (\tau_{ud}\cdot 1.0 \circ \tau_{ud}\cdot 2.0 + \tau_{\infty}\cdot 0); P_2$ with $\text{env} = \tau \circ \langle \text{Clock}_0, \text{Clock}_1 \rangle \circ \text{P}_{\text{clock}} \circ \langle \tau_{ud} \cdot 1 \rangle$ $\text{ud}_{\cdot 1}. \tau_{ud} \cdot 1.0 \circ \langle \tau_{ud} \cdot 2 \rangle \circ \text{ud}_{\cdot 2}. \tau_{ud} \cdot 2.0$

5. $P_1: (\tau_{ud}\cdot 0 \circ \tau_{ud}\cdot 2.0 + (t)\tau_{\infty}); P_2$ with $\text{env} = \tau \circ \langle \text{Clock}_0, \text{Clock}_1 \rangle \circ \text{P}_{\text{clock}} \circ \langle \tau_{ud} \cdot 1 \rangle$ $\text{ud}_{\cdot 1}. \tau_{ud} \cdot 1.0 \circ \langle \tau_{ud} \cdot 2 \rangle \circ \text{ud}_{\cdot 2}. \tau_{ud} \cdot 2.0$

6. $P_1: (\tau_{ud}\cdot 1.0 \circ \tau_{ud}\cdot 2.0 + (t)\tau_{\infty}); P_2$ with $\text{env} = \tau \circ \langle \text{Clock}_0, \text{Clock}_1 \rangle \circ \text{P}_{\text{clock}} \circ \langle \tau_{ud} \cdot 1 \rangle$ $\text{ud}_{\cdot 1}. \tau_{ud} \cdot 1.0 \circ \langle \tau_{ud} \cdot 2 \rangle \circ \text{ud}_{\cdot 2}. \tau_{ud} \cdot 2.0$

7. $P_1: (\tau_{\text{change}}\cdot 0 + \ldots + \tau_{\text{change}}\cdot m); P_2$ with $\text{env} = \tau \circ \langle \text{Clock}_0, \text{Clock}_1 \rangle \circ \text{P}_{\text{clock}} \circ \langle \tau_{\text{change}} \cdot 1 \rangle \circ \text{change}_{1, \ldots, \tau_{\text{change}}\cdot m}$ $\text{change}_{1, \ldots, \tau_{\text{change}}\cdot m}$

for $\tau_{\text{change}}$, monitor the sensitivity list (if the positive or negative clock wedge is monitored, $\tau_{\text{Clock}} \cdot \gamma$ may be needed).

4. Case study

We demonstrate how to use event structure to represent and deduce SystemC behaviors by the following case study. These codes were first seen in [9]. Let us show the SystemC codes (See Figure 5, 6).

Actually we have shown (the part of) the formal specification of above codes in section 3.2 here the complete version can be: $P := \tau_{\text{Clock}} \cdot P_1: \tau_{\text{end}} \cdot \vee (\tau_{\text{clock}} \cdot \tau_{\text{end}} \cdot P)$ with $P_1 := (\eta, \alpha_1, 0 + \alpha_2, 0); \alpha_3 \cdot \vee$

$\text{env}_P = \tau \circ \langle \text{Clock}_0 \rangle \circ \text{Clock}_1 \cdot \text{P}_{\text{clock}} \circ \langle \alpha_1 \rangle \circ \langle \alpha_2 \cdot 0 \rangle \circ \text{H}_1 \circ \langle \alpha_1 \rangle \circ \langle \alpha_3 \cdot 0 \rangle \circ \text{H}_2 \circ \langle \eta, 0 \rangle \circ \ldots$

Figure 6 count.cc

Figure 7 Event structure of count

that

$H_i(\text{Test}) = \text{Change}_{\text{count}} \cdot \vee \text{count}_{\cdot 1} \cdot \gamma_i \cdot 0 \neq \text{A}(\text{Test} \cdot 0) \wedge \text{same}_{\cdot i} \cdot (\gamma_i, \text{Test}) \wedge \text{Test} \in \text{A}(\text{env})$ with $i = 1, 2, 3$

Let $\alpha_1, \alpha_2, \gamma_1, \gamma_2, \gamma_3$ and $\eta$ be alias for $\text{Assign}_{\text{count}_{\cdot 1}}, \text{Assign}_{\text{count}_{\cdot 1}} = \text{count}_{\cdot 1} \cdot \eta$, $\text{Assign}_{\text{count}_{\cdot 1}} = \text{count}_{\cdot 1} \cdot \eta$, $\text{Test}_{\text{count}} \cdot \text{count}_{\cdot 1} \cdot \eta$, $\text{Test}_{\text{count}} \cdot \text{count}_{\cdot 1} \cdot \eta$, $\text{Test}_{\text{load}} = \text{true}$.

Similar to section 2.2, denotational semantics of $P$ may be depicted by a figure. Moreover, $P$ contains the process instantiation operator, which makes the figure much more complex. We describe $P$ in Figure 7, but the basic version, e.g., $\mathcal{F}_p(\bot)$ for $P$, can be found in Figure 3.

Unabridged SystemC codes not only involve the system descriptions (like in Figure 5 and Figure 6), but also the test-bench. Figure 8 and Figure 9 show test-bench codes of the above descriptions.

The formal specification of the above test-bench part is $Q = \tau_{\text{clock}} \cdot Q_1$ (pictured in Figure 10, (a) for $\mathcal{F}_p(\bot)$ and (b) for $Q$) with $Q_1 = \epsilon_1 \cdot 0; \epsilon_2 \cdot 0; (\text{Clock}_{\cdot 0} + (\infty) \tau_{\infty} \cdot 0); \epsilon_3 \cdot 0; (\text{Clock}_{\cdot 0} + (\infty) \tau_{\infty} \cdot 0); Q_1;$, and the environment, $\text{env}_Q = \tau \circ \langle \text{Clock}_0 \rangle \circ \text{Clock}_1 \cdot \text{P}_{\text{clock}} \circ \langle \alpha_1 \rangle \circ \langle \alpha_2 \cdot 0 \rangle \circ \text{H}_1 \circ \langle \alpha_1 \rangle \circ \langle \alpha_3 \cdot 0 \rangle \circ \text{H}_2 \circ \langle \eta, 0 \rangle \circ \ldots$

$\text{env}_Q = \tau \circ \langle \epsilon_1 \rangle \circ \epsilon_1 \cdot 0 \circ \text{H}_1 \circ \langle \epsilon_2 \rangle \circ \epsilon_2 \cdot 0 \circ \text{H}_2 \circ \langle \epsilon_3 \rangle$
/* count_sim.h */
2SC_MODULE(count_sim) {
3 se_out <bool> load;
4 se_out <int> din;
5 se_in <bool> clock;
6 se_out <int> dout;
7 void stimgen();
8 SCCTOR(count_sim) {
9 SC_THREAD(stigen);
10 sensitive_post(clock);
11 }
12 };

Figure 8 count_sim.h

/* count_sim.cc */
1 #include "systemc.h"
2 #include "count_sim.h"
3 void count_sim(stimel) {
4 while(true) {
5 load=true;
6 din=0;
7 wait();
8 load=false;
9 wait();
10 }
11 };

Figure 9 count_sim.cc

Figure 10 Event structure of count_sim

\[ \varepsilon, \eta_3, 0[H_3] \mid \cdots, \text{and} \]

\( H_i(\text{Test}) = \text{Change}_{\text{count.out}} \land A(\eta_i, 0) \neq A(\text{Test}.0) \land \text{same.c}(\eta_i, \text{Test}) \land \text{Test} \in A(\text{env}) \) with \( i = 1, 2 \)

\( H_3(\text{Test}) = \text{Change}_{\text{count.out}} \land A(\eta_3, 0) \neq A(\text{Test}.0) \land \text{same.c}(\eta_3, \text{Test}) \land \text{Test} \in A(\text{env}) \)

\( H_3(\eta_3) = \tau \land A(\eta_3, 0) = A(\text{Test}.0) \land \text{same.c}(\eta_3, \text{Test}) \land \text{Test} \in A(\text{env}) \) with \( i = 1, 2, 3 \)

(Let \( \varepsilon_1, \varepsilon_2 \) and \( \eta_1, \eta_2 \) be aliases for \( \text{Assign.load}=\text{true}, \text{Assign.din}=0, \text{Assign.load}=\text{false} \) and \( \text{Test.load}=\text{true}, \text{Test.din}=0, \text{Test.load}=\text{false}. \))

Then the formalization for the whole codes will be

\( B := P_1 \land \sqrt{\Delta_1} \land Q \land \Delta_1 \land \sqrt{\text{env}} \) with \( \text{env} \) equals to \( P \land \Delta \land \sqrt{\text{env}} \). Both \( P \) and \( Q \)

\( \tau \parallel (\text{Clock}_T \land \text{Clock}_L) \parallel P_{\text{lock}} \parallel env_P \parallel env_Q \) osp, which do not distinguish processes and simulation environment.

Figure 11 Event Structure of the case-study

Here we have introduced both the denotational semantics and operational one of the case study, we do this in a unified way, i.e., formalizing the codes and showing a uniform denotational semantics and operational semantics which do not distinguish processes and simulation environment.
5. Conclusion

In this article, we have introduced the way of rewriting SystemC statements by process algebra and a unified formal framework of simulation, and the event structure based denotational, operational semantics and the correspondence between them. At the end of this paper, we presented a case study to show a way to model real SystemC codes reasonably.

The work about the formalization of SystemC does not end. Our future research will focus on two valuable issues:

1. the action refinement for the top-down design;
2. the (event structure based) verification of unified process and environment.

Acknowledgement

The authors would like to thank to the NSF of China under Grant No. 60873118, 60973147, 61073108, 90812001, 61073193, 61073168, 61133016 and 90912003, the 973 Program of China (Grant No. 2010CB328000), the NSF of Guangxi under Grant No. 2011-GXNSFA018154, the Doctoral Fund of Ministry of Education of China under Grant No. 2009009110006 the Science and Technology Foundation of Guangxi under Grant No. 10169-1, and Guangxi Scientific Research Project No. 201012MS274, Grants (HCIC-201110) of Guangxi Key Laboratory of Hybrid Computational and IC Design Analysis Open Fund, the Key science and technology Foundation of Gansu Province (110-2FKDA010), Natural Science Foundation of Gansu Province (1107RJZA188).

References