Applied Mathematics & Information Sciences
An International Journal

A Mathematical Characterization of System Design and Modeling

Xibin Zhao[1], **Anping He*[2,3], *Jinzhao Wu*[2,4], *Guowu Yang*[5], *Yi Yang*[3], *Ning Zhou*[4], *Shihan Yang*[2], *Lian Li*[2]

[1] Key Laboratory for Information System Security of Ministry of Education, School of Software, Tsinghua University, China.
[2] Guangxi Key laboratory of Hybrid Computational and IC Design Analysis, Naning, China.
[3] School of Information Science and Engineering, Lanzhou University, China.
[4] School of Computer and Information Technology, Beijing Jiaotong University, China.
[5] School 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 (*HDL*s) are typically used to express designs at various levels of abstraction. *HDL*s 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 *HDL*s 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 *HDL*s, 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$^{\mathbb{PA}}$ [14], which is similar to K.L. Man's SystemC$^{\mathbb{FL}}$ [9,15,16], but Man's article focuses on SystemC$^{\mathbb{FL}}$ 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 $Assign$, $Test$, $Change$, $Notify$, $Clock_{\top}$ and $Clock_{\bot}$ 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., $\mathcal{D}$ and $\mathcal{A}$, 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 $\mathcal{E} = (E, \sharp, \vdash)$, where E is a countable set of events; $\sharp \subseteq E \times E$ is

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

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 be not 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 $\widetilde{\mathbb{N}} = \mathbb{N} \cup \{\infty\}$, where $\infty > n$ for any $n \in \mathbb{N}$. We use $CF(S)$ to denote that all the events in set S is conflict-free.

$$CF(S) = \forall e, e' \in S : (e \neq e') \Rightarrow \neg(e \sharp e')$$

and $CFL(S)$ to the set of events that conflict with S:

$$CFL(S) = \{e \mid e \in E \wedge e' \in S \wedge \neg CF(\{e, e'\})\}$$

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

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

such that

1. $\forall X \subseteq E, e \in E : X \vdash e \Rightarrow CF(X \cup \{e\})$*;*
2. $\forall X, Y \subseteq E, e \in E : (X \vdash e \wedge Y \vdash e) \Rightarrow (\neg CF(X \cup Y) \vee X = Y)$.

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

The event structure $\mathcal{TE} = (E, \sharp, \vdash, l, \mathcal{D}, \mathcal{T})$ has *correct timing* iff for any $e \in E$ and $X \vdash e$, $\mathcal{T}(X) \leq \mathcal{D}(e)$. 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 $(E, \sharp, \vdash, l, \mathcal{D}, \mathcal{T})$ with $E = \{e_\alpha, e_\beta, e_\eta, e_\gamma\}$, $\sharp = \{(e_\beta, e_\eta)\}$, $\vdash = \{(\{e_\alpha\}, e_\beta), (\{e_\alpha\}, e_\eta), (\{e_\eta\}, e_\gamma)\}$, $l = \{(e_\alpha, \alpha), (e_\beta, \beta), (e_\eta, \eta), (e_\gamma, \gamma)\}$, $\mathcal{D} = \{(e_\alpha, 3), (e_\beta, 14), (e_\gamma, 11)\}$ and $\mathcal{T} = \emptyset$.

Behaviors of system specified by a $\mathcal{TE}$ 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 $\sigma = (e_1, t_1) \cdots (e_n, t_n)$, where $t_1 \leq t_2 \leq \cdots \leq t_n$, with (e_i, t_i) showing e_i occurred at time t_i. Let $[\sigma]$ denote the sequence of events in σ without time, i.e., $[\sigma] = e_1 \cdots e_n$, $\overline{[\sigma]}$ the set of events in $[\sigma]$ and $\overline{\sigma}$ the set of timed events. We use σ_i to denote a trace of length i, i.e. $\sigma_i = (e_1, t_1) \cdots (e_i, t_i)$. 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: $en([\sigma]) = \{e \mid e \notin CFL(\overline{[\sigma]}) \wedge e \notin \overline{[\sigma]} \wedge (\exists X \subseteq \overline{[\sigma]} : X \vdash e)\}$. Let $time(\sigma, e)$ denote the minimal time instant from which e can occur, two aspects should be considered: *(i)* e's absolute delay $\mathcal{D}(e)$, *(ii)* the time relative to all e's causality, so $time(\sigma_i, e) = Max(\{\mathcal{D}(e)\} \cup H\})$ $with$ $H = \{t + t_i | \exists X \subseteq \overline{[\sigma]} : \mathcal{T}(X, e) = t\}$. Let $[\sigma_{i-1}] = e_1 \cdots e_{i-1}$ be the $(i-1)$-th prefix of $[\sigma]$, so the definition of trace is as following:

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

- $e_1 \cdots e_n$ *with* $e_i \in en([\sigma_{i-1}])$*;*
- $\forall i : (e_i) \Rightarrow (t_i \geqslant time(\sigma_i, e_i))$*;*
- $\forall i, j : (i < j) \Rightarrow (t_i \leqslant t_j)$.

A state of an execution of a timed event structure $\mathcal{TE}$ is called a *timed configuration* $TC = \overline{\sigma}$. Let $Conf(\mathcal{TE})$ denote the set of timed configurations of $\mathcal{TE}$. Say that there is a transition from a timed configuration TC to a timed configuration TC' and write $TC \to TC'$ iff $TC \subseteq TC'$. It is easy to see that $\to$ includes a partial order on $Conf(\mathcal{TE})$. Moreover, the class of this structure is denoted by 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 canceling, notification overriding, event waiting, time delaying, sensitivity lists, concurrent processes and delta-cycle. The syntax of (timed)

process algebra is as following:

$$P ::= \mathbf{0} \mid \surd \mid (t)\alpha.P_1 \mid P_1 + P_2 \mid P_1; P_2 \mid P_1 \parallel_G P_2 \mid P[H] \mid B \quad (1)$$

A set of event based inference rules are depicted in *Figure* 1, which is also called a *structural operational semantics* (SOS). $P_1 \xrightarrow{(e,\alpha,t)} P_1'$, denotes that at time t, behavior P_1 can perform an event e labeled with action α, and subsequently evolve into P_1'. Note an auxiliary construct ${}^{t'}[P]$ (also used in [5]) shows if P performs event ξ at time t, then ${}^{t'}[P]$ performs ξ at time $t + t'$.

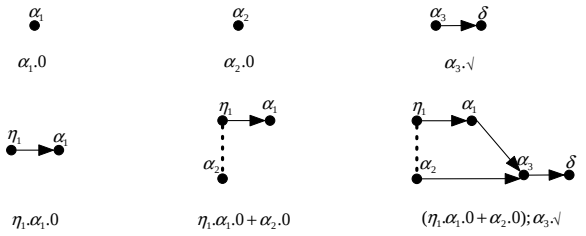

Figure 2 Examples of semantics for *action-prefix*, *sequential* and *choice*

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 $init(\mathcal{TE})$ denote the set of initial events of $\mathcal{TE}$, $exit(\mathcal{TE})$ the set of successful termination events, $pos(\mathcal{TE})$ that events with non-zero delay: $init(\mathcal{TE}) \triangleq \{e \in E \mid \neg(\exists X \subseteq E : X \vdash e)\}$, $exit(\mathcal{TE}) \triangleq \{e \in E \mid l(e) = \delta\}$ and $pos(\mathcal{TE}) \triangleq \{e \in E \mid \mathcal{D}(e) \neq 0\}$. Let $pin(\mathcal{TE}) \triangleq pos(\mathcal{TE}) \cup init(\mathcal{TE})$. Function $\mathcal{E}[\![\ ]\!]$ (also adopted in [5]) associates each process term $P_i \in P$ with an element of SES, e.g., $\mathcal{E}[\![\ ]\!] : P \mapsto SES$. Then the causality based denotational semantics can be defined with $\mathcal{E}[\![\ ]\!]$ respectively. Firstly, let $\mathcal{E}[\![P_i]\!] = \mathcal{TE}_i = \langle E_i, \sharp_i, \vdash_i, l_i, \mathcal{D}_i, \mathcal{T}_i\rangle$, 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 ($\surd$) and relabeling (H) are self-explanatory:

$\mathcal{E}[\![\mathbf{0}]\!] \triangleq \langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset\rangle$.

$\mathcal{E}[\![\surd]\!] \triangleq \langle E, \emptyset, \emptyset, l, \mathcal{D}, \emptyset\rangle$

that $E = \{e_\delta\}$, $l = \{(e_\delta, \delta)\}$, and $\mathcal{D} = \{(e_\delta, 0)\}$. And

$\mathcal{E}[\![P[H]]\!] \triangleq \langle E, \sharp, \vdash, H \circ l, \mathcal{D}, \mathcal{T}\rangle$

'$\circ$' is operator for composition of functions)

2. The semantics of action-prefix ($(t)\alpha.$)

For action prefix, an enabling relation is introduced from a new event e_α to all initial events in $\mathcal{TE}_1$:

$\mathcal{E}[\![(t)\alpha.P_1]\!] \triangleq \langle E, \sharp_1, \vdash, l_1 \cup \{(e_\alpha, \alpha)\}, \mathcal{D}, \mathcal{T}\rangle$

with:

$E = E_1 \cup \{e_\alpha\}$ for $e_\alpha \notin E$,
$\vdash = \vdash_1 \cup(\{\{e_\alpha\}\} \times pin(\mathcal{TE}_\infty))$,
$\mathcal{D} = \{(e_\alpha, t)\} \cup (E_1 \times \{0\})$,
$\mathcal{T} = \mathcal{T}_1 \cup \{((\{e_\alpha\}, e), \mathcal{D}_1(e)) | e \in pin(\mathcal{TE}_1)\}$.

3. The semantics of Choice ($+$)

$\mathcal{E}[\![P_1 + P_2]\!] \triangleq \langle E_1 \cup E_2, \sharp, \vdash_1 \cup \vdash_2, l_1 \cup l_2, \mathcal{D}_1 \cup \mathcal{D}_2, \mathcal{T}_1 \cup \mathcal{T}_2\rangle$

with:

$\sharp = \sharp_1 \cup \sharp_2 \cup \{init(\mathcal{TE}_1) \times init(\mathcal{TE}_2)\}$

Note: The choice between processes is resolved in interaction with the environment.

4. The semantics of sequential (;)

$\mathcal{E}[\![P_1; P_2]\!] \triangleq \langle E_1 \cup E_2, \sharp, \vdash, l, \mathcal{D}, \mathcal{T}\rangle$

with:

$\sharp = \sharp_1 \cup \sharp_2 \cup \{ (e, e') \mid e, e' \in exit(\mathcal{TE}_1) \wedge e_1 \neq e_2 \}$
$\vdash = \vdash_1 \cup \vdash_2 \cup(\{exit(\mathcal{TE}_1)\} \times \{pin(\mathcal{TE}_2)\})$
$l = (l_1 \backslash (exit(\mathcal{TE}_1) \times \{\delta\})) \cup l_2 \cup (exit(\mathcal{TE}_1) \times \{\tau\})$
$\mathcal{D} = \mathcal{D}_1 \cup (E_2 \times \{0\})$
$\mathcal{T} = \mathcal{T}_1 \cup \mathcal{T}_2 \cup \{((exit(\mathcal{TE}), e), \mathcal{D}_2(e)) | e \in pin(\mathcal{TE}_2)\}$

$\mathcal{E}[\![P_1; P_2]\!]$ is sequential operate. It equals to $\mathcal{TE}_1 \cup \mathcal{TE}_2$ where enable relations are introduced from the successful termination events of $\mathcal{TE}_1$ to the initial events of $\mathcal{TE}_2$.

5. The semantics of parallel ($\parallel_G$)

This operator shows the behavior of communicated processes, let set of synchronization events be $G \subseteq \mathcal{A}$, $E_i^s \triangleq \{ e \in E_i \mid l_i(e) \in G^\delta\}$ and non-synchronizing events $E_i^f \triangleq E_i \backslash E_i^s$:

$\mathcal{E}[\![P_1 \parallel_G P_2]\!] \triangleq \langle E, \sharp, \vdash, l, \mathcal{D}, \mathcal{T}\rangle$

that:

$E = (E_1^f \times \{*\}) \cup (\{*\} \times E_2^f) \cup \{ (e_1, e_2) \in E_1^s \times E_2^s \mid l_1(e_1) = l_2(e_2) \}$
$(e_1, e_2)\sharp(e_1', e_2')$ with
$(e_1 \sharp e_1') \vee (e_2 \sharp e_2') \vee (e_1 = e_1' \neq * \wedge e_2 \neq e_2') \vee (e_2 = e_2' \neq * \wedge e_1 \neq e_1')$
$X \vdash (e_1, e_2)$ with
$X = \{(e, *) | X_1 \vdash_1 e_1 \wedge e \in X_1 \wedge e \notin E_1^s\} \cup \{(*, e') | X_2 \vdash_2 e_2 \wedge e' \in X_2 \wedge e' \notin E_2^s\} \cup \{(e, e') | X_1 \vdash_1 e \wedge X_2 \vdash_2 e' \wedge e \in E_1^s \wedge e' \in E_2^s \wedge e \in X_1 \wedge e' \in X_2\}$

$$l(e_1, e_2) = \begin{cases} l_2(e_2) & if\ e_1 = *; \\ l_1(e_1) & others. \end{cases}$$

$\mathcal{D}((e_1, e_2)) = max(\mathcal{D}_1(e_1), \mathcal{D}_2(e_2))$ with $\mathcal{D}_i(*) = 0$
$\mathcal{T}(X, (e_1, e_2)) = max(\mathcal{T}_1(X_1, e_1), \mathcal{T}_2(X_2, e_2))$
with $X_1 \subseteq E_1 \wedge X_1 \vdash_1 e_1 \wedge X_2 \subseteq E_2 \wedge X_2 \vdash_2 e_2$

1. Successful terminated

$$\frac{}{\surd_{\xi} \xrightarrow{(\xi,\delta,t)} \mathbf{0}}$$

2. Action prefix

$$\frac{}{(t)\alpha_{\xi}.P_1 \xrightarrow{(\xi,\alpha,t')} {}^{t'}[P_1]} \quad (t' \geq t)$$

3. Alternative

$$3.1\ \frac{P_1 \xrightarrow{(\xi,\alpha,t)} P_1'}{P_1 + P_2 \xrightarrow{(\xi,\alpha,t)} P_1'} \qquad 3.2\ \frac{P_2 \xrightarrow{(\xi,\alpha,t)} P_2'}{P_1 + P_2 \xrightarrow{(\xi,\alpha,t)} P_2'}$$

4. Sequential

$$4.1\ \frac{P_1 \xrightarrow{(\xi,\alpha,t)} P_1'}{P_1; P_2 \xrightarrow{(\xi,\alpha,t)} P_1'; P_2} \quad (\alpha \neq \delta) \qquad 4.2\ \frac{P_2 \xrightarrow{(\xi,\alpha,t)} P_2'}{P_1; P_2 \xrightarrow{(\xi,\alpha,t)} P_2} \quad (\alpha = \delta)$$

5. Parallelism

$$5.1\ \frac{P_1 \xrightarrow{(\xi,\alpha,t)} P_1'}{P_1 \parallel_G P_2 \xrightarrow{((\xi,*),\alpha,t)} P_1' \parallel_G P_2} \quad (\alpha \notin G^{\delta}) \qquad 5.2\ \frac{P_2 \xrightarrow{(\xi,\alpha,t)} P_2'}{P_1 \parallel_G P_2 \xrightarrow{((*,\xi),\alpha,t)} P_1 \parallel_G P_2'} \quad (\alpha \notin G^{\delta})$$

$$5.3\ \frac{P_1 \xrightarrow{(\xi,\alpha,t)} P_1' \wedge P_2 \xrightarrow{(\phi,\alpha,t)} P_2'}{P_1 \parallel_G P_2 \xrightarrow{((\xi,\phi),\alpha,t)} P_1' \parallel_G P_2'} \quad (\alpha \in G^{\delta})$$

6. Relabeling

$$\frac{P \xrightarrow{(\xi,\alpha,t)} P'}{P[H] \xrightarrow{(\xi,H(\alpha),t)} P'[H]}$$

7. Process instantiation

$$7.1\ \frac{B \xrightarrow{(\xi,\alpha,t)} B'}{P_{\phi} \xrightarrow{(\phi\xi,\alpha,t)} \phi(B')} \quad (P := B) \qquad 7.2\ \frac{B \xrightarrow{(\xi,\alpha,t)} B'}{\phi(B) \xrightarrow{(\phi\xi,\alpha,t)} \phi(B')}$$

Figure 1 Event based operational semantics for Process Algebra

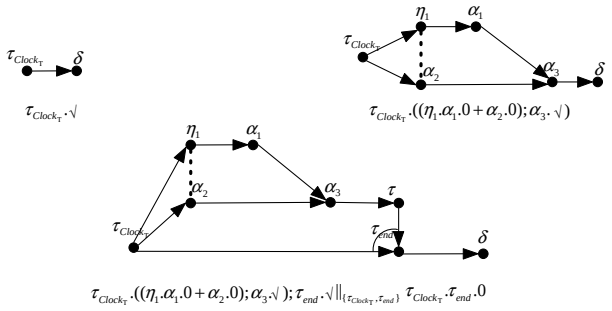

Figure 3 Examples of semantics for *parallel* and *relabeling*

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 $\mathcal{TE} = \mathcal{F}_B(\mathcal{TE})$. Let us see partial order relation on event structures, which is a time extension version of the one from Winskel [4]:

Definition 3.*Let* $\mathcal{TE}_i = \langle E_i, \sharp_i, \vdash_i, l_i, \mathcal{D}_i, \mathcal{T}_i \rangle$*, then* $\mathcal{TE} \trianglelefteq \mathcal{TE}_2$ *iff*

$$\begin{aligned}
&E_1 \subseteq E_2,\\
&\sharp_1 = \sharp_2 \cap (E_1 \times E_1),\\
&\vdash_1 = \{(X \cap E_1, e) \mid e \in E_1 \cap X \vdash_2 e\},\\
&l_1 = l_2 \upharpoonright E_1,\\
&\mathcal{D}_1 = \mathcal{D}_2 \upharpoonright E_1,\\
&\forall e \in E_1, \mathcal{T}_2(X, e) = \mathcal{T}_1(X \cap E_1, e).
\end{aligned}$$

Let $\mathcal{TE}_1 \trianglelefteq \mathcal{TE}_2 \trianglelefteq \cdots \trianglelefteq \mathcal{TE}_n$ be a partial order of $\langle SES, \trianglelefteq \rangle$, and $\mathcal{TE}_1', \mathcal{TE}_2', \cdots \in SES$ are upper bounds of the order. We can construct $\mathcal{TE} = \langle E, \sharp, \vdash, l, \mathcal{D}, \mathcal{T} \rangle$ to be the *l.u.b.* (least upper bound) of the chain $\mathcal{TE} \trianglelefteq \cdots \trianglelefteq \mathcal{TE}_n$, e.g., $E = (E_1' \cap E_2' \cap \cdots) \cup E_n$, $\sharp = (\sharp_1' \cap (E \times E_1')) \cup \cdots$, $l = (l_1' \upharpoonright E) \cup \cdots$, $\mathcal{D} = (\mathcal{D}_1' \upharpoonright E) \cup \cdots$ and $\mathcal{T} = (\mathcal{T}_1' \upharpoonright E) \cup \cdots$. According to the definition 3, $\mathcal{TE} \trianglelefteq \mathcal{TE}_1', \cdots$ and $\mathcal{TE}, \cdots \trianglelefteq \mathcal{TE}$. So $\langle SES, \trianglelefteq \rangle$ is a *c.p.o* (complete partial order). Moreover, it is easy to know that **0**, e.g., $\langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle$, is the least element under $\trianglelefteq$. So this partial order is a *pointed c.p.o.* [19].

Then let $\sqcup_i \mathcal{TE}_i$ be the least upper bound under this partial order, $\mathcal{TE}_1 \trianglelefteq \mathcal{TE}_2 \trianglelefteq \cdots \trianglelefteq \mathcal{TE}_n$ be a chain, the least upper bound (under $\trianglelefteq$) is as following:

$$\sqcup_i \mathcal{TE}_i \triangleq (\bigcup_i E_i, \bigcup_i \sharp_i, \vdash, \bigcup_i l_i, \bigcup_i \mathcal{D}_i, \mathcal{T})$$

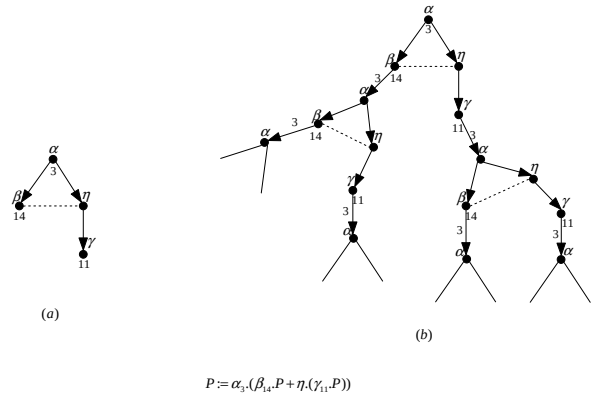

Figure 4 An example of semantics for $instantiation$

with:

$$\vdash = \{(\bigcup_j X_j, e) \mid \exists k : \forall j \geq k : X_j \vdash_j e \wedge X_{j+1} \cap E_j = X_j\}$$

$$\mathcal{T} = \{((\bigcup_j X_j, e), t) \mid \exists k : \forall j \geq k : \mathcal{T}_j(X_j, e) = t \wedge X_{j+1} \cap E_j = X_j\}$$

Let $\mathcal{F}$ be a function over SES, see the definition of continuity of $\mathcal{F}$:

Definition 4.*Let* $\langle SES, \trianglelefteq \rangle$ *be a* pointed c.p.o. *and* $\mathcal{F} : SES \rightarrow SES$. $\mathcal{F}$ *is* continuous *iff* $\mathcal{F}$ *is* monotonic *and for any chain* $\mathcal{TE} \trianglelefteq \mathcal{TE}_2 \trianglelefteq \cdots$, *we have* $E(\mathcal{F}(\sqcup_i \mathcal{TE}_i)) \subseteq E(\sqcup_i \mathcal{F}(\mathcal{TE}_i))$.

Then let us investigate the process operators $.,;,+,[]$. Firstly, we distinguish all action prefix and $\surd$ occurrences by subscription with a Greek letter, let the operator of subscription be ϕ. For instance, $P := (t)\alpha.P + \beta.P$ becomes $\phi(P) := (t)\alpha_\xi.P_\phi + \beta_\chi.P_\psi$. And secondly, we denote for operator $op \in PA$ the corresponding counterpart on event structures by $\overline{op}$.

Lemma 1.$\overline{\alpha_\xi .}, \overline{+}, \overline{\|_G}, \overline{;}, \overline{[]}$ *and* $\overline{\phi()}$ *are* continuous *on* $\langle SES, \trianglelefteq \rangle$.

Winskel has proven that $.,;,+,[],\|$ 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 $\mathcal{F}_B$ be a function over SES with all process operators introduced before (like $.,;,+,\ldots$), then the semantics of process instantiation is as following:

Definition 5.*The semantics of* $P := B$ *is* $\mathcal{E}[\![P]\!] \triangleq \sqcup_i \mathcal{F}_B^i(\perp)$.

In *Figure* 4, (a) shows the event structures for $\mathcal{F}_B(\perp)$ and (b) depicts $\mathcal{E}[\![B]\!]$.

2.3. Correspondence With Semantics

It is necessary to prove the consistency between the denotational semantics over event structures and it's 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 $\mathcal{A}^{\delta,\tau}$ 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* $(Conf(\mathcal{TE}), \rightarrow, L)$, *that*

$P \xrightarrow{(e,\alpha,t)} P'$ *with*
$e \in Ev \wedge P, P' \in Conf(\mathcal{TE}) \wedge \alpha \in \mathcal{A} \wedge t \in \widetilde{\mathbb{N}}$,
a transition *relation*;
$L = \{(e,t) \mid \exists \alpha \in \mathcal{A} \wedge P, P' \in Conf(\mathcal{TE}) \wedge P \xrightarrow{(e,\alpha,t)} P'\}$, 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 $\mathcal{E}[\![P]\!]$. According to definition 6, an event based transition system from $\mathcal{E}[\![P]\!]$ could be constructed, e.g., $ETS(\mathcal{E}[\![P]\!])$. These two systems are bisimular:

Lemma 2.$\forall B \in System^{PA}$, $TS(B)$ *and* $ETS(\mathcal{E}[\![B]\!])$ *are bisimular.*

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$^{\mathbb{PA}}$ [14], which is similar to K.L. Man's SystemC$^{\mathbb{FL}}$ [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.

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 τ, performing the invisible action doing nothing; *(2) non-user defined* actions, denoted by $\mathcal{A}_{NU}$, including $Assign$, $Test$, $Change$, $Notify$, Δ, $Clock_\perp$ and $Clock_\top$; *(3) user defined actions*, denoted by $\mathcal{A}_U$, including all actions enabling the user defined events; and (4) a special action δ indicates the *successful termination* of a process. Let $\mathcal{A} = \mathcal{A}_{NU} \cup \mathcal{A}_U$.

The precedences of the composition operators are, in descending order of their binding strength: $\{.,;,+,\|,[]\}$. 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 $same_v$ to indicate that the events ($Assign$, $Test$ or $Change$) deal with the same variables:

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

with $\alpha_1, \alpha_2 \in \{Assign, Test, Change\}$.

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 actions 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.

```
...
    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 clocked 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 Δ action occurs while one delta-cycle finish; and the $Clock_\top$ and $Clock_\perp$ 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, $Assign$, $Change$, $Test$, $Clock_\top$ and $Clock_\perp$ are relevant: for a concrete variable, once one $Assign$ happens, the corresponding $Test$ is enabled, occurence of two different $Test$ actions lead to a $Change$ be enabled (for monitoring sensitivity list). Moreover, $Notify$ is relevant to Δ (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. δ and τ

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, τ, is from a lower abstraction of system, which is silent and invisible to the higher researched level. In the lower level, τ may be plentiful, but system will always perform some kinds of invisible actions before halted. In this article, τ is only a placeholder in most cases if not specially described. Moreover, for the clarity and vividness, we sometimes add a suffix for τ, for example τ_∞ which represents τ being enabled after infinite time delay.

Similar to [5], δ 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

[2] $H : \mathcal{A} \cup \{\tau, \delta\} \to \mathcal{A} \cup \{\tau, \delta\}$ a relabeling function that satisfies $H(\tau) = \tau$, $H(\delta) = \delta$ and for $\alpha \in \mathcal{A} : H(\alpha) \neq \tau$ and $H(\alpha) \neq \delta$.

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 $(t)\alpha.B$ denotes, after t time units since the start of the system, a behavior which may engage in α and after which it behaves like P. Then the simplest process may be $(t)\alpha.\surd$ 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 $P_{clock} := Clock_{\top}.Clock_{\bot}.P_{clock}$.

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 similarities, 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 of 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 $P_1;(Assign.\surd)$, 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 $Test$s could enable the $Change$ for the same variable, *relabeling* operator should be applied, so a complete form is $P := P_1; Assign.\surd$ and $env = \tau \parallel_{\{Clock_{\top},Clock_{\bot}\}} P_{clock} \parallel_{\{Assign\}} (Assign.Test.\surd[H])$, that $H(Test') = Change \wedge \mathcal{A}(Test.\mathbf{0}) \neq \mathcal{A}(Test'.\mathbf{0}) \wedge same_v(Test, Test') \wedge (Test' \in \mathcal{A}(env))$ and $H(Test) = \tau$, $\mathcal{A}(Test.\mathbf{0}) = \mathcal{A}(Test'.\mathbf{0})$, and $same_v(Test, Test'), (Test' \in \mathcal{A}(env))$.

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:

```
P1 ···
    if (load) {
            count_val=din;
    } else{
            count_val=count_val+1;
    }
        dout=count_val;
P2 ···
```

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: $P := P_1;(\eta.\alpha_1.\mathbf{0} + \alpha_2.\mathbf{0}); \alpha_3.\mathbf{0}; P_2$ and $env = \tau \parallel_{\{Clock_{\top},Clock_{\bot}\}} P_{clock} \parallel env_{p_1} \parallel_{\{\alpha_1\}} (\alpha_1.\gamma_1.\mathbf{0}[H_1]) \parallel_{\{\alpha_2\}} (\alpha_2.\gamma_2.\mathbf{0}[H_2]) \parallel_{\{\alpha_3\}} (\alpha_3.\gamma_3.\mathbf{0}[H_3]) \parallel_{\{\eta\}} (\zeta.\eta.\mathbf{0}) \parallel env_{P_2}$, that

$$H_i(Test) = Change_{count\_val} \wedge \mathcal{A}(\gamma_i.\mathbf{0}) \neq \mathcal{A}(Test.\mathbf{0}) \wedge same_v(\gamma_i, Test) \wedge Test \in \mathcal{A}(env) \text{ with } i = 1, 2$$
$$H_3(Test) = Change_{dout} \wedge \mathcal{A}(\gamma_3.\mathbf{0}) \neq \mathcal{A}(Test.\mathbf{0}) \wedge same_v(\gamma_3, Test) \wedge Test \in \mathcal{A}(env)$$
$$H_i(\gamma_i) = \tau \wedge \mathcal{A}(\gamma_i.\mathbf{0}) = \mathcal{A}(Test.\mathbf{0}) \wedge same_v(\gamma_i, Test) \wedge (Test \in \mathcal{A}(env)) \text{ with } i = 1, 2, 3.And$$

α_1, α_2, α_3, ζ, γ_1, γ_2, γ_3 and η be aliases for $Assign_{count\_val=din}$, $Assign_{count\_val=count\_var+1}$, $Assign_{dout=count}$, $Assign_{load=true}$, $Test_{count\_val=din}$, $Test_{count\_val=count\_var+1}$, $Test_{dout=count}$, $Test_{load=true}$ separately.

Again, let us investigate the code for loop structure:

```
P1 ···
        while(true) {
                load=true;
                din=0;
                P2 ···
                load=false;
                P3 ···
        }
P4 ···
```

In general, the loop repeats some behaviors until a specific condition is not satisfied, for which *Process instantiation* would be used. Because the P_4 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 $P' = \epsilon_1.\mathbf{0}; \epsilon_2.\mathbf{0}; P_2; \epsilon_3.\mathbf{0}; P_3; P'$. And the environment, $env = \tau \parallel_{\{Clock_{\top},Clock_{\bot}\}} P_{clock} \parallel env_{P_1} \parallel env_{P'}$ with $env_{P'} = \tau \parallel_{\{\epsilon_1\}} (\epsilon_1.\varepsilon_1.\mathbf{0}[H_1]) \parallel_{\{\epsilon_2\}} (\epsilon_2.\varepsilon_2.\mathbf{0}[H_2]) \parallel env_{P_2} \parallel_{\{\epsilon_3\}} (\epsilon_3.\varepsilon_3.\mathbf{0}[H_3]) \parallel env_{P_3} \parallel \cdots$, which will collect the $Test$ and $Change$ actions generated by P', and

$$H_i(Test) = Change_{count\_val} \wedge \mathcal{A}(\varepsilon_i.\mathbf{0}) \neq \mathcal{A}(Test.\mathbf{0}) \wedge same_v(\varepsilon_i, Test) \wedge Test \in \mathcal{A}(env) \text{ with } i = 1, 2,$$
$$H_3(Test) = Change_{dout} \wedge \mathcal{A}(\varepsilon_3.\mathbf{0}) \neq \mathcal{A}(Test.\mathbf{0}) \wedge same_v(\varepsilon_3, Test) \wedge Test \in \mathcal{A}(env), \text{ and}$$
$$H_i(\varepsilon_i) = \tau \wedge \mathcal{A}(\varepsilon_i.\mathbf{0}) = \mathcal{A}(Test.\mathbf{0}) \wedge same_v(\varepsilon_i, Test) \wedge (Test \in \mathcal{A}(env)) \text{ with } i = 1, 2, 3.And$$

ϵ_1, ϵ_2, ϵ_3 and ε_1, ε_2 and ε_3 be aliases for $Assign_{load=true}$, $Assign_{din=0}$, $Assign_{load=false}$ and $Test_{load=true}$, $Test_{din=0}$ and $Test_{load=false}$ separately.

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 $(P_1 \parallel \cdots \parallel P_n) \parallel env$, and regard-

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_is are synchronous on action Δ and $Notify$ enables state of Δ in P_i. So our method is to concatenate every P_i with $\Delta.\surd$, as well as make $Notify$ enable the Δ in P_i. Now, the simulation model would be $(P_1; \Delta.\surd \parallel_{\{\Delta\}} \cdots \parallel_{\{\Delta\}} P_n; \Delta.\surd) \parallel env$.

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, a monitoring method process instance is triggered, and then executes from beginning to end. Intuitively, if event e_α is in the sensitivity list of method process P, α will trigger P once α is enabled, however, different occurrences of e_α 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, $P' := \tau_\alpha.P; \tau_{end}.\surd \parallel_{\{\tau_\alpha, \tau_{end}\}} (\tau_\alpha.\tau_{end}.P')$, P' (with $env = \tau \parallel_{\{Clock_\top, Clock_\bot\}} P_{clock} \parallel_{\{\tau_\alpha\}} (\alpha.\tau_\alpha.\mathbf{0})$) represents the process P with a sensitive action α, 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 $\{e1, e2, e_{Change}\}$ (section 3.1.1), say once ud_e1, ud_e2 or $Change$ happened, the process P_1 runs, so $P = (\tau_{ud\_e1}.\mathbf{0} \parallel \tau_{ud\_e2}.\mathbf{0} \parallel \tau_{Change}.\mathbf{0}); P_1; \tau_{end}.\surd \parallel_{\{\tau_{ud\_e1}, \tau_{end}\}} (\tau_{ud\_e1}.\tau_{end}.P) \parallel_{\{\tau_{ud\_e2}, \tau_{end}\}} (\tau_{ud\_e2}.\tau_{end}.P) \parallel_{\{\tau_{Change}, \tau_{end}\}} (\tau_{Change}.\tau_{end}.P)$ is the formalization of the process with sensitivity list (env_P is not shown here for simplicity). But a function associated with a thread or clocked thread process instance is called once and only once by the kernel, except when a clocked thread process is reset [3], so the form of the thread process P with sensitivity list $\{e_\alpha\}$ will be simply like $\alpha.P$, and clocked thread process a normal thread process only with sensitivity list $\{e_{Clock_\top}, e_{Clock_\bot}\}$.

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 action will be enabled in the next delta-cycle, so they can be modeled as:

1.$P := P_1; (Notify.\mathbf{0}); P_2$ with $env = \tau \parallel_{\{Clock_\top, Clock_\bot\}} P_{clock} \parallel_{\{Notify\}} (Notify.ud\_e.\mathbf{0})$
2.$P := P_1; (Notify.\mathbf{0}); P_2$ with $env = \tau \parallel_{\{Clock_\top, Clock_\bot\}} P_{clock} \parallel_{\{Notify\}} ((t)Notify.ud\_e.\mathbf{0})$ and
3.$P := P_1; (Notify.\mathbf{0}); P_2$ with $env = \tau \parallel_{\{Clock_\top, Clock_\bot\}} P_{clock} \parallel_{\{Notify\}} (Notify.\Delta.ud\_e.\mathbf{0})$

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: $P := P_1; (\tau_\infty.\mathbf{0}); P_2$ $env = \tau \parallel_{\{Clock_\top, Clock_\bot\}} P_{clock} \parallel_{\{ud\_e\}} ((\infty)\tau_\infty.ud\_e.\mathbf{0})$. 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.$P_1; (t)\tau.\mathbf{0}; P_2$,
2.$P_1; (\tau_{ud\_e}.\mathbf{0} + \tau_\infty.\mathbf{0}); P_2$ with $env = \tau \parallel_{\{Clock_\top, Clock_\bot\}} P_{clock} \parallel_{\{\tau_{ud\_e}\}} ud\_e.\tau_{ud\_e}.\mathbf{0}$

```
//count.h
1 # include "systemc.h"
2 SC_MODULE(count){
3       sc_in <bool> load;
4       sc_in <int> din;
5       sc_in <bool> clock;
6       sc_out <int> dout;
7       int count_val;

8       void count_up();

9       SC_CTOR(count) {
10            SC_METHOD(count_up);
11            sensitive_pos << clock;
12    }
13 };
```

Figure 5 count.h

```
//count.cc
1 # include "systemc.h"
2 # include "count.h"
3  void count::count_up( ) {
4       if (load) {
5              count_val=din;
6            } else{
7              count_val=count_val+1;
8            }
9             dout=count_val;
10 };
```

Figure 6 count.cc

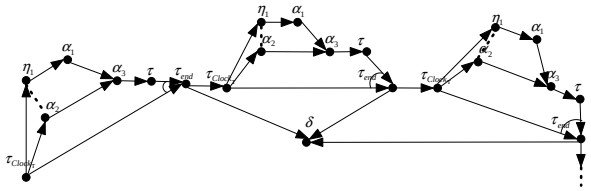

Figure 7 Event structure of count

3.$P_1;(\tau_{ud\_e1}.\mathbf{0}+\tau_{ud\_e2}.\mathbf{0});P_2$ with
$env=\tau \parallel_{\{Clock_\top,Clock_\bot\}} P_{clock} \parallel_{\{\tau_{ud\_e1}\}} ud\_e1.\tau_{ud\_e1}.\mathbf{0} \parallel_{\{\tau_{ud\_e2}\}} ud\_e2.\tau_{ud\_e2}.\mathbf{0}$

4.$P_1;(\tau_{ud\_e1}.\mathbf{0} \parallel \tau_{ud\_e2}.\mathbf{0}+\tau_\infty.\mathbf{0});P_2$ with
$env=\tau \parallel_{\{Clock_\top,Clock_\bot\}} P_{clock} \parallel_{\{\tau_{ud\_e1}\}} ud\_e1.\tau_{ud\_e1}.\mathbf{0} \parallel_{\{\tau_{ud\_e2}\}} ud\_e2.\tau_{ud\_e2}.\mathbf{0}$

5.$P_1;(\tau_{ud\_e1}.\mathbf{0}+\tau_{ud\_e2}.\mathbf{0}+(t)\tau.\mathbf{0});P_2$ with
$env=\tau \parallel_{\{Clock_\top,Clock_\bot\}} P_{clock} \parallel_{\{\tau_{ud\_e1}\}} ud\_e1.\tau_{ud\_e1}.\mathbf{0} \parallel_{\{\tau_{ud\_e2}\}} ud\_e2.\tau_{ud\_e2}.\mathbf{0}$

6.$P_1;(\tau_{ud\_e1}.\mathbf{0} \parallel \tau_{ud\_e2}.\mathbf{0}+(t)\tau.\mathbf{0});P_2$ with
$env=\tau \parallel_{\{Clock_\top,Clock_\bot\}} P_{clock} \parallel_{\{\tau_{ud\_e1}\}} ud\_e1.\tau_{ud\_e1}.\mathbf{0} \parallel_{\{\tau_{ud\_e2}\}} ud\_e2.\tau_{ud\_e2}.\mathbf{0}$

7.$P_1;(\tau_{Change_1}.\mathbf{0}+\cdots+\tau_{Change_m}.\mathbf{0});P_2$ with
$env=\tau \parallel_{\{Clock_\top,Clock_\bot\}} P_{clock} \parallel_{\{\tau_{Change_1}\}} Change_1.\tau_{Change_1}.\mathbf{0} \parallel \cdots \parallel_{\{\tau_{Change_m}\}} Change_m.\tau_{Change_m}.\mathbf{0}$

for τ_{Change_i} monitor the sensitivity list (if the positive or negative clock wedge is monitored, $\tau_{Clock_\top}$ or $\tau_{Clock_\bot}$ 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: $P := \tau_{Clock_\top}.P_1;\tau_{end}.\surd \parallel_{\{\tau_{Clock_\top},\tau_{end}\}} (\tau_{Clock_\top}.\tau_{end}.P)$ with

$P_1 := (\eta_1.\alpha_1.\mathbf{0}+\alpha_2.\mathbf{0});\alpha_3.\surd$

$env_P = \tau \parallel_{\{\tau_{Clock_\top}\}} (Clock_\top.\tau_{Clock_\top}.\mathbf{0}) \parallel_{\{\alpha_1\}} (\alpha_1.\gamma_1.\mathbf{0}[H_1]) \parallel_{\{\alpha_2\}} (\alpha_2.\gamma_2.\mathbf{0}[H_2]) \parallel_{\{\alpha_3\}} (\alpha_3.\gamma_3.\mathbf{0}[H_3]) \parallel_{\{\eta_1\}} (\epsilon_1.\eta_1.\mathbf{0}) \parallel \cdots$

that

$H_i(Test) = Change_{count\_val} \wedge \mathcal{A}(\gamma_i.\mathbf{0}) \neq \mathcal{A}(Test.\mathbf{0}) \wedge same_v(\gamma_i, Test) \wedge Test \in \mathcal{A}(env)$ with $i = 1, 2$

$H_3(Test) = Change_{dout} \wedge \mathcal{A}(\gamma_3.\mathbf{0}) \neq \mathcal{A}(Test.\mathbf{0}) \wedge same_v(\gamma_3, Test) \wedge Test \in \mathcal{A}(env)$

$H_i(\gamma_i) = \tau \wedge \mathcal{A}(\gamma_i.\mathbf{0}) = \mathcal{A}(Test.\mathbf{0}) \wedge same_v(\gamma_i, Test) \wedge (Test \in \mathcal{A}(env))$ with $i = 1, 2, 3$

(Let α_1, α_2, α_3, γ_1, γ_2, γ_3 and η_1 be aliases for $Assign_{count\_val=din}$, $Assign_{count\_val=count\_var+1}$, $Assign_{dout=count}$, $Test_{count\_val=din}$, $Test_{count\_val=count\_var+1}$, $Test_{dout=count}$, and $Test_{load=true}$.)

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., $\mathcal{F}_B(\bot)$ for P, can be found in *Figure* 3.

Unabriged 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 $Q = \tau_{Clock_\top}.Q_1$ (pictured in *Figure* 10, (a) for $\mathcal{F}_B(\bot)$ and (b) for Q) with $Q_1 = \epsilon_1.\mathbf{0};\epsilon_2.\mathbf{0};(Clock_\top.\mathbf{0}+(\infty)\tau_\infty.\mathbf{0});\epsilon_3.\mathbf{0};(Clock_\top.\mathbf{0}+(\infty)\tau_\infty.\mathbf{0});Q_1$, and the environment, $env_Q = \tau \parallel_{\{\tau_{Clock_\top}\}} (Clock_\top.\tau_{Clock_\top}.\mathbf{0}) \parallel env'_Q$ with $env'_Q = \tau \parallel_{\{\epsilon_1\}} \epsilon_1.\eta_1.\mathbf{0}[H_1] \parallel_{\{\epsilon_2\}} \epsilon_2.\eta_2.\mathbf{0}[H_2] \parallel_{\{\epsilon_3\}}$

```
// count_sim.h
1 # include "systemc.h"
2 SC_MODULE(count_stim){
3     sc_out <bool> load;
4     sc_out <int> din;
5     sc_in <bool> clock;
6     sc_out <int> dout;
7     void stimgen( );

8     SC_CTOR(count_stim) {
9           SC_THREAD(stimgen);
10           sensitive_pos(clock);
11    }
12 };
```

Figure 8 count_sim.h

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

Figure 9 count_sim.cc

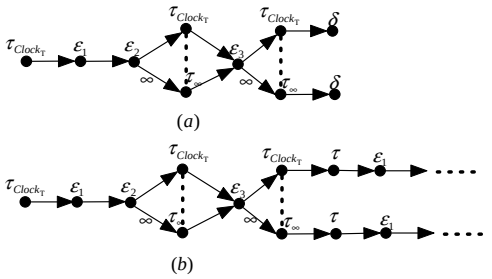

Figure 10 Event structure of count_sim

$\epsilon_3.\eta_3.\mathbf{0}[H_3] \parallel \cdots$, and

$$H_i(Test) = Change_{count\_val} \wedge \mathcal{A}(\eta_i.\mathbf{0}) \neq \mathcal{A}(Test.\mathbf{0}) \wedge same_v(\eta_i, Test) \wedge Test \in \mathcal{A}(env) \text{ with } i = 1, 2$$

$$H_3(Test) = Change_{dout} \wedge \mathcal{A}(\eta_3.\mathbf{0}) \neq \mathcal{A}(Test.\mathbf{0}) \wedge same_v(\eta_3, Test) \wedge Test \in \mathcal{A}(env)$$

$$H_i(\eta_i) = \tau \wedge \mathcal{A}(\eta_i.\mathbf{0}) = \mathcal{A}(Test.\mathbf{0}) \wedge same_v(\eta_i, Test) \wedge (Test \in \mathcal{A}(env)) \text{ with } i = 1, 2, 3$$

(Let ϵ_1, ϵ_2, ϵ_3 and η_1, η_2 and η_3 be aliases for $Assign_{load=true}$, $Assign_{din=0}$, $Assign_{load=false}$ and $Test_{load=true}$, $Test_{din=0}$, $Test_{load=false}$.)

Then the formalization for the whole codes will be $B := P; \Delta.\surd \parallel_{\{\Delta\}} Q; \Delta.\surd \parallel env$ with env equals to

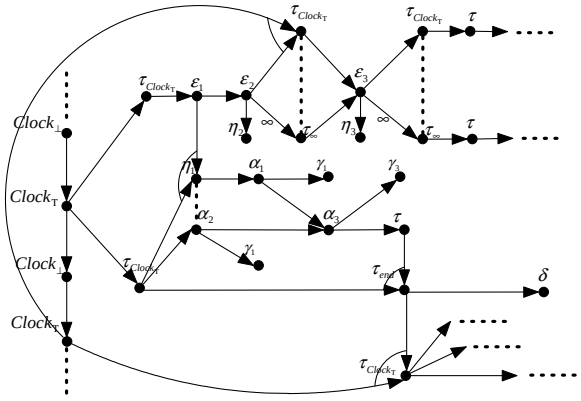

Figure 11 Event Structure of the case-study

$\tau \parallel_{\{Clock_\top, Clock_\bot\}} P_{clock} \parallel env_P \parallel env_Q$. Both P and Q involve the process instantiation, as a result, the scale of final event structure is infinity, we depict the event structure in *Figure* 11.

Then, let us introduce the operational semantics of this case-study, we do not elaborate the verbose form of B, but show a typical form to depict the way of using transition rules to deduce a system. Let us use the follows as an example: Q_1 ($Q_1 = \epsilon_1.\mathbf{0}; \epsilon_2.\mathbf{0}; (Clock_\top.\mathbf{0} + (\infty)\tau_\infty.\mathbf{0}); \epsilon_3.\mathbf{0}; (Clock_\top.\mathbf{0} + (\infty)\tau_\infty.\mathbf{0}); Q_1$). We first equip process instantiation Q_1 and all occurrences of action-prefix and $\surd$ with unique identifiers (Section 2.1), let $Q_1 := B$ that $B = \epsilon_{1_\theta}.\mathbf{0}; \epsilon_{2_\kappa}.\mathbf{0}; (Clock_\top.\xi.\mathbf{0} + (\infty)\tau_\infty\rho.\mathbf{0}); \epsilon_{3_\lambda}.\mathbf{0}; (Clock_{\top_\mu}.\mathbf{0} + (\infty)\tau_{\infty_\nu}.\mathbf{0}); Q_{1_\phi}$. Then we can have the following derivation:

$$Q_1$$
$$\xrightarrow{(\theta,\epsilon_1,0)}{}^{0} [\epsilon_{2_\kappa}.\mathbf{0}; (Clock_\top.\xi.\mathbf{0} + (\infty)\tau_\infty\rho.\mathbf{0}); \epsilon_{3_\lambda}.\mathbf{0}; (Clock_{\top_\mu}.\mathbf{0} + (\infty)\tau_{\infty_\nu}.\mathbf{0}); Q_{1_\phi}](rule\ 2)$$
$$\xrightarrow{(\kappa,\epsilon_2,0)}{}^{0} [^0[(Clock_\top.\xi.\mathbf{0} + (\infty)\tau_\infty\rho.\mathbf{0}); \epsilon_{3_\lambda}.\mathbf{0}; (Clock_{\top_\mu}.\mathbf{0} + (\infty)\tau_{\infty_\nu}.\mathbf{0}); Q_{1_\phi}]](rule\ 2)$$
$$\xrightarrow{(\xi,Clock_\top,2)}{}^{2} [^0[^0[\epsilon_{3_\lambda}.\mathbf{0}; (Clock_{\top_\mu}.\mathbf{0} + (\infty)\tau_{\infty_\nu}.\mathbf{0}); Q_{1_\phi}]]](rule\ 3.1)$$
$$\xrightarrow{(\lambda,\epsilon_3,2)}{}^{0} [^2[^0[^0[(Clock_{\top_\mu}.\mathbf{0} + (\infty)\tau_{\infty_\nu}.\mathbf{0}); Q_{1_\phi}]]]](rule\ 2)$$
$$\xrightarrow{(\mu,Clock_\top,5)}{}^{3} [^0[^2[^0[^0[Q_{1_\phi}]]]]](rule\ 3.1)$$
$$\xrightarrow{(\phi\theta,\epsilon_1,5)}{}^{3} [^0[t_1[^0[^0[\phi(^0[Q_{1_\phi}])]]]]](rule\ 7.2)$$
$$\xrightarrow{(\phi\kappa,\epsilon_2,7)}{}^{3} [^0[t_1[^0[^0[\phi(^2[^0[Q_{1_\phi}]])]]]](rule\ 2)$$
$$\cdots$$

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 dose 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, 60773108, 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. 20090009110006 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

[1] R.M. Aiex, *Conjectured statistics for the q,t-Catalan numbers*, Advances in Math. 208 (2003), pp. 13–26.

[2] *SystemC 2.0.1 Language Reference Manual*, Open SystemC Initiative (OSCI), 2003.

[3] *IEEE Standard SystemC Language Reference Manual* IEEE computer Society, IEEE Std 1666-2005, 2005.

[4] Winskel, G., *An introduction to event structures*, In the lecture notes for the REX summerschool in temporal logic, May 88, in Springer Lecture Notes in C.S., vol.354, 1988.

[5] Joost-Pieter Katoen *Quantitative and qualitative extensions of event structures* CTIT Ph. D-thesis series, 96-09.

[6] Chris J. Myers, *Computer Aided Synthesis and Verification of Gate-Level Timed Circuits*, PhD Thesis, Stanford University, October, 1995

[7] Mila Majster-Cederbaum and Jinzhao Wu, *Action Refinement for True Concurrent Real Time*, In proc. IEEE International Conf. Engineering of Complex Computer Systems (ICECCS), 2001, pp58–68.

[8] van Glabbeek, R. J., *The linear time-branching time spectrum (extended abstract)*, In Proc. on Theories of concurrency : unification and extension, 1990, pp 278-297.

[9] Man, K. L., *SystemC$^{\mathbb{FL}}$: Formalization of SystemC*, In Proc. of 12th IEEE Mediterranean Electrotechnical Conference (MELECON), 2004.5, pp.12–15.

[10] Xiaoqing Peng, Huibiao Zhu, Jifeng He and Naiyong Jin, *An Operational Semantics of an Event-Driven System-Level Simulator*, In 30th Annual IEEE/NASA Software Engineering Workshop, 2006.4, pp. 190-202.

[11] A. Salem, *Formal Semantics of Synchronous SystemC*, in Design Automation and Test in Europe (DATE), IEEE Computer Society, 2003, pp. 10376-10381.

[12] J. Ruf, D. Hoffmann, J. Gerlach, T. Kropf, W. Rosenstiehl, W. Mueller, *The Simulation Semantics of SystemC*, in Design Automation and Test in Europe (DATE), IEEE Press, Piscataway, NJ, USA, 2001, pp. 64-70.

[13] E. Riccobene, P. Scandurra, *Modeling SystemC Process Behavior by the UML Method State Machines*, in Rapid Integration of Software Engineering Techniques, Vol. 3475 of Lecture Notes in Computer Science, Springer, 2005, pp. 112-121.

[14] A. He, J.Wu, L. Li, *The causality based denotational semantics for systemc*, in Proc. of ICICTA, 2008, pp. 1215-1220.

[15] K. L. Man, *SystemC$^{\mathbb{FL}}$: Formal Specification and Analysis of Hardware/Software Co-designs*, in Transactions on Circuits and Systems 3 (5) (2006), pp.361-368.

[16] K. L. Man, *SystemC$^{\mathbb{FL}}$: A Formalism for Hardware/Software Co-design*, in Proc. 17th European Conference on Circuit Theory and Design 2005, pp. 193-196.

[17] R. S. Dubtsov, *Stable Event Structures and Marked Scott Domains: An Adjunction*, in Lecture Notes in Computer Science (Vol. 4378), Springer, 2007, pp. 443-450.

[18] T. Bolognesi, E. Brinksma, *Introduction to the iso specification language LOTOS*, Computer Networks and ISDN Systems, 14 (1). pp. 25-59.

[19] B. A. Davey, H. A. Priestley, *Introduction to Lattices and Order*, Cambridge University Press, 2002.

[20] R. Milner, *A Calculus of Communicating Systems*, in Lecture Notes in Computer Science(Vol. 92), Springer-Verlag, 1982.