Under consideration for publication in Formal Aspects of Computing

Denotational Semantics and Its Algebraic Derivation for an Event-driven System-level Language

Huibiao Zhu[1] Jifeng He[1] Shengchao Qin[2] Phillip J. Brooke[2]

[1]Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai 200062, China
E-mail: hbzhu@sei.ecnu.edu.cn; jifeng@sei.ecnu.edu.cn
[2]School of Computing, University of Teesside, Middlesbrough, UK
E-mail: s.qin@tees.ac.uk; pjb@scm.tees.ac.uk

Abstract. As a system-level modelling language, SystemC possesses several novel features such as delayed notifications, notification cancelling, notification overriding and delta-cycle. It is challenging to formalise SystemC. In this paper, we study the denotational semantics for SystemC using *Unifying Theories of Programming* (abbreviated as *UTP*) [HH98]. Two trace variables are introduced, one to record the state behaviours and another to record the event behaviours. The timed model is formalised in a three-dimensional structure. A set of algebraic laws is explored, which can be proved via the presented denotational semantics.

In this paper, we also consider the inverse work; i.e., generating the denotational semantics from algebraic semantics for SystemC. A complete set of parallel expansion laws is explored, where the location status of an instantaneous action is studied. We introduce the concept of head normal form for each program and every program is expressed in the form of guarded choice with location status. Based on this, the derivation strategy for deriving denotational semantics from algebraic semantics is provided.

1. Introduction

SystemC is a system-level modelling language which can be used to model a system at different abstract levels. Modelling and simulation in SystemC gives the designers early insights about the potential design problems that could arise. Compared with traditional hardware description languages, SystemC possesses several new and interesting features, including delayed notifications, notification cancelling, notification overriding and delta-cycle.

In SystemC, processes can trigger events actively while in Verilog [IEE01] events are generated based on changes of states. In SystemC, events represent some general conditions during the execution of the program. An event can be notified on many separate occasions. There are three kinds of event notifications: immediate event notifications, delta-cycle delayed notifications and timed notifications. Delayed notifications can be cancelled via cancel statements before they are triggered. Delayed notifications on the same event override each other and only one delayed notification survives.

Correspondence and offprint requests to: Huibiao Zhu, E-mail: hbzhu@sei.ecnu.edu.cn
This paper combines and extends the work published at UTP 2008 [ZHPJ10]and UTP 2010 [ZYH10].

Although SystemC comes with a user manual ([Ope01, Ope03]), a formal semantics of SystemC is needed for various applications in simulation, synthesis, and formal verification. This paper considers the denotational semantics of SystemC, where our approach is based on *Unifying Theories of Programming* (abbreviated as *UTP*) [HH98]. *UTP* was developed by Hoare and He in 1998 [HH98] and has been successfully applied in studying the semantics of programming languages and their algebraic laws, as well as the refinement calculus of different level programs. The new features of SystemC make it worthwhile to formalise its denotational semantics via *UTP* approach. In order to deal with the event notification and event driven feature, and the shared-variable feature, two trace variables are introduced, one is to record the state behaviours and another is to record the event behaviours. Our timed model is formalised in a three-dimensional structure.

As described in Hoare and He's Unifying Theories of Programming [HH98], three different mathematical models are often used to represent a theory of programming, namely, the operational, the denotational, and the algebraic approaches [Plo81, Sto77, HHH+87]. Each of these representations has distinctive advantages for theories of programming. These three semantics should provide the same understanding of the language from different viewpoints. Therefore, the linking of these three semantics is a challenging task. The traditional way to link denotational and algebraic semantics is that algebraic semantics can be explored based on the achieved denotational semantics.

To link denotational and algebraic semantics for SystemC, this paper considers the inverse work; i.e., generating the denotational semantics from algebraic semantics for SystemC. With the introduction of the concept of guarded choice, a complete set of parallel expansion laws is studied. In order to index an instantaneous action to which exact component of a parallel process, the concept of location status (i.e., locality) is introduced. To support the generating of denotational semantics, we introduce the concept of head normal form for each program. We provide the definition for deriving denotational semantics from algebraic semantics. The derived denotational semantics gives us a way to reason about program properties easily.

The rest of this paper is organized as follows. In section 2 we select a kernel subset of SystemC and present an introduction to the language. We provide the denotational semantic model in this section. The timed model of SystemC is considered in a three-dimensional structure. A set of healthiness conditions is explored in order to achieve the denotational semantics. Section 3 is devoted to the denotational semantics using the *UTP* approach. Two traces are applied for the formalization, one is to record the state behaviour and another is to record the event behaviour. Section 3 also studies the algebraic laws for sequential constructs, which can be proved via the achieved denotational semantics. Section 4 investigates the derivation of denotational semantics from algebraic semantics for SystemC. We give the concept of guarded choice with locality and investigate a complete set of parallel expansion laws in this section. The definition of head normal form for each statement is provided. Based on this, we provide a strategy for deriving denotational semantics from algebraic semantics in this section. Finally section 5 concludes the paper and presents some possible future work.

2. The Semantic Model of SystemC

2.1. The Syntax of SystemC

In this paper we select a kernel subset of SystemC for exploring its semantics. Although it is a subset of SystemC, it still covers the interesting and major features, such as delay notifications, notification cancelling, notification overriding, channels, concurrent processes and delta-cycle. In this section, we present the syntax of the selected subset and give a brief introduction to its interesting features.

For simplicity, we omit the syntactic elements for representing the architecture of a SystemC program. The subset language adopts a C-like syntax:

$$
\begin{aligned}
PP &::= P \mid PP \parallel PP \\
P &::= \mathbf{Skip} \mid v := exp \mid chan\_stmt \mid event\_stmt \mid wait\_stmt \\
&\quad \mid P; P \mid \mathbf{if}\ b\ \mathbf{then}\ P\ \mathbf{else}\ P \mid \mathbf{while}\ b\ \mathbf{do}\ P \\
chan\_stmt &::= ch??v \mid ch!!exp \\
event\_stmt &::= notify(e_{\Delta 0}) \mid notify(e_{\Delta 1}) \mid notify(e_{\sharp T}) \mid cancel(e)
\end{aligned}
$$

$$
\begin{aligned}
wait\_stmt &::= wait(\Delta 1) \mid wait(\sharp T) \mid wait(e\_list) \\
e\_list &::= single\_e \mid \mathbf{or}_{i\in I}\{single\_e_i\} \\
single\_e &::= e \mid pe(ch) \mid ne(ch)
\end{aligned}
$$

The meanings of statements such as **Skip**, assignment ($v := exp$), sequential composition ($P; Q$), conditional (**if** b **then** P **else** Q) and iteration (**while** b **do** P) are similar to the conventional programming language.

The channel output statement $ch!!exp$ is executed in the evaluation phase, which generates a request to update the channel. These update requests will be carried out in the following update phase. The channel input statement $ch??v$ assigns the current value of channel ch to variable v.

An event is notified by statement $notify$. An event can be notified immediately (i.e., $notify(e_{\Delta 0})$) or after a period of time (i.e., $notify(e_{\Delta 1})$) or $notify(e_{\sharp T})$). $notify(e_{\Delta 1})$) generates event e and this event e will be active after one delta-cycle (i.e., one micro time unit). $notify(e_{\sharp T})$) generates event e and this event e will be active after a period of specified simulation time T (i.e., T macro time units). Statement $cancel(e)$ cancels the delayed notifications on event e.

A process may wait for the arrival or firing of an event. These events can be classified into two types; i.e., single events or complex events. Single events can have three forms; i.e., e, $pe(ch)$ and $ne(ch)$, where event e can be generated by event notifications. $wait(pe(ch))$ is fired only when the current value of channel ch is greater than its previous value, whereas $wait(ne(ch))$ stands for the opposite firing case. Complex events can be of the form $\mathbf{or}_{i\in I}\{single\_e_i\}$. For the waiting of complex events, if anyone is fired or becomes active, the whole waiting behaviour becomes fired or active.

Different from traditional hardware description languages, time delay has two types, micro time advance and macro time advance. $wait(\Delta 1)$ stands for one unit micro time (i.e, one delta-cycle) advancing, whereas $wait(\#T)$ stands for T units macro time advancing.

$P \parallel Q$ means P runs in parallel with Q. Their communication is through channels and variables. Further, their synchronization is based on events.

If any branch processes of a parallel process are ready to run, one branch will be selected to be executed. The selection is nondeterministic. Channels will be updated when a waiting command is encountered during the current execution. If all branch processes are still waiting, then time will be advanced. Micro time (delta-cycle) will be advanced first. If that does not activate any processes, then macro time will be advanced. The execution is proceeded by the following steps.

(1) Evaluation Phase. Select a ready process to execute. The order of selection is nondeterministic. The selected process executes until a waiting command is encountered. This sequence of instantaneous commands forms an atomic action, which is uninterrupted.
The execution of a process may cause immediate event notifications to occur. It may also generate pending requests to update channels in the following update phase.

(2) Update Phase. Carry out all pending channel update requests generated in the previous evaluation phase, which may generate some events $pe(ch)$ or $ne(ch)$. Then go to step (1).

(3) Micro Time (Delta-cycle) Advancing Phase. If there are no processes ready to run and no pending channel update requests, but there exist pending delta-cycle notifications or delta-cycle timeouts, advance the delta-cycle. Then determine which processes are ready to run and go to step (1).

(4) Macro Time Advancing Phase. If there are no processes ready to run, no pending channel update requests, no pending delta-cycle notifications and no delta-cycle timeouts, advance the current macro time by one time unit. And determine which processes become ready to run due to events or timeouts that are triggered at the current time. If any processes are ready to run, then go to step (1), otherwise advance the current macro time by one time unit again.

2.2. The Denotational Semantics Model

SystemC possesses the feature of shared-variable concurrency. To deal with this, we introduce a sequence type variable $tr1$ for recording the state change of a program. Moreover, SystemC not only has the feature

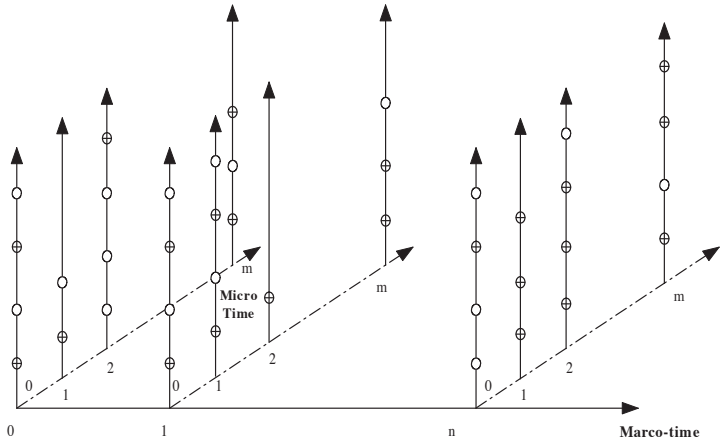

Fig. 1.

of traditional time delay, it also contains the feature of Δ time delay (i.e., micro time delay). Therefore, the structure of $tr1$ can be depicted as Figure 1.

At the relative macro time "i" point, time may also advance in Δ time step, standing for the micro-time advancing. Therefore, a sequence of behaviours may be recorded at each Δ time point. These behaviours can be classified into two types; i.e., contributed by the process itself or its environment. In Figure 1, the symbols "$\oplus$" and "$\circ$" stand for the contribution by the process itself and its environment respectively.

In order to record these behaviours, the concept of snapshot is introduced, expressed as (σ, f), where σ stands for the contribution of the behaviour and f stands for the flag. "$f = 1$" indicates that the behaviour is contributed by the process itself and "$f = 0$" indicates that the behaviour is contributed by its environment. Below is the formal structure of trace $tr1$.

$$Element1 = \{(\sigma, f) | \sigma \in State \wedge f \in \{0, 1\}\},$$
$$tr1 \in \mathbf{seq}(\mathbf{seq}(\mathbf{seq}(Element1)))$$

Here, $\mathbf{seq}(T)$ stands for a sequence type, where each sequence is composed of elements from type T.

We select the components of a snapshot using projections.

$$\pi_1((\sigma, f)) =_{df} \sigma \quad \text{and} \quad \pi_2((\sigma, f)) =_{df} f$$

In SystemC, waiting guards can be triggered by events, which can be generated by the process itself or its environment. We use the trace variable $tr2$ to record all the events generated by the process or its environment. $tr2$ has the same time structure, as shown in the above Figure 1. It can be defined as below.

$$Element2 = \{(e, f) | e \in Event \wedge f \in \{0, 1\}\}$$
$$tr2 \in \mathbf{seq}(\mathbf{seq}(\mathbf{seq}(Element2)))$$

For any $tr1$ (or $tr2$) type trace s, $len(s)$ stands for the length of sequence s; i.e., it stands for the length of macro-time advancing. $s[0]$ and $s[len(s) - 1]$ stand for traces of the start point and end point of the current macro-time observation interval. Furthermore, $s[i][j]$ stands for the trace behaviour at the point of macro-time i and micro-time j.

Example 2.1 Let $P_i = notify(e_{i\Delta 0})$; $notify(f_{i\Delta 0})$; $u_i := u_i + 1$; $v_i := v_i + 2 \quad (i = 1, 2)$. Assume that the initial states for the above four shared variables are 0. Consider the traces $tr1$, $tr2$ for process P_1, P_2 and $P_1 \parallel P_2$.

As the four statements in P_1 and P_2 form an atomic action respectively. Either $notify(e_{1\Delta 0})$ or $notify(e_{2\Delta 0})$ can be scheduled first. For all these considered traces, their lengths are 0, and their lengths at the current macro time point are also 0.

If $notify(e_{1\Delta 0})$ is scheduled first, below are the three $tr1$ traces at the point of macro time 0 and micro time

0 for P_1, P_2 and $P_1 \parallel P_2$ respectively.

$$\langle(\sigma_1,1)\rangle^\frown\langle(\sigma_2,0)\rangle, \qquad \langle(\sigma_1,0)\rangle^\frown\langle(\sigma_2,1)\rangle, \qquad \langle(\sigma_1,1)\rangle^\frown\langle(\sigma_2,1)\rangle$$

where, $\sigma_1 = \{u_1 \mapsto 1,\ v_1 \mapsto 2,\ u_2 \mapsto 0,\ v_2 \mapsto 0\}$,

$\sigma_2 = \{u_1 \mapsto 1,\ v_1 \mapsto 2,\ u_2 \mapsto 1,\ v_2 \mapsto 2\}$

Here, σ_1 stands for the contribution of "$u_1 := u_1+1\ ;\ v_1 := v_1+2$", whereas σ_2 stands for the contribution of the execution of "$u_2 := u_2+1\ ;\ v_2 := v_2+2$". The sequence $\langle(\sigma_1,1)\rangle^\frown\langle(\sigma_2,0)\rangle$ indicates that P_1 performs the execution of "$u_1 := u_1+1\ ;\ v_1 := v_1+2$" first, and it then also needs to record its environment's (i.e., P_2) execution of "$u_2 := u_2+1\ ;\ v_2 := v_2+2$". Similarly, the sequence $\langle(\sigma_1,0)\rangle^\frown\langle(\sigma_2,1)\rangle$ indicates that, before P_2 performs "$u_2 := u_2+1\ ;\ v_2 := v_2+2$", it also needs to record its environment's (i.e., P_1) execution of "$u_1 := u_1+1\ ;\ v_1 := v_1+2$". Therefore, the whole system (i.e., $P_1 \parallel P_2$) needs to execute "$u_1 := u_1+1\ ;\ v_1 := v_1+2$" first, after that it also needs to execute "$u_2 := u_2+1\ ;\ v_2 := v_2+2$".

In this case, three $tr2$ traces at the point of macro time 0 and micro time 0 for P_1, P_2 and $P_1 \parallel P_2$ are shown below respectively.

$$\langle(e_1,1)\rangle^\frown\langle(f_1,1)\rangle^\frown\langle(e_2,0)\rangle^\frown\langle(f_2,0)\rangle, \qquad \langle(e_1,0)\rangle^\frown\langle(f_1,0)\rangle^\frown\langle(e_2,1)\rangle^\frown\langle(f_2,1)\rangle$$
$$\langle(e_1,1)\rangle^\frown\langle(f_1,1)\rangle^\frown\langle(e_2,1)\rangle^\frown\langle(f_2,1)\rangle$$

On the other hand, if $notify(e_{2\Delta 0})$ is scheduled first, the analysis is similar. □

As $tr1$ and $tr2$ have a three-dimensional structure, we introduce the prefix definition between two $tr1$ (or $tr2$) type traces, denoted as $\preceq_1$.

Definition 2.2

$$s \preceq_1 t =_{df} \exists m, n \bullet \left(\begin{array}{l} m = len(s) \wedge n = len(t) \wedge m \leq n \wedge \\ \forall i \in \{0..m-2\} \bullet s[i] = t[i] \wedge \\ \exists k \bullet \left(\begin{array}{l} k = len(s[m-1]) \wedge \\ \forall l \in \{0..k-2\} \bullet s[m-1][l] = t[m-1][l] \wedge \\ \qquad\qquad s[m-1][k-1] \preceq t[m-1][k-1] \end{array} \right) \end{array} \right)$$

□

For $s \preceq_1 t$ (i.e., the prefix of three-dimensional time structure), the length of s is smaller than or equal to the length of t. This means that the final macro time point of s is smaller than or equal to that of t. Further, for every macro time point i less than the final macro time point of s, $s[i]$ and $t[i]$ are of two-dimensional time structure and they should be equal. For the final macro time point $m-1$ of s and its corresponding final micro time point $k-1$, for $l \in \{0..k-2\}$, $s[m-1][l]$ and $t[m-1][l]$ can be different, and they should satisfy the traditional prefix condition $s[m-1][l] \preceq t[m-1][l]$, reflected in the last line of the definition.

For traditional sequences s and t, $t-s$ stands for the sequence that subtracts sequence s from t with respect to the traditional prefix structure $\preceq$. On the other hand, if s and t are sequences of $tr1$ (or $tr2$) type structure, $t-s$ has a similar meaning with respect to the new $\preceq_1$ prefix structure.

The execution of an atomic action is represented by a single snapshot. To describe the behaviour of an individual shared variable assignment, we introduce a variable ttr to model the accumulated changes made by the statements of the atomic action. An assignment is simply formulated as storing the result in variable ttr. Meanwhile, the current value of channel ch is also stored in variable ttr. On the completion of an atomic action, the corresponding snapshot is attached to the end of the trace to record its behaviour.

The event generated by the channel receiving will not be immediately attached to the end of the trace variable $tr2$. After all the behaviours in an atomic action complete, the process enters into the update phase. Hence we use a trace variable RQ to record new channel states due to the channel receiving.

Example 2.3 Let $P =_{df} x := x+1\ ;\ y := y = 1\ ;\ ch!!(x+y)$. Assume that shared-variables x and y are 0 and 1 respectively when P is activated. Also assume that the value recorded in channel is initially 0.

The execution of $x := x+1$ produces $ttr = \{x \mapsto 1,\ y \mapsto 0\}$, whereas the execution of $y := y+1$ produces $ttr = \{x \mapsto 1,\ y \mapsto 2\}$.

Further, the execution of "$ch!!(x+y)$" produces $RQ = \langle(ch,2)\rangle$, where value 2 stands for the current value of expression "$x+y$".

When the event guard $wait(e)$ is encountered, it will attach snapshot $(ttr, 1)$ to the end of trace $tr1$. Meanwhile, an event $pe(ch)$ is attached to the end of trace $tr2$ in the form of $(pe(ch), 1)$, due to the execution of $ch!!(x + y)$ and the encountering of $wait(e)$. □

Three kinds of event notifications are introduced in SystemC for generating events. $notify(e_{\Delta 0})$ is used to generate event e, which will be active immediately. For $notify(e_{\Delta 1})$, it can generate event e that will be active in one micro time unit. Moreover, $notify(e_{\#T})$ also generates event e. However, it can only be active in T macro time units. For recording the events contributed by the above last two notification commands, we introduce two set type variables, $EN2$ and $EN3$. Here, $EN2$ records the generated events, which will be active in one micro time unit. $EN3$ contains the pairs (e, T), which indicates that event e will be active in T macro time units.

Example 2.4 Let $P = notify(e_{1\,\Delta 0})\ ;\ notify(e_{2\,\Delta 1})\ ;\ notify(e_{3\,\#2})\ ;\ notify(f_{1\,\#4})$. Assume that $EN2 = \{e_1\}$ and $EN3 = \{(e_3, 1), (f_1, 5)\}$. Here e_1, e_2, e_3 and f_1 are all events. Now we consider new $EN2$ and $EN3$ after the execution of all these notifications.

The first immediate notification will record event e_1 in the trace variable $tr2$, which may fire the environment's waiting command immediately. Moreover, event e_1 should also be removed from $EN2$, while $EN3$ remains unchanged. The execution of the second notification command will add event e_2 to $EN2$ and also keep $EN3$ unchanged.

As $(e_3, 1)$ already belongs to $EN3$, the execution of the third command will not add anything to $EN3$. Furthermore, the fourth command will remove $(f_1, 5)$ from $EN3$ and add $(f_1, 4)$ to $EN3$ because the time stamp in $(f_1, 4)$ is smaller than the time stamp in $(f_1, 5)$. Therefore, the final values of $EN2$ and $EN3$ are:

$$EN2 = \{e_2\} \quad \text{and} \quad EN3 = \{(e_3, 1),\ (f_1, 4)\}$$ □

The execution of a SystemC process can never undo an atomic action that has already been performed. A formula P which identifies a program must therefore imply this fact; i.e., it has to meet the following healthiness condition:

$$(H1)\ P\ =\ P \wedge Inv(tr1, tr2), \qquad \text{where } Inv(tr1, tr2)\ =_{df}\ (tr1 \preceq_1 tr1') \wedge (tr2 \preceq_1 tr2')^1$$

Here $Inv(tr1, tr2)$ indicates $tr1$ and $tr2$ are the prefix of $tr1'$ and $tr2'$ respectively, which indicates that trace can only get longer. As in relational calculus, for any denotational variable u, we use u and u' to stand for the initial value and final value for the current execution respectively.

A SystemC process may perform an infinite computation and enter a *divergent state*. To distinguish its chaotic behaviour from the stable ones we introduce the variables ok, $ok' : Bool$ into the semantical model, where $ok = true$ indicates that the process has been started, and $ok' = true$ states that the process is *stable* currently. $ok\ =\ false$ means that the program has never started and even the initial values are unobservable.

Definition 2.4 Let Q and R be formulae not containing ok and ok'. Define

$$Q \vdash R\ =_{df}\ \neg ok \wedge Inv(tr1, tr2)\ \vee\ \neg Q\ \vee\ (ok' \wedge R)$$

A *design* is a formula that is expressed in this form. □

A time-controlled statement cannot start its execution before its guard is triggered. To distinguish its waiting behaviour from terminating one, we introduce another pair of variables $wait, wait' : Bool$. When $wait$ is true the program is started in an intermediate state, and when $wait'$ is true the program is idle. Therefore, for sequential composition "$R\ ;\ P$", all the intermediate observations of R are also the intermediate observations of "$R\ ;\ P$". Control can pass from R to P only when R is in its terminating state, distinguished by the fact that $wait'$ is false. If program P is asked to start in a waiting state of R, it leaves the state unchanged.

$$(H2)\ P\ =\ I\!I \lhd wait \rhd P,$$

where, $I\!I\ =_{df}\ \mathbf{true} \vdash (\bigwedge_{s \in \{tr1, tr2, ttr, X, RQ, EN2, EN3, wait\}} s' = s)$

and $P \lhd b \rhd Q\ =_{df}\ (P \wedge b) \vee (\neg b \wedge Q)$

[1] In this paper we use X and X' to stand for the initial value and final value for variable X respectively.

Here, X stands for the vector containing all the local variables for the current program. $X' = X$ indicates that all the local variables remain unchanged.

Definition 2.5 Formula P is healthy iff there exists a design $D = (Q \vdash (W \lhd wait' \rhd T))$ such that $P = \mathbf{H}(D)$, where

$$\mathbf{H}(Y) =_{df} (I\!I \lhd wait \rhd (Y \wedge Inv(tr1, tr2))$$ □

Theorem 2.6 $\mathbf{H}(Y)$ satisfies healthiness condition $(H1)$ and $(H2)$. □

Now we give the definition for sequential composition.

Definition 2.7 Let P_1 and P_2 be formulae. Define

$$P_1 \ ; \ P_2 =_{df} \exists S \bullet (P_1[S/V'] \wedge P_2[S/V])$$

where, V stands for the list of all denotational variables in our model; i.e., ok, $tr1$, $tr2$, ttr, X, RQ, $EN2$,$EN3$, $wait$. □

For the healthy formula $\mathbf{H}(Q \vdash W \lhd wait' \rhd T)$, $\neg Q$, W and T represent the divergent behaviour, waiting behaviour and terminating behaviour respectively. Now we provide a simple refinement calculus for healthy formulae and show that they are closed under sequential composition, conditional choice, disjunction and conjunction.

Theorem 2.8

If $\neg Q_i = \neg Q_i \wedge Inv(tr1, tr2)$, $W_i = W_i \wedge Inv(tr1, tr2)$, $T_i = T_i \wedge Inv(tr1, tr2)$ for $i = 1, 2$, then

(1) $\mathbf{H}(Q_1 \vdash W_1 \lhd wait' \rhd T_1) \ ; \ \mathbf{H}(Q_2 \vdash W_2 \lhd wait' \rhd T_2)$
$= \mathbf{H}(\neg(\neg Q_1 \ ; \ Inv(tr1, tr2)) \wedge \neg(T_1 \ ; \ \neg Q_2) \vdash (W_1 \vee (T_1 \ ; \ W_2)) \lhd wait' \rhd (T_1 \ ; \ T_2))$

(2) $\mathbf{H}(Q_1 \vdash W_1 \lhd wait' \rhd T_1) \lhd b \rhd \mathbf{H}(Q_2 \vdash W_2 \lhd wait' \rhd T_2)$
$= \mathbf{H}((Q_1 \lhd b \rhd Q_2) \vdash (W_1 \lhd b \rhd W_2) \lhd wait' \rhd (T_1 \lhd b \rhd T_2))$

(3) $\mathbf{H}(Q_1 \vdash W_1 \lhd wait' \rhd T_1) \ \vee \ \mathbf{H}(Q_2 \vdash W_2 \lhd wait' \rhd T_2)$
$= \mathbf{H}((Q_1 \wedge Q_2) \vdash (W_1 \vee W_2) \lhd wait' \rhd (T_1 \vee T_2))$

(4) $\mathbf{H}(Q_1 \vdash W_1 \lhd wait' \rhd T_1) \ \wedge \ \mathbf{H}(Q_2 \vdash W_2 \lhd wait' \rhd T_2)$
$= \mathbf{H}((Q_1 \vee Q_2) \vdash ((Q_1 \Rightarrow W_1) \wedge (Q_2 \Rightarrow W_2)) \lhd wait' \rhd ((Q_1 \Rightarrow T_1) \wedge (Q_2 \Rightarrow T_2)))$ □

The first law stands for the calculation of the sequential composition of two processes. The divergent behavior for the whole system can be divided into two cases. The first one is simply the divergent behavior of the first process (expressed as "$\neg Q_1; Inv(tr1, tr2)$"), whereas the second case is the terminating behavior of the first process followed by the divergent behavior of the second process (expressed as "$T_1 \ ; \ \neg Q_2$"). Moreover, for the waiting behavior of the whole system, it can also be divided into two cases. The first case is the waiting behavior of the first process and the second case is the terminating behavior of the first process followed by the the waiting behavior of the second process. For the terminating behavior of the whole system, it can be described as the sequential composition of the terminating behaviors of the first process and the second process.

The other three laws stand for the calculation of the behavior of disjunction, conjunction and conditional choice of two processes, respectively. Their analysis is similar.

The laws for disjunction and conjunction can be generalised to the union and intersection of arbitrary sets. This indicates that healthy formulae form a complete lattice under the implication order. We use HF to denote the set of all healthy formulae. The weakest fixed point of a monotonic function Φ on HF can be defined by

$$\mu_{HF} X \bullet \Phi(X) =_{df} \sqcap \{F \mid F \Rightarrow \Phi(F) \textbf{ and } F \in HF\}$$

In the subsequent sections we will formalize a SystemC process P as a healthy formula of the form

$$\mathbf{H}(\neg div(P) \vdash wait(P) \lhd wait' \rhd ter(P))$$

where $div(P)$, $wait(P)$ and $ter(P)$ stand for the divergent behaviour, waiting behaviour and termination behaviour of P respectively.

3. The Denotational Semantics for SystemC

3.1. Sequential Constructs

Program variable assignment can be classified into two types: shared variable assignment and local variable assignment.

Let

$$Env(s) =_{df} \forall i, j \bullet ((0 \le i \le len(s)) \wedge (0 \le j \le len(s[i]))) \Rightarrow \pi_2(s[i][j]) \in 0^*$$

$$Instenv(s) =_{df} len(s) = 0 \wedge len(s[0]) = 0 \wedge Env(s)$$

$Env(s)$ is used to describe the phenomena that the new states (or new events) are generated by the environment. Here $\pi_2(s[i][j]) \in 0^*$ stands for the action sequence at the macro time point i and micro time point j are contributed by the environment. Meanwhile, $Instenv(s)$ behaves like $Env(s)$, and the macro time and micro time do not advance.

$$InstEnv =_{df} \mathbf{H}\left(\mathbf{true} \vdash \begin{pmatrix} \neg wait' \wedge \bigwedge_{t\in\{tr1,tr2\}} Instenv(t'-t) \\ \wedge\, ttr' = \pi_1(last(last(last(tr1')))) \\ \wedge\, same(\{X, RQ, EN2, EN3\}) \end{pmatrix}\right)$$

where $same(A) =_{df} \bigwedge_{x\in A}(x' = x)$. Here $last(s)$ stands for the last element of sequence s.

Formula $InstEnv$ indicates that the trace behaviours of $tr1$ and $tr2$ should all satisfy a condition expressed in the function $Instenv$ and the state of the last snapshot of trace $tr1$ is assigned to variable ttr. All other variables remain unchanged.

Now we consider the behaviour of **Skip**. If it is the first statement of an atomic action, its behaviour can be formalised using formula $InstEnv$. Otherwise, it behaves like $I\!I$.

$$\mathbf{Skip} =_{df} InstEnv \lhd ttr = null \rhd I\!I$$

Next we consider the definition of shared variable assignment. Let

$$sassign(v, e) =_{df} \mathbf{H}\left(\mathbf{true} \vdash \begin{pmatrix} \neg wait' \wedge ttr' = ttr[e/v] \wedge \\ same(\{tr1, tr2, X, RQ, EN2, EN3\}) \end{pmatrix}\right)$$

Formula $sassign(v, e)$ indicates that the value of expression e is assigned to v via the state variable ttr. Based on this, we can define shared-variable assignment $v := e$.

$$v := e =_{df} \mathbf{Skip}\ ;\ sassign(v, e)$$

For the definition of local variable assignment, we introduce the function $lassign(x, f)$.

$$lassign(x, f) =_{df} \mathbf{H}\left(\mathbf{true} \vdash \begin{pmatrix} \neg wait' \wedge x' = f \wedge \\ same(\{tr1, tr2, ttr, X\backslash\{x\}, RQ, EN2, EN3\}) \end{pmatrix}\right)$$

The definition of local variable assignment $x := f$ can be described as:

$$x := f =_{df} \mathbf{Skip}\ ;\ lassign(x, f)$$

Sequential composition $(P\ ;\ Q)$ behaves like P before P terminates, and then behaves like Q afterwards.

$$(P\ ;\ Q) =_{df} (P)\ ;\ (Q)$$

The definition of conditional can be based on **Skip**.

$$\textbf{if } b \textbf{ then } P \textbf{ else } Q =_{df} \textbf{Skip} \ ; \ (P \lhd b \rhd Q)$$

The iteration construct is defined in the same way as its counterpart in conventional programming languages.

$$\textbf{while } b \textbf{ do } P =_{df} \mu_{HF} X \bullet \textbf{if } b \textbf{ then } (P; X) \textbf{ else Skip}$$

where $\mu_{HF} X \bullet F(X)$ denotes the weakest fixed point of the monotonic function F over the set of healthy formulae.

3.2. Channel Communication

Firstly, we consider the message output via a channel. We define it as

$$ch!!exp =_{df} \textbf{Skip} \ ; \ RqUpdate(ch, exp)$$

The execution of channel output command can be classified into two cases. One is that the channel is in the first statement of an atomic action, while another stands for the opposite. This can be classified using formula **Skip**. The recording of channel output is expressed using formula $RqUpdate(ch, exp)$, i.e., mainly recording the value of expression exp in channel ch.

$$RqUpdate(ch, exp) =_{df} \mathbf{H}\left(\textbf{true} \vdash \begin{pmatrix} \neg wait' \wedge same(\{tr1, tr2, ttr, X, EN2, EN3\}) \\ \wedge\, RQ' = RQ \backslash (ch, -)^\frown \langle (ch, exp(y)) \rangle \end{pmatrix}\right)$$

where:

(1) y in the above two formulae stands for expression $\pi_1(last(last(last(tr1))))$.

(2) $^\frown$ stands for the concatenation of two traditional sequences.

(3) "\" is used to remove the pairs from the update sequence. It can be defined as below:

$$\langle\rangle \backslash (ch, m) =_{df} \langle\rangle$$
$$(\langle (ch, -) \rangle^\frown t) \backslash (ch, m) =_{df} t \backslash (ch, m)$$
$$(\langle (ch1, n) \rangle^\frown t) \backslash (ch, m) =_{df} \langle (ch1, n) \rangle^\frown (t \backslash (ch, m))$$

Here, $ch1 \neq ch$ and "$-$" matches to any elements.

The last line (i.e., $RQ' = RQ \backslash (ch, -)^\frown \langle (ch, exp(y)) \rangle$) in formula $RqUpdate(ch, exp)$ indicates that before appending the value and its associate channel ch to the trace variable, the snapshots concerned with the corresponding channel ch need to be removed before recording the new value of the channel.

Next we can consider message input via a specific channel $ch??w$, which can be considered as assigning the value on the channel.

If w is a shared variable, then

$$ch??w =_{df} \textbf{Skip} \ ; \ sassign(w, ch)$$

If w is a local variable, then

$$ch??w =_{df} \textbf{Skip} \ ; \ lassign(w, ch)$$

3.3. Event Notification

Now we consider the immediate event notification $notify(e_{\Delta 0})$. First, we give the definition for formula $InstEApp(e)$.

$$InstEApp(e) =_{df} \mathbf{H}\left(\mathbf{true} \vdash \left(\begin{array}{c} \neg wait' \wedge len(tr2' - tr2) = 0 \wedge \\ len((tr2' - tr2)[0]) = 0 \wedge \\ (tr2' - tr2)[0][0] = \langle (e,1) \rangle \wedge \\ same(tr1, ttr, X, RQ, EN2, EN3) \end{array}\right)\right)$$

Formula $InstEApp(e)$ indicates that event e is attached to the end of trace variable $tr2$ without macro or micro time advancing. The two formulae "$len(tr2' - tr2) = 0$" and "$len((tr2' - tr2)[0]) = 0$" indicate that neither macro time nor micro time will advance. The attaching behaviour is expressed using formula "$(tr2' - tr2)[0][0] = \langle (e,1) \rangle$".

Next we give the definition for formula $EveUpd0(e)$.

$$EveUpd0(e) =_{df} \mathbf{H}\left(\mathbf{true} \vdash \left(\begin{array}{c} \neg wait' \wedge same(tr1, tr2, ttr, X, RQ) \\ \wedge EN2' = f1(EN2, e) \\ \wedge EN3' = g1(EN3, e) \end{array}\right)\right)$$

where $f1(A, e) =_{df} \{x \mid x \in A \wedge x \neq e\}$

$g1(A, e) =_{df} \{(x, T) \mid (x, T) \in A \wedge x \neq e\}$

We describe $EveUpd0(e)$'s purpose: For immediate event notification $notify(e_{\Delta 0})$, after event e is attached to the end of the trace variable, two set type variables $EN2$ and $EN3$ need to be modified due to the attachment of event e to trace variable $tr2$. This modification is reflected by the two functions $f1(A, e)$ and $g1(A, e)$. Event e needs to be removed from $EN2$, whereas the pairs concerning event e also need to be removed from $EN3$.

$notify(e_{\Delta 0})$ can then be defined

$$notify(e_{\Delta 0}) =_{df} \mathbf{Skip} \ ; \ InstEApp(e) \ ; \ EveUpd0(e)$$

Now we consider the definition of $notify(e_{\Delta 1})$. It generates event e and this event e will be active after one micro time unit. Firstly we can give the definition for function $EveUpd\Delta(e)$. It models the behaviour that event e needs to be added to $EN2$, while removing the event e related pairs from $EN3$. This is reflected by the functions $f2$ and $g2$.

$$EveUpd\Delta(e) =_{df} \mathbf{H}\left(\mathbf{true} \vdash \left(\begin{array}{c} \neg wait' \wedge same(tr1, tr2, ttr, X, RQ) \\ \wedge EN2' = f2(EN2, e) \\ \wedge EN3' = g2(EN3, e) \end{array}\right)\right)$$

where $f2(A, e) =_{df} A \cup \{e\}$

$g2(A, e) =_{df} \{(x, T) \mid (x, T) \in A \wedge x \neq e\}$

Different from $notify(e_{\Delta 0})$, the execution of $notify(e_{\Delta 1})$ only makes the changes for variable $EN2$ and $EN3$, while leaving other variables unchanged. The update of $EN2$ and $EN3$ is reflected by the two functions $f2$ and $g2$. Then $notify(e_{\Delta 1})$ can be defined

$$notify(e_{\Delta 1}) =_{df} \mathbf{Skip} \ ; \ EveUpd\Delta(e)$$

For the definition of $notify(e_{\#T})$, we first give the definition for formula $EveUpd\#((e, T))$ below.

$$EveUpd\#((e,T)) =_{df} \mathbf{H}\left(\mathbf{true} \vdash \left(\begin{array}{c} \neg wait' \wedge same(tr1, tr2, ttr, X, RQ) \\ \wedge EN2' = f3(EN2, e, T) \\ \wedge EN3' = g3(EN2, EN3, e, T) \end{array}\right)\right)$$

where

$$f3(A,e,T) =_{df} A$$

$$g3(A,B,e,T) =_{df} \begin{cases} B \cup \{(e,T)\} & \text{if } e \notin A \wedge \forall T1 \in N \bullet (e,T1) \notin B \\ B & \text{if } e \in A \text{ or } \exists T1 \in N \bullet T1 \leq T \wedge (e,T1) \in B \\ B \backslash \{(e,T3) \mid \exists T3 \in N \bullet (e,T3) \in B\} \cup \{(e,T)\} & \text{if } \exists T2 \in N \bullet T2 > T \wedge (e,T2) \in B \end{cases}$$

The behaviour of macro time event notification $notify(e_{\#T})$ is mainly represented by the two functions $f3$ and $g3$ via formula $EveUpd\#((e,T))$. Macro time event notification does not affect $EN2$. However, it affects $EN3$, which can be dealt with in several cases shown above.

The behaviour of function $g3(EN2, EN3, e, T)$ can be classified into three cases. The first expresses the case that $EN2$ does not contain event e and $EN3$ does not contain event e related pairs. Then the result of this macro time event notification simply adds the pair (e,T) to $EN3$. The second expresses the case that either e is already in $EN2$ or there exists event e related pairs whose macro time stamp is not greater than T in $EN3$. For this case, $EN3$ remains unchanged. Furthermore, if there exist event e related pairs whose macro time stamp is greater than T, then $EN3$ needs to be modified. For this case, event e related pairs need to be removed from $EN3$, and the pair (e,T) needs to be added.

Then $notify(e_{\#T})$ can be defined as below:

$$notify(e_{\#T}) =_{df} \textbf{Skip} \ ; \ EveUpd\#((e,T))$$

Finally we consider the event cancel statement $cancel(e)$. The cancellation is mainly represented by formula $EveUpd0(e)$.

$$cancel(e) =_{df} \textbf{Skip} \ ; \ EveUpd0(e)$$

3.4. Event Waiting

This section considers the semantics of the event waiting statement. Firstly, we give some preliminary definitions.

$$attach =_{df} \mathbf{H}\left(\textbf{true} \vdash \left(\begin{array}{l} \neg wait' \wedge ttr = null \wedge \\ same(tr2, X, RQ, EN2, EN3) \wedge \\ \\ tr1' = tr1 \lhd ttr = null \ \vee \ last(y) = ttr \rhd \left(\begin{array}{c} y' = y^\frown\langle(ttr,1)\rangle \wedge \\ len(tr1' - tr1) = 0 \wedge \\ len((tr1' - tr1)[0]) = 0 \end{array}\right) \end{array}\right)\right)$$

where $y = last(last(tr1))$ and $y' = last(last(tr1'))$ in the above formula. The trace variable $tr1$ is in the form of three dimensional structure. Here y stands for the one dimensional trace for variable $tr1$ at the last macro time and micro time point. Formula $y' = y^\frown\langle(ttr,1)\rangle$ indicates that snapshot $\langle(ttr,1)\rangle$ is attached to the end of y. The purpose of the behaviour of $attach$ is to append the contribution stored in ttr to the end of trace variable $tr1$.

Next we define $update(RQ)$, which is used to generate events from sequence RQ. The generated events will be appended to the end of trace variable $tr2$. $update(s)$ can be defined as

if $s = \langle\rangle$, then $update(s) =_{df} II$

otherwise,

$$update(s) =_{df} \mathbf{H}\left(\textbf{true} \vdash \left(\begin{array}{c} \neg wait' \wedge s' = tail(s) \wedge \\ (\bigvee_{i\in\{1,2,3\}} CompAtt(s,i)) \wedge \\ same(tr1, ttr, X, EN2, EN3) \end{array}\right)\right) \ ; \ update(s)$$

where $tail(s)$ stands for the sequence s but the first element.

Here, $CompAtt(s, op)$ can be defined as:

$$
\begin{array}{l}
(1)\ \text{if } op = 1, \quad \text{then} \\
\qquad CompAtt(s, op) \\
\quad =_{df}\ ttr(\pi_1(head(s))) < \pi_2(head(s)) \wedge y' = y^\frown\langle(pe(\pi_1(head(s))), 1)\rangle \wedge \\
\qquad ttr' = ttr[\pi_2(head(s))/\pi_1(head(s))] \\
(2)\ \text{if } op = 2, \quad \text{then} \\
\qquad CompAtt(s, op) \\
\quad =_{df}\ ttr(\pi_1(head(s))) > \pi_2(head(s)) \wedge y' = y^\frown\langle(ne(\pi_1(head(s))), 1)\rangle \wedge \\
\qquad ttr' = ttr[\pi_2(head(s))/\pi_1(head(s))] \\
(3)\ \text{if } op = 3, \quad \text{then} \\
\qquad CompAtt(s, op) \\
\quad =_{df}\ ttr(\pi_1(head(s))) = \pi_2(head(s)) \wedge (tr2' = tr2) \wedge (ttr' = ttr)
\end{array}
$$

where, $y' = last(last(tr2'))$ and $y = last(last(tr2))$ in the above definition. Here $head(s)$ stands for the first element of sequence s.

The behaviour of $CompAtt(s, op)$ is to generate the exact event based on the two values recorded in ttr and the first element of trace s for the corresponding channel. If the first value is less than the second, a positive edge event on the channel will be generated. Inversely a negative edge event will be generated. Further, if the two values are the same, no event will be generated.

For $update(RQ)$, the consideration for the update needs to go through all the pairs in sequence RQ. For each pair, the update might generate events which will be added to the end of the trace variable $tr2$.

Now we consider the semantics for the triggering of a single event $wait(et)$. There are two event triggering cases. The first case is the self-triggering case; i.e., the event is triggered by the process itself, which indicates that the event is generated by the most recently completed atomic action. We use formula $selftrig(et)$ to represent this case. In this case, the update based on sequence RQ needs to be executed, as well as attaching the result of the recent completed atomic action. It should also need to be judged whether the current situation belongs to the self-triggering case, which is described by formula $selfjudge$.

$$selftrig(et) =_{df} \mathbf{Skip2}\ ;\ update(RQ)\ ;\ (ttr \neq null) \wedge attach\ ;\ selfjudge(et)$$

where:

$$\mathbf{Skip2} =_{df} InstEnv2 \lhd ttr = null \rhd II$$

and

$$InstEnv2 =_{df} \mathbf{H}\left(\mathbf{true} \vdash \left(\begin{array}{l} \neg wait' \wedge ttr' = \pi_1(last(last(last(tr1)))) \\ \wedge same(\{tr1, tr2, X, RQ, EN2, EN3\}) \end{array}\right)\right)$$

and

$$selfjudge(et) =_{df} \mathbf{H}\left(\mathbf{true} \vdash \left(\begin{array}{l} \neg wait' \wedge last(last(last(tr2))) = (et, 1) \\ \wedge same(tr1, ttr, X, RQ, EN2, EN3) \end{array}\right)\right)$$

The second case is the environment triggering case; i.e., an event is generated by the environment and this event triggers the waiting behaviour. For this case, the update based on sequence RQ and the attachment for the recent atomic action need to be executed. Then the process waits for the environment to generate the event which can trigger the current waiting command. The whole behaviour can be partitioned into two phases. The first one is the waiting period described by formula $await(et)$, during which the environment can generate events and these events can not trigger our waiting command. The second phase is the triggering

behaviour, described by formula $trig(et)$.

$$\begin{array}{l} \quad await(et) \\ =_{df}\ \mathbf{Skip2}\ ;\ update(RQ)\ ; \\ \quad (ttr = null \vee last(last(last(tr2))) \neq (et,1)) \wedge attach\ ;\ aawait(et) \end{array}$$

and

$$aawait(et) =_{df} \mathbf{H}\left(\mathbf{true} \vdash \left(\begin{array}{c} \forall i,j \bullet et \notin \pi_1((tr2' - tr2)[i][j]) \wedge \\ \bigwedge_{x \in \{tr1,tr2\}} Env(x' - x) \wedge \\ same(RQ, X, ttr, EN2, EN3) \end{array}\right)\right)$$

$$trig(et) =_{df} \mathbf{H}\left(\mathbf{true} \vdash \left(\begin{array}{c} Instenv(tr2' - tr2) \wedge \\ last(last(last(tr2' - tr2))) = \langle (et,0) \rangle \wedge \\ same(tr1, ttr, RQ, EN2, EN3) \end{array}\right)\right)$$

We can define

$$wait(et) =_{df} selftrig(et)\ \vee\ (await(et)\ ;\ trig(et))$$

Next we consider the semantics of compound event "**or**". Let

$$selfjudge(\mathbf{or}_{i \in I}\{et_i\}) =_{df} \mathbf{H}\left(\mathbf{true} \vdash \left(\begin{array}{c} \neg wait' \wedge same(tr1, X, ttr, EN2, EN3) \\ \wedge\ (\bigvee_{i \in I} last(last(last(tr2))) = (et_i, 1)\ ) \end{array}\right)\right)$$

$$aawait(\mathbf{or}_{i \in I}\{et_i\}) =_{df} \mathbf{H}\left(\mathbf{true} \vdash \left(\begin{array}{c} \forall i,j,k \bullet et_i \notin \pi_1((tr2' - tr2)[j][k]) \wedge \\ \bigwedge_{x \in \{tr1,tr2\}} Env(x' - x) \wedge \\ same(RQ, X, ttr, EN2, EN3) \end{array}\right)\right)$$

$$trig(\mathbf{or}_{i \in I}\{et_i\}) =_{df} \mathbf{H}\left(\mathbf{true} \vdash \left(\begin{array}{c} Instenv(tr2' - tr2) \wedge \\ \bigvee_{i \in I}(\ last(last(last(tr2' - tr2))) = \langle (et_i, 0) \rangle\ ) \wedge \\ same(tr1, ttr, RQ, EN2, EN3) \end{array}\right)\right)$$

We can define

$$\begin{array}{l} \quad wait(\mathbf{or}_{i \in I}\{et_i\}) \\ =_{df}\ selftrig(\mathbf{or}_{i \in I}\{et_i\})\ \vee\ (\ await(\mathbf{or}_{i \in I}\{et_i\})\ ;\ trig(\mathbf{or}_{i \in I}\{et_i\})\ ) \end{array}$$

For time delay statements, we first consider the Δ delay (micro time delay).

$$hold\Delta(0) =_{df} \mathbf{H}\left(\mathbf{true} \vdash \left(\begin{array}{c} \neg wait' \wedge \bigwedge_{x \in \{tr1,tr2\}} Instenv(x' - x) \wedge \\ same(ttr, X, RQ, EN2, EN3) \end{array}\right)\right)$$

$$\mathbf{phase}\Delta =_{df} \mathbf{H}\left(\mathbf{true} \vdash \left(\begin{array}{l} same(ttr, X, RQ, EN2, EN3) \wedge \\ \bigwedge_{x \in \{tr1,tr2\}} len(x' - x) = 0 \wedge \\ \left(\begin{array}{l} (tr1' = tr1 \wedge tr2' = tr2)\ \lhd wait' \rhd \\ \left(\begin{array}{l} \bigwedge_{x \in \{tr1,tr2\}} (len((x' - x)[0]) = 1 \wedge \\ \quad len((x' - x)[0][0]) = 0 \wedge len((x' - x)[0][1]) = 0) \end{array}\right) \end{array}\right) \end{array}\right)\right)$$

$hold\Delta(0)$ stands for the environment behaviours at the current micro time point, i.e., the environment can

generate new states and new events at the current micro time point. **phase**Δ purely represents the one unit micro time advancing, ie., there are no new generated states and events during the pure micro time advancing.

Next we introduce formula $Wupd\Delta$. $Wupd\Delta$ indicates that a sequence of events will be attached to the end of trace $tr2$ at the current time point. These sequences are the permutations of all the events recorded in $EN2$, expressed using formula "$permu(EN2)$".

$$Wupd\Delta =_{df} \mathbf{H}\left(\mathbf{true}\left(\begin{array}{c}\neg wait' \wedge len(tr2'-tr2)=0 \wedge \\ len((tr2'-tr2)[0])=0 \wedge \\ \pi_1((tr2'-tr2)[0][0]) \in permu(EN2) \wedge \\ \pi_2((tr2'-tr2)[0][0]) \in 1^* \wedge \\ same(ttr,tr1,X,RQ,EN3) \wedge EN2'=\emptyset\end{array}\right)\right)$$

where $permu(A)$ stands for the set containing all permutations of set A.

Based on the above formalizations for $UpdAtt$, $hold\Delta(1)$ and $Wupd\Delta$, we can give the definition for $wait(\Delta 1)$.

$$wait(\Delta 1) =_{df} UpdAtt \ ; \ hold\Delta(0) \ ; \ \mathbf{phase}\Delta \ ; \ Wupd\Delta$$

where $UpdAtt =_{df} \mathbf{Skip2} \ ; \ update(RQ) \ ; \ attach.$

Next we consider the semantics of macro-time delay. Firstly, we introduce formula $hold\#(n)$. It models the behaviour that macro time can advance n time units. If time has not advanced n units, the process is still at the waiting state. Otherwise, the process is at the terminating state. During the time advancing period, only the environment can generate new states or new events.

$$hold\#(n) =_{df} \mathbf{H}\left(\mathbf{true} \vdash \left(\begin{array}{l}wait' \wedge len(tr1'-tr1)<n \wedge Env(tr1'-tr1) \wedge \\ len(tr2'-tr2)<n \wedge Env(tr2'-tr2) \wedge \\ same(ttr,X,RQ,EN2,EN3) \\ \vee \\ \neg wait' \wedge len(tr1'-tr1)=n \wedge Env(tr1'-tr1) \wedge \\ len(tr2'-tr2)=n \wedge Env(tr2'-tr2) \wedge \\ same(ttr,X,RQ,EN2,EN3)\end{array}\right)\right)$$

The formula **phase**# purely represents the one unit macro time advancing, ie., there are no new generated states and events during the pure macro time advancing.

$$\mathbf{phase}\# =_{df} \mathbf{H}\left(\mathbf{true} \vdash \left(\begin{array}{l}same(ttr,X,RQ,EN2,EN3) \wedge \\ \left(\begin{array}{l}(tr1'=tr1 \wedge tr2'=tr2) \ \lhd wait' \rhd \\ \left(\begin{array}{l}\bigwedge_{x\in\{tr1,tr2\}}(len(x'-x)=1 \wedge \\ \quad len((x'-x)[0])=0 \wedge len((x'-x)[0][0])=0 \wedge \\ \quad len((x'-x)[1])=0 \wedge len((x'-x)[1][0])=0)\end{array}\right)\end{array}\right)\end{array}\right)\right)$$

After n macro time units elapse, new events need to be attached to the end of trace $tr2$ at the current micro time point. These events are taken from the pairs in $EN3$ whose time stamp is n. We use $Wupd\#(n)$ to model these behaviours.

$$Wupd\#(n) =_{df} \mathbf{H}\left(\mathbf{true} \vdash \left(\begin{array}{l} len(tr2' - tr2) = 0 \wedge len((tr2' - tr2)[0]) = 0 \wedge \\ \pi_1((tr2' - tr2)[0][0]) \in permu(\{e \mid (e,n) \in EN3\}) \wedge \\ \pi_2((tr2' - tr2)[0][0]) \in 1^* \wedge same(tr1, ttr, X, RQ, EN2) \wedge \\ EN3' = \{(e,T) \mid (e,n+T) \in EN3\} \end{array}\right)\right)$$

Based on the above definitions, we can define macro-time delay.

$$wait(n) =_{df} UpdAtt \ ; \ hold\#(n-1) \ ; \ \mathbf{phase}\# \ ; \ Wupd\#(n)$$

3.5. Parallel Composition

For defining parallel composition, we first provide several merge functions.

$$pmerge(s,t,u) =_{df} \left(\begin{array}{c} \pi_1(s[0..len(t)-1]) = \pi_1(t[0..len(t)-1]) \wedge \\ \pi_1(s[0..len(t)-1]) = \pi_1(u[0..len(t)-1]) \wedge \\ (\ \pi_2(u[0..len(t)-1]) = \pi_2(s[0..len(t)-1]) + \\ \qquad\qquad \pi_2(t[0..len(t)-1])\ ) \wedge \\ 2 \notin \pi_2(u[0..len(t)-1]) \wedge len(u) = len(s) \end{array}\right)$$

where $\langle i_1, \ldots, i_n\rangle + \langle j_1, \ldots, j_n\rangle =_{df} \langle (i_1 + j_1), \ldots, (i_n + j_n)\rangle$

A snapshot is expressed as a pair (σ, f). The first two lines indicate that the sequence of the states (or events) for a parallel process is the same as the sequence of states (or events) for its two components. The third and fourth lines inform that any state contributed by a parallel process is actually the contribution by one of its components. These two lines also indicate that any state (or event) contributed by the environment of a parallel process cannot be the contribution of either of its components. The fifth line means that any state contributed by a parallel process cannot be contributed by both of its components.

$pmerge(s,t,u)$ is to merge two sequences s and t, the result is stored in sequence u. Here, s and t are one-dimensional sequences; i.e., the sequence of type $tr1$ (or $tr2$) at some micro time points. For $pmerge(s,t,u)$, the length of sequence s is greater than or equal to the length of sequence t.

Next we introduce $merge(s,t,u)$, which merges two sequences s and t into one single sequence. Its definition is based on the above $pmerge$ function. For $merge(s,t,u)$, there are no length restrictions on sequences s and t. If $len(s) > len(t)$, function $pmerge(s,t,u)$ is called. The rest elements of s will be directly added to the current sequence u. If $len(s) = len(t)$, only function $pmerge(s,t,u)$ is called. On the other hand, if $len(t) > len(s)$, function $pmerge(t,s,u)$ is called. The rest elements of t will be directly added to the current sequence u.

$$merge(s,t,u) =_{df} \left(\begin{array}{l} len(s) > len(t) \Rightarrow \left(\begin{array}{l} pmerge(s,t,u) \wedge \\ u[len(t)..len(s)-1] = s[len(t)..len(s)-1] \end{array}\right) \wedge \\ len(s) = len(t) \Rightarrow pmerge(s,t,u) \wedge \\ len(s) < len(t) \Rightarrow \left(\begin{array}{l} pmerge(t,s,u) \wedge \\ u[len(s)..len(t)-1] = t[len(s)..len(t)-1] \end{array}\right) \end{array}\right)$$

Now we introduce additional merge behaviour. $Pmerge(s,t,u)$ merges two sequences s and t into one single sequence u, where the types of these sequences are of $tr1$ and $tr2$. Similarly, the length of s is also greater than or equal to the length of t.

$$Pmerge(s,t,u)$$
$$=_{df} \forall i \bullet 0 \le i \le len(t) - 1 \Rightarrow$$

$$\left(\begin{array}{l} len(s[i]) > len(t[i]) \Rightarrow \left(\begin{array}{l} \forall j \bullet 0 \le j \le len(t[i]) - 1 \Rightarrow \\ \quad merge(s[i][j], t[i][j], u[i][j]) \wedge \\ u[i][len(t[i])..len(s[i]) - 1] = \\ \quad s[i][len(t[i])..len(s[i]) - 1] \end{array}\right) \wedge \\ len(s[i]) = len(t[i]) \Rightarrow \left(\begin{array}{l} \forall j \bullet 0 \le j \le len(t[i]) - 1 \Rightarrow \\ \quad merge(s[i][j], t[i][j], u[i][j]) \end{array}\right) \wedge \\ len(s[i]) < len(t[i]) \Rightarrow \left(\begin{array}{l} \forall j \bullet 0 \le j \le len(s[i]) - 1 \Rightarrow \\ \quad merge(t[i][j], s[i][j], u[i][j]) \wedge \\ u[i][len(s[i])..len(t[i]) - 1] = \\ \quad t[i][len(s[i])..len(t[i]) - 1] \end{array}\right) \end{array}\right)$$

Based on $Pmerge()$", we introduce $Merge(s, t, u)$. It has similar behaviours as function $merge(s, t, u)$. The difference is that the types of sequence s and t here are of $tr1$ and $tr2$.

$$Merge(s, t, u) =_{df} \left(\begin{array}{l} len(s) > len(t) \Rightarrow \left(\begin{array}{l} Pmerge(s, t, u) \wedge \\ u[len(t)..len(s) - 1] = s[len(t)..len(s) - 1] \end{array}\right) \wedge \\ len(s) = len(t) \Rightarrow Pmerge(s, t, u) \quad \wedge \\ len(s) < len(t) \Rightarrow \left(\begin{array}{l} Pmerge(t, s, u) \wedge \\ u[len(s)..len(t) - 1] = t[len(s)..len(t) - 1] \end{array}\right) \end{array}\right)$$

Finally we introduce the merge operator $\otimes$ for two behaviours P and Q. Its definition is based on the above $Merge$ function. Function $Merge(tr1_P, tr1_Q, tr1)$ merges the two traces $tr1$ of both P and Q and the resulted trace is the $tr1$ trace of the whole system. The behaviour of $Merge(tr2_P, tr2_Q, tr2)$ is to generate the $tr2$ trace of the whole system from its two components.

$$\begin{array}{l} P \otimes Q \\ =_{df} \exists\, tr1_P, tr2_P, tr1_Q, tr2_Q, ttr_P, ttr_Q, EN2_P, EN3_P, EN2_Q, EN3_Q, RQ_P, RQ_Q \bullet \\ \qquad \left(\begin{array}{ll} P[tr1_P, tr2_P, ttr_P, RQ_P, EN2_P, EN3_P / & \\ \quad tr1, tr2, ttr, RQ, EN2, EN3] & \wedge \\ Q[tr1_Q, tr2_Q, ttr_Q, RQ_Q, EN2_Q, EN3_Q / & \\ \quad tr1, tr2, ttr, RQ, EN2, EN3] & \wedge \\ Merge(tr1_P, tr1_Q, tr1) & \wedge \\ Merge(tr2_P, tr2_Q, tr2) & \wedge \\ RQ' = \langle\rangle \wedge EN2' = \emptyset & \wedge \\ EN3' = EN3_P \cup EN3_Q & \end{array}\right) \end{array}$$

We are now ready to define the denotational semantics for $P \parallel Q$ by considering the divergent, waiting and terminating behaviours of $P \parallel Q$.

- It stays at a waiting state if either component does so.

$$\begin{array}{rl} wait(P \parallel Q)) =_{df} & (\ wait(P) \otimes wait(Q) \ \vee \ wait(P) \otimes ter(Q) \vee \\ & \quad ter(P) \otimes wait(Q)\ ) \end{array}$$

- It terminates when both components complete their execution.

$$ter(P \parallel Q) =_{df} (\ ter(P) \otimes ter(Q)\ )$$

- It behaves chaotically when either component is divergent.

$$
\begin{array}{l}
\quad div(P \parallel Q) \\
=_{df} (\ div(P) \otimes div(Q)\ \vee\ div(P) \otimes wait(Q)\ \vee\ div(P) \otimes ter(Q)\ \vee \\
\qquad div(Q) \otimes wait(P)\ \vee\ div(Q) \otimes ter(P)\ )
\end{array}
$$

3.6. Algebraic Laws for Sequential Programs

Algebra is well-suited for direct use by engineers in symbolic calculation of parameters and the structure of an optimal design [HHH+87, HH98]. This section aims to explore a set of algebraic laws for SystemC. These laws can be verified with respect to the semantics given in the above subsections.

For assignment, conditional, iteration, nondeterministic choice and sequential composition, our language enjoys similar algebraic properties as those reported in [He94, HH98]. In the follows, we shall only focus on novel algebraic properties for sequential programs of SystemC. We leave the investigation for parallel expansion laws in the next section by introducing an extra operator named "*guarded choice*" with location status.

3.6.1. Channel Statements

The behaviour of the channel input statement $ch??v$ is to assign the current value of ch to variable v, which has no effect on channel ch. So the algebraic laws associated with channel input statements are similar to those associated with assignments.

The channel output statement is executed during the evaluation phase of a delta-cycle. The new value will not be available to be read until the next delta-cycle.

L1 $ch!!exp\ ;\ S = S\ ;\ ch!!exp$

where $S \in \{\ \mathbf{Skip},\ x := exp,\ ch??x,\ notify(e_{\Delta 0}),\ notify(e_{\Delta 1}),\ notify(e_{\sharp T}),\ cancel(e)\ \}$

If multiple channel output statements occur to the same channel, the last statement executed determines the new value of the channel.

L2 $ch!!exp\ ;\ ch1!!exp1\ ;\ ch!!exp' = ch1!!exp1\ ;\ ch!!exp'$, where $ch \neq ch1$.

From L1 and L2, we can have:

- For each channel, at most one output statement takes effect in an atomic action.

3.6.2. Event Statements

Events are used to synchronize concurrent processes. Therefore, the execution order between statements dealing with events and statements dealing with variables and channels can be swapped in an atomic action.

L1 $S1; S2 = S2; S1$, where

$S1 \in \{notify(e_{\Delta 0}), notify(e_{\Delta 1}), notify(e_{\sharp T}), cancel(e)\}$,

$S2 \in \{\mathbf{Skip}, x := exp, ch??x, ch!!exp\}$

The effect of delayed notifications does not occur immediately, so the order of delayed notifications on different events can be changed in an atomic action.

L2 $notify(e_{DT1})\ ;\ notify(f_{DT2}) = notify(f_{DT2}); notify(e_{DT1})$

where $DT1 \in \{\Delta 0, \Delta 1, \sharp T\}$, $DT2 \in \{\Delta 1, \sharp T\}$

An immediate notification can override the pending notification on the same event.

L3 $notify(e_{DT}); notify(e_{\Delta 0}) = notify(e_{\Delta 0})$, where $DT \in \{\Delta 1, \sharp T\}$

Only pending notifications can be cancelled. At any moment, at most one pending notification can exist for one event.

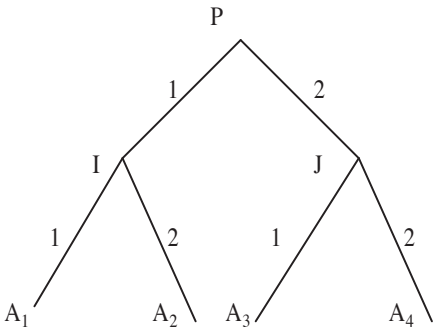

Fig. 2.

L4 (1) $notify(e_{\Delta 0}); cancel(e) = notify(e_{\Delta 0})$
(2) $notify(e_{DT}); cancel(e) = cancel(e)$
(3) $cancel(e); cancel(e) = cancel(e)$
where $DT \in \{\Delta 1, \sharp T\}$

More than one delayed notification on the same event override each other and the one scheduled to occur earlier overrides that scheduled to occur later. Delta-cycle delayed notifications are scheduled to occur earlier than timed notifications.

L5 (1) $notify(e_{DT1}); notify(e_{DT2}) = notify(e_{DT2});\ notify(e_{DT1}) = notify(e_{\Delta 1})$
(2) $notify(e_{\sharp T1}); notify(e_{\sharp T2}) = notify(e_{\sharp T2});\ notify(e_{\sharp T1}) = notify(e_{\sharp T1})$
where $T1 \leq T2$, $DT1, DT2 \in \{\Delta 1, \sharp T\}$ and $(DT1 = \Delta 1) \vee DT2 = \{\Delta 1\}$.

From the above laws, we can have:

- For each event, at most one delta-cycle delayed notification takes effect during one delta-cycle.
- For each event, at most one timed delayed notification takes effect during one simulation time unit.

4. Algebraic Derivation of Denotational Semantics

4.1. Location Status and Types of Guarded Choice

Example 4.1
Let $P = I \parallel J$, $I = A_1 \parallel A_2$ and $J = A_3 \parallel A_4$, where $A_i = notify(e_{i\Delta 0})\ ;\ notify(f_{i\Delta 0})\ ;\ u_i := u_i + 1\ ;\ v_i := v_i + 2$ $(i = 1, 2, 3, 4)$. Figure 2 is the graph that illustrates the structure of P. The behaviour of A_i forms an atomic action. If $notify(e_{i\Delta 0})$ in A_1 is scheduled, A_i $(i = 2, 3, 4)$ cannot be scheduled until the completion of the execution of the statements in A_1. In order to support the parallel expansion laws, we introduce the concept of locality (i.e., location status). For example, if $notify(e_{i\Delta 0})$ is scheduled, we want the expansion laws to correctly indicate the next behaviour should be $notify(f_{1\Delta 0})$, i.e., all $notify(e_{i\Delta 0})$ $(i = 2, 3, 4)$ cannot be scheduled at this moment.

In order to solve this, now we assign a label for each edge. If it is the left edge, the label is 1, otherwise the label is 2. For every point, its thread sequence is the label sequence from the root of the tree to the considered point. This sequence can index the exact component an instantaneous is due to. For example, if the instantaneous action is due to process A_1, the thread sequence is $\langle 1 \rangle^\frown \langle 1 \rangle$. Further, if the instantaneous action is due to process A_2, the thread sequence is $\langle 1 \rangle^\frown \langle 2 \rangle$. □

Now we introduce the concept of location status for a program, which is one of the following two forms:

(1) *index*, which can be $\langle\rangle$ or a non-empty thread sequence.

(2) *null*, which indicates a process is at the state, where the atomic action completes its execution. Further, the environment can get the chance to perform its instantaneous action.

For the aim of linking the various semantics of SystemC, we introduce the concept of guarded choice. A guarded choice is composed of a set of guarded components. The introduction of guarded choice is to support the parallel expansion laws. Guarded choice can be formalized with location status (i.e., *tag*), which is defined as below.

Definition 4.2

(1) h (P, tag) is a guarded component if it can be one of the forms below. Here, b is a Boolean condition and *index* can be $\langle\rangle$ or a non-empty thread sequence.

$$V\ (P, index), \quad wait(e)\ (P, null), \quad \#1\ (P, null), \quad \Delta 1\ (P, null)$$

where V can be one of the following forms:

$$b\&(x := e), \quad ch!!exp, \quad ch??v,$$
$$notify(e_{\Delta 0}), \quad notify(e_{\Delta 1}), \quad notify(e_{\Delta -1}), \quad notify(e_{\#T})$$

(2) $[\![\{h_1\ (P_1, tag_1),\ \ldots,\ h_n\ (P_n, tag_n)\}$ is a guarded choice if every element $h_i\ (P_i, tag_i)$ is a guarded component. □

In the above definition, the guarded component $V\,(P, index)$ indicates that the instantaneous action V will be executed. After the execution, the subsequent program P will be at the location status *index*. For the event waiting component (i.e., $wait(e)$ (P, *null*)) and delay guarded component (i.e., $\#1$ (P, *null*) and $\Delta 1$ (P, *null*)), after the firing of event guard or time elapsing, the subsequent behaviour should be the location status *null*.

Guarded choice can be divided into five types. The first type of guarded choice is composed of some instantaneous actions including assignment, channel output, channel input, and event notifications. Which one is selected to execute is nondeterministic. Its following behavior is described at the location status *index*.

(type-1) $[\![_{i\in I}\{b_i\&(x_i := e_i)\ (P_i, index_i)\}[\![[\![_{j\in J}\{ch_j\ !!\ exp_j\ (Q_j, index_j)\}$
$[\![[\![_{k\in K}\{ch_k\ ??\ v_k\ (R_k, index_k)\}[\![[\![_{l\in L}\{notify(e_{l_x})\ (T_l, index_l)\}$

The second type of guarded choice is only composed of a set of event guard components. Assume that all the guard events are different from each other. Any can be fired when the corresponding event happens. After the event guard is fired, its subsequent behavior is at the location status *null*.

(type-2) $[\![_{i\in I}\{wait(e_i)\ (P_i, null)\}$

The third type of guarded choice has one delta-cycle time delay (i.e., one micro time delay) or one macro time delay component. After the elapsing of the corresponding one time unit, the following behavior will be described as P at the location status *null*.

(type-3) (1) $[\![\{\Delta 1\ (P, null)\}$
(2) $[\![\{\#1\ (P, null)\}$

The fourth type of guarded choice is composed of a set of instantaneous action components and a set of event guard components. The whole process waits for any of the event guards to be triggered and any of the instantaneous actions can also have chances to be scheduled.

(type-4) $[\![_{i\in I}\{b_i\&(x_i := e_i)\ (P_i, index_i)\}[\![[\![_{j\in J}\{ch_j\ !!\ exp_j\ (Q_j, index_j)\}$
$[\![[\![_{k\in K}\{ch_k\ ??\ v_k\ (R_k, index_k)\}[\![[\![_{l\in L}\{notify(e_{l_x})\ (T_l, index_l)\}$
$[\![[\![_{m\in M}\{wait(e_m)(S_m, null)\}$

The fifth type of guarded choice is composed of a set of event guard components and a time delay component. The process waits for any of the event guards to be fired at the current time point. Time will elapse one delta-cycle time unit or one macro time unit when there are no more event guards to be triggered.

(type-5) (1) $[\![_{i\in I}\{wait(e_i)\ (P_i, null)\}[\![[\![\{\Delta 1\ (Q, null)\}$

(2) $[\!]_{i\in I}\{wait(e_i)\ (P_i, null)\}[\!][\!]\{\#1\ (Q, null)\}$

4.2. Algebraic Semantics for Parallel Construct

In this section we explore the algebraic laws for SystemC. We mainly focus on the laws for parallel composition. Parallel composition is symmetric and associative. As our parallel model is an interleaving model, the laws below indicate how a parallel process can be sequentialized. Our algebraic laws below are expressed in the form $(P, tag) = (Q, tag)$, indicating that programs P and Q behave the same at the location status tag. For simplicity, $(P, tag) = (Q, tag)$ is also written as $P =_{tag} Q$.

Firstly we define two functions $\mathbf{par}(P, Q)$ and $\mathbf{par1}(P, Q, i, index)$, which can reduce the number of parallel expansion laws by covering several cases at the same time. We use ε to stand for the empty process.

$$\mathbf{par}(P,Q) =_{df} \begin{cases} (\varepsilon,\ null) & \text{if } P=\varepsilon \text{ and } Q=\varepsilon \\ (P \parallel Q,\ null) & \text{otherwise} \end{cases}$$

$$\mathbf{par1}(P,Q,i,index) =_{df} \begin{cases} (\varepsilon,\ null) & \text{if } P=\varepsilon \text{ and } Q=\varepsilon & (1)\\ (\varepsilon \parallel Q,\ null) & \text{if } P=\varepsilon \text{ and } Q\neq\varepsilon \text{ and } i=1 & (2)\\ (P \parallel \varepsilon,\ null) & \text{if } P\neq\varepsilon \text{ and } Q=\varepsilon \text{ and } i=2 & (3)\\ (P \parallel Q,\ \langle 1\rangle^\frown index) & \text{if } P\neq\varepsilon \text{ and } i=1 & (4)\\ (P \parallel Q,\ \langle 2\rangle^\frown index) & \text{if } Q\neq\varepsilon \text{ and } i=2 & (5) \end{cases}$$

For $\mathbf{par1}(P, Q, i, index)$, it stands for the parallel composition of $P \parallel Q$ at the corresponding location status. Here, i is used to indicate which thread is active, where "1" (or "2") indicate that the left component (or the right component) is active, and $index$ stands for the location status of the active component. The main purpose is to calculate the exact location status of $P \parallel Q$ at different cases. If P and Q are both empty processes, $P \parallel Q$ is also empty and its location status is $null$. The first line represents this case. For $P \parallel Q$, if the left (or right) hand side completes a sequence of instantaneous actions and becomes empty, the whole process should still be written in the form of parallel composition and the location status is $null$. The second line and third line represent this case. Furthermore, if one component of $P \parallel Q$ is executing instantaneous actions and has not reached to the empty state, the location status of $P \parallel Q$ is expressed as $\langle i\rangle^\frown index$. The fourth line and fifth line represent this case.

In the following algebraic laws, U_i and V_j stand for the instantaneous actions and t can be $\Delta 1$ or $\#1$. The notation (par-i-j) stands for the parallel expansion laws whose two parallel components are of type i and type j. The first five laws stand for the case that the first component of a parallel process is of type one.

If the second component of a parallel process is a guarded choice of a set of instantaneous actions, the behavior of the parallel process can be described as the guarded choice of a set of instantaneous components. The behavior after the selected instantaneous action is the parallel composition of the subsequent process with the other parallel branch. This case is expressed in law (par-1-1).

(par-1-1) Let $P =_{null} [\!]_{i\in I}\{U_i\ (P_i, index_i)\}$ and $Q =_{null} [\!]_{j\in J}\{V_j\ (Q_i, index_j)\}$
Then $P \parallel Q$
$=_{null} [\!]_{i\in I}\{U_i\ \mathbf{par1}(P_i, Q, 1, index_i)\}\ [\!]\ [\!]_{j\in J}\{V_j\ \mathbf{par1}(P, Q_j, 2, index_j)\}$

If the second component of a parallel process is an event-guarded choice, the behavior of the parallel process can be described as the guarded choice of a set of instantaneous actions and a set of event guard components. This case is expressed in law (par-1-2).

(par-1-2) Let $P =_{null} [\!]_{i\in I}\{U_i\ (P_i, index_i)\}$ and $Q =_{null} [\!]_{j\in J}\{wait(e_j)\ (Q_j, null)\}$
Then $P \parallel Q$
$=_{null} [\!]_{i\in I}\{U_i\ \mathbf{par1}(P_i, Q, 1, index_i)\}[\!][\!]_{j\in J}\{wait(e_j)\ \mathbf{par}(P, Q_j)\}$

If the second component of a parallel process is the time delay guarded choice, only the instantaneous actions can have a chance to be scheduled. This is expressed in law (par-1-3).

(par-1-3) Let $P =_{null} [\!]_{i\in I}\{U_i\ (P_i, index_i)\}$ and $Q =_{null} [\!]\{t\ (R, null)\}$

Then $P \parallel Q \ =_{null} \ \|_{i\in I}\{U_i \ \mathbf{par1}(P_i, Q, 1, index_i)\}$

If the second component is a guarded choice comprised of a set of instantaneous actions and a set of event-guarded components, instantaneous actions from both parts can have chances to be scheduled. The event guards can also have chances to be fired. This is expressed in law (par-1-4).

(par-1-4) Let $P \ =_{null} \ \|_{i\in I}\{U_i \ (P_i, index_i)\}$ and
$Q \ =_{null} \ \|_{j\in J}\{V_j \ (Q_j, index_j)\}[]\|_{k\in K}\{wait(e_k) \ (R_k, null)\}$
Then $P \parallel Q$
$=_{null} \ \|_{i\in I}\{U_i \ \mathbf{par1}(P_i, Q, 1, index_i)\}[]\|_{j\in J}\{V_j \ \mathbf{par1}(P, Q_j, 2, index_j)\}$
$[]\|_{k\in K}\{wait(e_k) \ \mathbf{par}(P, R_k)\}$

If the second component is a guarded choice comprised of a set of event-guarded components and the time delay component, the instantaneous actions from the first parallel component can be scheduled. Meanwhile, event guards from the second parallel component can also have chances to be fired. However, as the first component has instantaneous behaviors initially, the whole system cannot make time advance initially.

(par-1-5) Let $P \ =_{null} \ \|_{i\in I}\{U_i \ (P_i, index_i)\}$ and
$Q \ =_{null} \ \|_{j\in J}\{wait(e_j) \ (Q_j, null)\}[]\|\{t \ (R, null)\}$
Then $P \parallel Q$
$=_{null} \ \|_{i\in I}\{U_i \ \mathbf{par1}(P_i, Q, 1, index_i)\}[]\|_{j\in J}\{wait(e_j) \ \mathbf{par}(P, Q_j)\}$

The next four laws stand for the case that the first component of a parallel process is of type two. If the second parallel component is also a guarded choice of a set of event guard components, the scheduling rule is arranged in the form of three types of guarded choice. The first and the second are composed of a set of event guard components of one parallel branch, where events are different from those in another parallel branch respectively. The behavior after the triggered event guard is the parallel composition of the subsequent process with another parallel part. The third type of guarded choice describes the common guard event of the two parallel parts and the subsequent behavior is defined by the parallel composition of the corresponding following processes. This case is illustrated in law (par-2-2).

(par-2-2) Let $P \ =_{null} \ \|_{i\in I}\{wait(e_i) \ (P_i, null)\}$ and $Q \ =_{null} \ \|_{j\in J}\{wait(f_j) \ (Q_j, null)\}$
Let $E = \{e_i | i \in I\}$, $F = \{f_j | j \in J\}$, $I' = \{i \mid e_i \in E \wedge e_i \notin F\}$,
$J' = \{j \mid f_j \in F \wedge f_j \notin E\}$, $IJ = \{(i, j) \mid i \in I \wedge j \in J \wedge e_i \in E \wedge f_j \in F \wedge e_i = f_j\}$
Then $P \parallel Q$
$=_{null} \ \|_{i\in I'}\{wait(e_i) \ \mathbf{par}(P_i, Q)\}[]\|_{j\in J'}\{wait(f_j) \ \mathbf{par}(P, Q_j)\}$
$[]\|_{(i,j)\in IJ}\{wait(e_i) \ \mathbf{par}(P_i, Q_j)\}$

If the second parallel part is in the form of the third, fourth or fifth type of guarded choice, the whole system can be expressed in the expansion laws shown in the following three cases.

(par-2-3) Let $P \ =_{null} \ \|_{i\in I}\{wait(e_i) \ (P_i, null)\}$ and $Q \ =_{null} \ \|\{t \ (R, null)\}$
Then $P \parallel Q \ =_{null} \ \|_{i\in I}\{wait(e_i) \ \mathbf{par}(P_i, Q)\}[]\|\{t \ \mathbf{par}(P, R)\}$

(par-2-4) Let $P \ =_{null} \ \|_{i\in I}\{wait(e_i) \ (P_i, null)\}$ and
$Q \ =_{null} \ \|_{j\in J}\{V_j \ (Q_j, index_j)\}[]\|_{k\in K}\{wait(f_k) \ (R_k, null)\}$
Then $P \parallel Q$
$=_{null} \ \|_{j\in J}\{V_j \ \mathbf{par1}(P, Q_j, 2, index_j)\}[]\|_{i\in I'}\{wait(e_i) \ \mathbf{par}(P_i, Q)\}$
$[]\|_{k\in K'}\{wait(f_k) \ \mathbf{par}(P, R_k)\}[]\|_{(i,k)\in IK}\{wait(e_i) \ \mathbf{par}(P_i, R_k)\}$

(par-2-5) Let $P \ =_{null} \ \|_{i\in I}\{wait(e_i) \ (P_i, null)\}$ and
$Q \ =_{null} \ \|_{j\in J}\{wait(f_j) \ (Q_j, null)\}[]\|\{t \ (R, null)\}$
Then $P \parallel Q$
$=_{null} \ \|_{i\in I'}\{wait(e_i) \ \mathbf{par}(P_i, Q)\}[]\|_{j\in J'}\{wait(f_j) \ \mathbf{par}(P, Q_j)\}$
$[]\|_{(i,j)\in IJ}\{wait(e_i) \ \mathbf{par}(P_i, Q_j)\}[]\|\{t \ \mathbf{par}(P, R)\}$

Now we consider the parallel expansion laws for the case that the first component of a parallel process is of the third type of guarded choice (i.e., time delay component). This can be illustrated in the next three laws shown in (par-3-3), (par-3-4) and (para-3-5). If the second component of a parallel process is also time delay, law (par-3-3) can be expressed in the following three laws representing the parallel composition of various time forms.

(par-3-3-1) Let $P =_{null} [\![\{\Delta 1\ (R, null)\}$ and $Q =_{null} [\![\{\Delta 1\ (S, null)\}$
Then $P \parallel Q =_{null} [\![\{\Delta 1\ \mathbf{par}(R, S)\}$

(par-3-3-2) Let $P =_{null} [\![\{\Delta 1\ (R, null)\}$ and $Q =_{null} [\![\{\#1\ (S, null)\}$
Then $P \parallel Q =_{null} [\![\{\Delta 1\ \mathbf{par}(R, Q)\}$

(par-3-3-3) Let $P =_{null} [\![\{\#1\ (R, null)\}$ and $Q =_{null} [\![\{\#1\ (S, null)\}$
Then $P \parallel Q =_{null} [\![\{\#1\ \mathbf{par}(R, S)\}$

If the second component is comprised of a set of instantaneous actions and a set of event-guarded components, for the whole system, instantaneous actions can be scheduled. The event guards can also have chances to be fired. Time cannot advance initially. This is expressed in law (para-3-4).

(par-3-4) Let $P =_{null} [\![\{t\ (R, null)\}$ and
$Q =_{null} [\![_{i\in I}\{V_i\ (Q_i, index_i)\}]\!]\![_{j\in J}\{wait(e_j)\ (R_j, null)\}$
Then $P \parallel Q$
$=_{null} [\![_{i\in I}\{V_i\ \mathbf{par1}\ (P, Q_i, 2, index_i)\}]\!]\{[\![_{j\in J}\{wait(e_j)\ \mathbf{par}\ (P, R_j)\}$

If the second component is comprised of a set of event-guarded components and the time delay component, all the event guards can have chances to be fired. Furthermore, time can also advance initially. As the time delays in the first and second component of a parallel process can each have two types, the time delay type of the parallel process can be expressed by using the defined function **par2**. Law (par-3-5) can illustrate this case.

(par-3-5) Let $P =_{null} [\![\{t\ (R, null)\}$ and
$Q =_{null} [\![_{j\in J}\{wait(e_j)\ (Q_j, null)\}]\!]\![\{t_S\ (S, null)\}$
Then $P \parallel Q =_{null} [\![_{j\in J}\{wait(e_j)\ \mathbf{par}\ (P, Q_j)\}]\!]\mathbf{par2}(P, Q2)$

In the above law, $Q2$ stand for the second guarded choice of Q. Function $\mathbf{par2}(P2, Q2)$ can be defined as below.

Let $P2 =_{null} [\![\{t_1\ (P',\ null)\}$ and $Q2 =_{null} [\![\{t_2\ (Q',\ null)\}$
Then

$$\mathbf{par2}(P2, Q2) =_{df} \begin{cases} [\![\{t_1\ \mathbf{par}(P', Q')\} & \text{if } t_1 = t_2 = \Delta 1\ \vee\ t_1 = t_2 = \#1 \\ [\![\{t_1\ \mathbf{par}(P', Q)\} & \text{if } t_1 = \Delta 1\ \wedge\ t_2 = \#1 \\ [\![\{t_2\ \mathbf{par}(P, Q')\} & \text{if } t_1 = \#1\ \wedge\ t_2 = \Delta 1 \end{cases}$$

The next two laws stand for the case that one component of a parallel process belongs to the form of the fourth type of guarded choice. If the second component of a parallel process also belongs to the fourth type of guarded choice, the instantaneous actions from both components can be scheduled. On the other hand, the event guards from both components can also have chances to be fired and the firing can have three cases. This can be illustrated in law (para-4-4).

(par-4-4) Let $P =_{null} [\![_{i\in I}\{U_i\ (P_i, index_i)\}]\!]\![_{j\in J}\{wait(e_j)\ (R_j, null)\}$
$Q =_{null} [\![_{k\in K}\{V_k\ (Q_k, index_k)\}]\!]\![_{l\in L}\{wait(f_l)\ (R_l, null)\}$
Then $P \parallel Q$
$=_{null} [\![_{i\in I}\{U_i\ \mathbf{par1}\ (P_i, Q, 1, index_i)\}]\!]\![_{j\in J'}\{wait(e_j)\ \mathbf{par}\ (R_j, Q)\}$
$[\![]\!][_{k\in K}\{V_j\ \mathbf{par1}\ (R, Q_k, 2, index_k)\}]\!]\![_{l\in L'}\{wait(e_l)\ \mathbf{par}\ (P, R_l)\}$

$$[\!]_{(j,l)\in JL}\{wait(e_j)\ \mathbf{par}\ (P_j, R_l)\}$$

If the second component of a parallel process belongs to the fifth type of guarded choice, the analysis is similar. As the first component of the parallel process has instantaneous actions initially, the whole system cannot make time advance initially. This is illustrated in law (par-4-5).

(par-4-5) Let $P =_{null} [\!]_{i\in I}\{U_i\ (P_i, index_i)\}[\!][\!]_{j\in J}\{wait(e_j)\ (R_j, null)\}$

$Q =_{null} [\!]_{k\in K}\{wait(f_k)\ (Q_k, null)\}[\!][\!]\{t\ (S, null)\}$

Then $P \parallel Q$

$=_{null} [\!]_{i\in I}\{U_i\ \mathbf{par1}\ (P_i, Q, 1, index_i)\}[\!][\!]_{k\in K'}\{wait(f_k)\ \mathbf{par}\ (P, Q_k)\}$

$[\!][\!]_{j\in J'}\{wait(e_j)\ \mathbf{par}\ (R_j, Q)\}[\!][\!]_{(j,k)\in JK}\{wait(e_j)\ \mathbf{par}\ (R_j, Q_k)\}$

The law below stands for the case that both of the two components of a parallel process belong to the form of the fifth type of guarded choice. Events from both of the two parallel components can have chances to be fired. The firing can be classified into three cases. Meanwhile, time can also advance initially, which is specified by function **par2**.

(par-5-5) Let $P =_{null} [\!]_{i\in I}\{wait(e_i)\ (P_i, null)\}[\!][\!]\{t_R\ (R, null)\}$

$Q =_{null} [\!]_{j\in J}\{wait(f_j)\ (Q_j, null)\}[\!][\!]\{t_S\ (S, null)\}$

Then $P \parallel Q$

$=_{null} [\!]_{i\in I}\{wait(e_i)\ \mathbf{par}\ (P_i, Q)\}[\!][\!]_{j\in J}\{wait(e_j)\ \mathbf{par}\ (P, Q_j)\}$

$[\!][\!]_{(i,j)\in IJ}\{wait(e_i)\ \mathbf{par}\ (P_i, Q_j)\}[\!]\mathbf{par2}(P2, Q2)$

Further, if one parallel part is at the state of the execution of an instantaneous action and another parallel part is of any form. Then the whole process continues the execute of the instantaneous action. The case is expressed in law (par-*II*).

(par-*II*) Let $P =_{index} [\!]\{U\ (P', index)\}$

Then $P \parallel Q =_{(1)^\frown index} [\!]\{U\ \mathbf{par1}(P', Q, 1, index)\}$

$Q \parallel P =_{(2)^\frown index} [\!]\{U\ \mathbf{par1}(Q, P', 2, index)\}$

The following five laws stand for the case that one component of a process is empty. Another component can be of any forms.

(par-*III*-1) Let $P =_{null} [\!]_{i\in I}\{U_i\ (P_i,\ index_i)\}$

Then $P \parallel \varepsilon =_{null} [\!]_{i\in I}\{U_i\ \mathbf{par1}(P_i, \varepsilon, 1,\ index_i)\}$

$\varepsilon \parallel P =_{null} [\!]_{i\in I}\{U_i\ \mathbf{par1}(\varepsilon, P_i, 2,\ index_i)\}$

(par-*III*-2) Let $P =_{null} [\!]_{i\in I}\{wait(e_i)\ (P_i,\ null)\}$

Then $P \parallel \varepsilon =_{null} [\!]_{i\in I}\{wait(e_i)\ \mathbf{par}(P_i, \varepsilon)\}$

$\varepsilon \parallel P =_{null} [\!]_{i\in I}\{wait(e_i)\ \mathbf{par}(\varepsilon, P_i)\}$

(par-*III*-3) Let $P =_{null} [\!]\{t\ (P',\ null)\}$

Then $P \parallel \varepsilon =_{null} [\!]\{t\ \mathbf{par}(P', \varepsilon)\}$

$\varepsilon \parallel P =_{null} [\!]\{t\ \mathbf{par}(\varepsilon, P')\}$

(par-*III*-4) Let $P =_{null} [\!]_{i\in I}\{U_i\ (P_i,\ index_i)\}[\!][\!]_{j\in J}\{wait(e_i)\ (Q_j,\ null)\}$

Then $P \parallel \varepsilon =_{null} [\!]_{i\in I}\{U_i\ \mathbf{par1}(P_i, \varepsilon, 1,\ index_i)\}$

$[\!][\!]_{j\in J}\{wait(e_j)\ \mathbf{par}(Q_j, \varepsilon)\}$

$\varepsilon \parallel P =_{null} [\!]_{i\in I}\{U_i\ \mathbf{par1}(\varepsilon, P_i, 2,\ index_i)\}$

$[\!][\!]_{j\in J}\{wait(e_j)\ \mathbf{par}(\varepsilon, Q_j)\}$

(par-*III*-5) Let $P =_{null} [\!]_{i\in I}\{wait(e_i)\ (P_i,\ null)\} \parallel \{t\ (Q,\ null)\}$

$$\text{Then } P \parallel \varepsilon =_{null} [\!]_{i\in I}\{wait(e_i)\ \mathbf{par}(P_i,\varepsilon)\} [\!] \{t\ \mathbf{par}(Q,\varepsilon)\}$$
$$\varepsilon \parallel P =_{null} [\!]_{i\in I}\{wait(e_i)\ \mathbf{par}(\varepsilon,P_i)\} [\!] \{t\ \mathbf{par}(\varepsilon,Q)\}$$

4.3. Head Normal Form

Now we assign every program P a head normal form at location status tag, expressed in the form $HF((P,tag))$. Our consideration for deriving denotational semantics from algebraic semantics is based on the concept of head normal form. The head normal form $HF((P,tag))$ is to make one step forward expansion for program P at the location status tag.

For an instantaneous action, its location status is $null$ or $\langle\rangle$. The location status for the remaining process (the empty process) after the first step expansion is an empty sequence.

(1) $HF((v := e, tag)) =_{df} (\ [\!]\{true\&(v := e)\ (\varepsilon,\langle\rangle)\},\ tag\ )$
$HF((\mathbf{Skip}, tag)) =_{df} (\ [\!]\{true\&(x := x)\ (\varepsilon,\langle\rangle)\}\ ,\ tag\ )$
where $tag = null$ or $\langle\rangle$.

(2) $HF((X, tag)) =_{df} (\ [\!]\{X\ (\varepsilon,\langle\rangle)\}\ ,\ tag\ )$
where $tag = null$ or $\langle\rangle$.
X can be $ch??v$, $ch!!exp$, $notify(e_{\Delta 0})$, $notify(e_{\Delta 1}$, $notify(e_{\#T})$.
$HF((cancel(e), tag)) =_{df} (\ [\!]\{notify(e_{\Delta -1})\ (\varepsilon,\langle\rangle)\},\ tag\ )$

For conditional statement, the selection for the satisfactory and unsatisfactory cases can be modeled as the **Skip** behavior. Similar analysis can also be applied to iteration.

(3) $HF((P \lhd b \rhd Q, tag)) =_{df} (\ [\!]\{b\&x := x\ (P,\langle\rangle),\ \neg b\&x := x\ (Q,\langle\rangle)\},\ tag\ )$
$HF((b * P, tag)) =_{df} (\ [\!]\{b\&x := x\ (P\ ;\ b * P,\langle\rangle)\},\ \neg b\&x := x\ (\varepsilon,\langle\rangle)\},\ tag\ )$

The head normal form of $P\ ;\ Q$ mainly depends on the head normal form of P.

(4) Assume $HF((P, tag)) = (\ [\!]_{i\in I}\{X_i\ (P_i, tag_i)\},\ tag\ )$
Then $HF((P;Q, tag)) =_{df} (\ [\!]_{i\in I}\{X_i\ (\mathbf{seq}(P_i,Q), tag_i)\},\ tag\ )$
where, $\mathbf{seq}(X,Y) =_{df} \begin{cases} Y & \text{if } X = \varepsilon \\ X;Y & \text{otherwise} \end{cases}$

Below is the definition for the head normal form of time delay and event guard.

(5) $HF((\Delta 1, tag)) =_{df} (\ [\!]\{\Delta 1\ (\varepsilon, null)\},\ tag\ )$
$HF((\#1, tag)) =_{df} (\ [\!]\{\#1\ (\varepsilon, null)\},\ tag\ )$
$HF((\#T, tag)) =_{df} (\ [\!]\{\#1\ (\#(T-1), null)\},\ tag\ )$, where $T > 1$.
$HF((wait(e), tag)) =_{df} (\ [\!]\{wait(e)\ (\varepsilon, null)\},\ tag\ )$

For a parallel process, it can be at the location status $null$ or $index$. The definition of the head normal form for a parallel process is based on its location status.

(6) $HF((P \parallel Q, null)) =_{df} (T, null)$
where, T is the result by applying the above parallel expansion laws of $HF((P, null))$ and $HF((Q, null))$ at the location status $null$.
$HF((P \parallel Q, index)) =_{df} (T, index)$
where T is the result by applying the above parallel expansion laws at the location status $index$.

The above head normal forms can be used in deriving the operational semantics from algebraic semantics for SystemC.

Example 4.3 Let $P_1 = v_1 := 1\ ;\ ; notify(e_{2\Delta 1}); notify(e_{3\#3})$,
$P_2 = v_2 := 2\ ;\ notify(f_{2\Delta 1}); notify(f_{3\#3})$,
$Q_1 = wait(e_2); wait(e_3), \quad Q_2 = wait(f_2); wait(f_3)$

Consider the head normal form for program P, where $(P_1 \parallel P_2) \parallel (Q_1 \parallel Q_2)$.

For program P, its head normal form can be described as:

$$\begin{aligned} & HF((P, null)) \\ = \ & (\, [\!]\{\, v_1 := 1\ ((P_{11} \parallel P_2) \parallel (Q_1 \parallel Q_2),\ \langle 1\rangle^\frown\langle 1\rangle), \\ & \quad v_2 := 2\ ((P_1 \parallel P_{21}) \parallel (Q_1 \parallel Q_2),\ \langle 1\rangle^\frown\langle 2\rangle), \\ & \quad wait(e_2)\ ((P_1 \parallel P_2) \parallel (wait(e_3) \parallel Q_2),\ null), \\ & \quad wait(f_2)\ ((P_1 \parallel P_2) \parallel (Q_1 \parallel wait(f_3)),\ null)\, \} \\ & \, , null\,) \end{aligned}$$

where $P_{11} = notify(e_{2\Delta 1}); notify(e_{3\#3}), \quad P_{21} = notify(f_{2\Delta 1}); notify(f_{3\#3})$

Further,

$$\begin{aligned} & HF(((P_{11} \parallel P_2) \parallel (Q_1 \parallel Q_2),\ \langle 1\rangle^\frown\langle 1\rangle)) \\ = \ & (\, [\!]\{notify(e_{2\Delta 1})\ ((notify(e_{3\#3}) \parallel P_2) \parallel (Q_1 \parallel Q_2),\ \langle 1\rangle^\frown\langle 1\rangle)\} \\ & \, ,\ \langle 1\rangle^\frown\langle 1\rangle\,) \end{aligned}$$

$$\begin{aligned} & HF(((notify(e_{3\#3}) \parallel P_2) \parallel (Q_1 \parallel Q_2),\ \langle 1\rangle^\frown\langle 1\rangle)) \\ = \ & (\, [\!]\{notify(e_{3\#3}),\ ((\varepsilon \parallel P_2) \parallel (Q_1 \parallel Q_2), \langle 1\rangle^\frown\langle 1\rangle)\} \\ & \, ,\ \langle 1\rangle^\frown\langle 1\rangle\ ) \end{aligned}$$

The analysis of the head normal forms for other programs above is similar. □

4.4. Deriving Denotational Semantics from Algebraic Semantics

In section 3, we defined the denotational semantics for each statement of SystemC. In this section we explore the derivation of denotational semantics from algebraic semantics for SystemC. The derivation strategy is explored. Our approach is based on the head normal form of each process, i.e., we have five types of guarded choices.

Let

$$C(tag) =_{df} \begin{cases} ttr = null & \text{if } tag = null \\ ttr \neq null & \text{if } tag = index \end{cases}$$

We use the notation $A((P, tag))$ to represent the derived denotational semantics from algebraic semantics for program P at the location status tag. Further, the notation $A(P)$ stands for the the derived denotational semantics from algebraic semantics for program P. In section 3, we defined the denotational semantics for SystemC. We use the notation $D(P)$ to represent the defined denotational semantics for program P.

If the head normal form of a process belongs to the first type, its denotational semantics can be described as the semantics of the instantaneous action followed by the denotational semantics of the corresponding subsequent process at the new location state. The notation $D(b\&x_i := e)$ stands for the defined denotational semantics of $x := e$ at Boolean condition b.

(1) If $HF((P, tag)) = (\quad [\!]_{i\in I}\{b_i\&(x_i := e_i)\ (P_i, index_i)\}$
$[\!][\!]_{j\in J}\{ch_j\ !!\ exp_j\ (Q_j, index_j)\}$
$[\!][\!]_{k\in K}\{ch_k\ ??\ v_k\ (R_k, index_k)\}$
$[\!][\!]_{l\in L}\{notify(e_{l_x})\ (T_l, index_l)\}$
$, tag\ )$

then

$$A((P,tag)) =_{df} C(tag) \wedge \left(\begin{array}{l} \bigvee_{i\in I}(\ D(b_i \& x_i := e_i)\ ;\ A((P_i, index_i))\ ) \\ \vee \bigvee_{j\in J}(\ D(ch_j !! exp_j)\ ;\ A((Q_j, index_j))\ ) \\ \vee \bigvee_{k\in K}(\ D(ch_k ?? v_k)\ ;\ A((R_k, index_k))\ ) \\ \vee \bigvee_{l\in L}(\ D(notify(e_{l_x}))\ ;\ A((T_l, index_l))\ ) \end{array} \right)$$

If the head normal form of a process belongs to the second type, its behaviour can be divided into two cases. The first case indicates that one of the events can be self-fired. The second case indicates that none of the events can be self-fired. Then the process will wait for any of the events to be fired. During the waiting period, none of the events can be fired. After that, one of the events will get fired. For the above two cases, if one event is fired, the subsequent behaviour will be the corresponding process at the location status *null*.

(2) If $HF((P,tag)) = (\ [\!]_{i\in I}\{wait(e_i)\ (P_i, null)\},\ \ tag\ )$

then

$$A((P,tag)) =_{df} C(tag) \wedge \left(\begin{array}{l} \bigvee_{i\in I}(selftrig(e_i)\ ;\ A((P,null))) \\ \vee \\ (await(e)\ ;\ \bigvee_{i\in I}(trig(e_i)\ ;\ A((P_i,null)))\ ) \end{array} \right)$$

where $e =_{df} \mathbf{or}_{i\in I}\{e_i\}$

Now we consider the case that the head normal form of a process belongs to the third type. The time delay can be divided into two cases; i.e., micro-time and macro-time. The process first behaves the same as the corresponding one unit time delay. After that, the behaviour can be expressed as the subsequent behaviour of the process at the location status *null*.

(3) If $HF((P,tag)) = [\!]\{\Delta 1\ (P, null)\}$,

then $A((P,tag)) =_{df} C(tag) \wedge (D(\Delta 1)\ ;\ A((P,null)))$

If $HF((P,tag)) =_{df} [\!]\{\#1\ (P,null)\}$

then $HF((P,tag)) =_{df} C(tag) \wedge (D(\#1)\ ;\ A((P,null)))$

If the head normal form of a process belongs to the fourth type. The analysis can be divided into two cases. The first case indicates that one of the events can be fired. The second case indicates that none of the elements can be self-fired. The process waits for any events to be fired and one of the events will be fired during the waiting period. The waiting period will not let macro and micro time advance. Finally, either any instantaneous action will be scheduled or one event will be fired.

(4) If $HF((P,tag)) = (\ \ [\!]_{i\in I}\{U_i\ (P_i, index_i)\}$
$[\!][\!]_{j\in J}\{wait(e_j)(Q_j, null)\}$
$, tag\ \ )$

then

$A((P,tag))$

$$=_{df} C(tag) \wedge \left(\begin{array}{l} \bigvee_{j\in J}(selftrig(e_j)\ ;\ A((Q_j,null)) \\ \vee \\ await(e) \wedge hold\Delta(0)\ ;\ \left(\begin{array}{l} \bigvee_{i\in I}(D(U_i)\ ;\ A((P_i, index_i))) \ \vee \\ \bigvee_{j\in J}(trig(e_i)\ ;\ A((Q_j,null))) \end{array} \right) \end{array} \right)$$

where $e =_{df} \mathbf{or}_{j\in J}\{e_j\}$

If the head normal form of a process belongs to the fifth type, the analysis can be proceeded according to the time delay type. If the time delay is micro time, the analysis can be divided into three cases. The first case indicates that one of the events gets self-fired. The second and third case indicates that none of the events are self-fired. The process will wait for any event to be fired. During the waiting period none of these events will be fired and the waiting period is one micro time unit long. The second case indicates that one event will be fired without micro time advancing. For the third case, time will advance one micro time unit.

(5) If $HF((P,tag)) = (\ [\!]_{i\in I}\{wait(e_i)\ (P_i,null)\}[\!][\!]\{\Delta 1\ (Q,null)\}\ ,tag\ )$

then

$$A((P,tag)) =_{df} C(tag) \wedge \left(\begin{array}{l} \bigvee_{i\in I}(selftrig(e_i)\ ;\ A((P,null))) \\ \vee \\ (await(e) \wedge hold\Delta(0)\ ;\ \bigvee_{i\in I}(trig(e_i)\ ;\ A((P_i,null)))) \\ \vee \\ (await(e) \wedge hold\Delta(0)\ ;\ \mathbf{phase}\Delta\ ;\ A((Q,null))) \end{array} \right)$$

where, $e =_{df} \mathbf{or}_{i\in I}\{e_i\}$

Further, for the fifth type of guarded choice, we explore the case where the time delay is macro. This analysis can also be divided into three cases, which are similar to micro time. For the second and third cases, the holding behaviour will change from $hold\Delta(0)$ into $hold\#(0)$. For the third type, time advancing will change from micro time into macro time.

If $HF((P,tag)) = (\ [\!]_{i\in I}\{wait(e_i)\ (P_i,null)\}[\!][\!]\{\#1\ (Q,null)\}\ ,tag\ )$

then

$$A((P,tag)) =_{df} C(tag) \wedge \left(\begin{array}{l} \bigvee_{i\in I}(selftrig(e_i)\ ;\ A((P_i,null))) \\ \vee \\ (await(e)hold\#(0)\ ;\ \bigvee_{i\in I}(trig(e_i)\ ;\ A((P_i,null)))) \\ \vee \\ (await(e) \wedge hold\#(0)\ ;\ \mathbf{phase}\#\ ;\ A((Q,null))) \end{array} \right)$$

where $e =_{df} \mathbf{or}_{i\in I}\{e_i\}$

Based on the above definitions, we now have a way to calculate the denotational semantics from algebraic semantics for SystemC.

Definition 4.4 (Calculating Denotational Semantics from Algebraic Semantics)

$$A(P) =_{df} \begin{cases} A((P,null)) & \text{if } P \text{ is parallel process} \\ A((P,null))\ \vee\ A((P,\langle\ \rangle)) & \text{otherwise} \end{cases}$$

□

We know that parallel composition can only appear as the outermost construct. Therefore, the calculation of denotational semantics from algebraic semantics can be divided into two cases, i.e., parallel process and sequence process. For a parallel process, the calculation of denotational semantics can only be at the location status *null*. For a sequential process, the calculation of denotational semantics can be at the location status *null* and $\langle\ \rangle$.

For the definition of $A(P)$, we know that it is based on the head normal form. As the calculation of head normal form is in the form of one step expansion. Hence, our methodology for calculating the denotational semantics from algebraic semantics is limited to finite programs.

5. Related Work

This paper has applied *Unifying Theories of Programming* (abbreviated as *UTP*) in formalising the denotational semantics for SystemC. *UTP* was developed by Hoare and He in 1998 [HH98]. *UTP* covers wide areas of fundamental theories of programs in a formalised style and acts as a consistent basis for the principles of programming languages. The *UTP* approach has been successfully applied in studying the semantics and algebraic laws of programming languages, including probabilistic programming, object-oriented programming, real-time systems, etc.

Probabilistic systems have been investigated using denotational [MM04] and operational [DGJP04] approaches. The probability guarded command language (PGCL) is an extension of the guarded command language with probabilistic choice. Its denotational semantics was formalised by He [HSM97] using the *UTP* approach. A set of algebraic laws was achieved based on denotational semantics. Further, Bresciani and Butterfield explored a theory of designs [BB13] based on distributions over the state space and studied the denotational semantics for PGCL. Healthiness conditions have been explored for probabilistic programs based on the concept of distributions over the state space. The *UTP* approach has been applied in object-oriented designs by He and his colleagues [HLL06]. A denotational semantics has been defined for an object-oriented language. A refinement calculus has also been explored. These refinement laws indicate the essential principles of object-oriented design. Cavalcanti *et al.* proposed the safety-critical Java memory model [CWW13], where safe and predictable dynamic memory management was explored. The semantics was formalised in the *UTP* framework. *Circus* is a specification language which can define data and behavioural aspects of systems [WC01, WC02]. Oliveira *et al.* provided a new denotational semantics for *Circus* and mechanized the semantics in a theorem prover ProofPower-Z [OCW09, OCW13], which supports automatic proof of refinement laws. Sherif *et al.* introduced *Circus Time*, a timed extension of *Circus* [SCHS10]. Its semantics was also explored by using *UTP* approach. A framework for validation of timed properties was provided, which was based on FDR, the CSP model checker.

This paper explored the denotational semantics for SystemC. Compared with the above *UTP* applications, as a system-level modelling language, SystemC not only has real-time and shared-variable features, but also possesses novel features such as delayed notifications, notification cancelling, notification overriding and delta-cycle. Therefore, the *UTP* approach for studying the denotational semantics for SystemC is challenging.

Several efforts have been made to define the formal semantics of SystemC. Müller *et al.* presented a simulation semantics [RHG+01] in the form of Abstract State Machines [BS03]. That semantics covers method, thread, and clocked thread behavior as well as their interactions with the simulation kernel process. Gawanmeh et al. [GHT04] extended the work in [RHG+01] to deal with more complex components of SystemC, including primitive and hierarchical channels, SystemC design rules and a SystemC simulator. A denotational semantics for a synchronous subset of SystemC was proposed by Salem in [Sal03], where the *update* and the *evaluate* phases were formalized using two function domains. Habibi and Tahar presented a semantics of the main part of SystemC in terms of fixpoint [HT05]. The soundness and correctness of the semantics of basis class SC_Module has been proved w.r.t. to a trace semantics of a whole SystemC program. We have also provided an operational semantics for SystemC [PZHJ06]. Based on the operational semantics, bisimulation has been studied for the language by introducing some aspects of reasonable abstractions.

For the study of the linking theory of semantics, Hoare and He have studied the derivation of operational semantics from the algebraic semantics [HH93, HH98]. An operational semantics of CSP [Hoa85] was derived, based on CSP's algebraic laws according to a derivation strategy (called the action transition relation). An operational semantics of Dijkstra's Guarded Command Language (GCL) was also derived based on GCL's algebra according to the derivation strategy (called the step relation). The total correctness of the derived GCL's operational semantics was also discussed in [HJS97]. Recently, Hoare proposed a challenging research topic of the semantic linking between algebra, denotations, transitions and deductions [Hoa11, HvS12]. Various familiar operational calculi have been derived from the algebraic semantics [vSH13]. Compared with the above explorations, this paper studied the linking theory between the denotational semantics and algebraic semantics for SystemC. Our approach is to derive denotational semantics from algebraic semantics. We introduced the concept of guarded choice and provided a full set of parallel expansion laws.

6. Conclusion

Compared with traditional programming languages, SystemC possesses several novel features, including delayed notifications, notification cancelling, notification overriding and delta-cycle. In this paper we studied its denotational semantics via the concept of *Unifying Theories of Programming* [HH98]. The timed model was formalised in a three dimensional structure. A refinement calculus was designed for this three dimensional denotational model. A set of algebraic laws has been studied, especially those which can represent the novel features of SystemC. These laws can be verified via our denotational model.

Meanwhile we also studied the calculation (i.e., derivation) of denotational semantics from algebraic semantics for SystemC. We introduced the concept of guarded components and guarded choice. We systematically explored a full set of parallel expansion laws for SystemC. Our derivation approach is based on the introduction of head normal form. Based on the concept of head normal form, we provided the strategy for deriving denotational semantics from algebraic semantics. Program equivalence can also be explored by using the derived denotational semantics.

For the future, we are continuing to work on the unifying theories [HH98, Zhu05] for SystemC. We plan to embed the achievements of the denotational semantics in the framework of PVS [OSRSC99] to support the mechanical proof of the algebraic laws. The embedding of the derived denotational semantics from algebraic semantics in PVS is also challenging and we aim to support automatic verification based on the *UTP* approach.

Acknowledgement. This work was partly supported by the Danish National Research Foundation and the National Natural Science Foundation of China (Grant No. 61061130541) for the Danish-Chinese Center for Cyber Physical Systems, also supported by National Basic Research Program of China (Grant Nos. 2011CB302904), National High Technology Research and Development Program of China (Grant Nos. 2011AA010101 and 2012AA011205), National Natural Science Foundation of China (Grant No. 61021004), and Shanghai Leading Academic Discipline Project (No. B412).

References

[BB13] Riccardo Bresciani and Andrew Butterfield. A probabilistic theory of designs based on distributions. In *Proc. UTP 2012: 4th International Symposium, on Unifuying Theories of Programming*, Paris, France, 27-28 August, 2012, volume 7681 of *Lecture Notes in Computer Science*, pages 105–123. Springer-Verlag, 2013.

[BS03] Egon Börger and Robert Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

[CWW13] Ana Cavalcanti, Andy J. Wellings, and Jim Woodcock. The safety-critical Java memory model formalised. *Formal Aspects of Computing*, 25(1):37–57, 2013.

[DGJP04] Josee Desharnais, Vineet Gupta, Radha Jagadeesan, and Prakash Panangaden. Metrics for labelled Markov processes. *Theoretic Computer Science*, 318(3):323–354, 2004.

[GHT04] Amjad Gawanmeh, Ali Habibi, and Sofiène Tahar. An executable operational semantics for SystemC using Abstract State Machines. Technical report, Department of Electrical and Computer Engineering, Concordia University, Montreal, Canada, March 2004.

[He94] Jifeng He. *Provably Correct Systems: Modelling of Communication Languages and Design of Optimized Compilers*. The McGraw-Hill International Series in Software Engineering, 1994.

[HH93] C. A. R. Hoare and Jifeng He. From algebra to operational semantics. *Information Processing Letters*, 45:75–80, 1993.

[HH98] C. A. R. Hoare and Jifeng He. *Unifying Theories of Programming*. Prentice Hall International Series in Computer Science, 1998.

[HHH+87] C. A. R. Hoare, I. J. Hayes, Jifeng He, C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sørensen, J. M. Spivey, and B. Sufrin. Laws of programming. *Communications of the ACM*, 38(8):672–686, 1987.

[HJS97] C. A. R. Hoare, He Jifeng, and A. Sampaio. Algebraic derivation of an operational semantics. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, Foundations of Computer Science series. The MIT Press, 1997.

[HLL06] Jifeng He, Xiaoshan Li, and Zhiming Liu. rCOS: A refinement calculus of object systems. *Theoretical Computer Science*, 365(1-2):109–142, 2006.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science, 1985.

[Hoa11] C. A. R. Hoare. Algebra of concurrent programming. In *Meeting 52 of WG 2.3*, 2011.

[HSM97] Jifeng He, Karen Seidel, and Annabelle McIver. Probabilistic models for the guarded command language. *Science of Computer Programming*, 28(2-3):171–192, 1997.

[HT05] Ali Habibi and Sofiène Tahar. SystemC fixpoint semantics. Technical report, Department of Electrical and Computer Engineering, Concordia University, Montreal, Canada, January 2005.

[HvS12] Tony Hoare and Stephan van Staden. In praise of algebra. *Formal Aspects of Computing*, 24(4-6):423–431, 2012.

[IEE01] IEEE. *IEEE Standard Hardware Description Language based on the Verilog Hardware Description Language*, volume IEEE Standard 1364-2001. IEEE, 2001.

[MM04] Annabelle McIver and Carroll Morgan. *Abstraction, Refinement and Proof of Probability Systems*. Monographs in Computer Science. Springer, October 2004.

[OCW09] Marcel Oliveira, Ana Cavalcanti, and Jim Woodcock. A UTP semantics for Circus. *Formal Aspects of Computing*, 21(1-2):3–32, 2009.

[OCW13] Marcel Oliveira, Ana Cavalcanti, and Jim Woodcock. Unifying theories in ProofPower-Z. *Formal Aspects of Computing*, 25(1):133–158, 2013.

[Ope01] Open SystemC Initiative(OSCI). *Functional Specification for SystemC 2.0*, October 2001.

[Ope03] Open SystemC Initiative(OSCI). *SystemC 2.0.1 Language Reference Manual*, 2003.

[OSRSC99] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

[Plo81] Gordon Plotkin. A structural approach to operational semantics. Technical Report 19, University of Aahus, 1981. Also published in *The Journal of Logic and Algebraic Programming*, volumes 60-61:17–139, 2004.

[PZHJ06] Xiaoqing Peng, Huibiao Zhu, Jifeng He, and Naiyong Jin. An operational semantics of an event-driven system-level simulator. In *Proc. SEW-30: The 30th IEEE/NASA Software Engineering Workshop*, Columbia, Maryland, USA, pages 190–200. IEEE Computer Society Press, April 2006.

[RHG+01] Jürgen Ruf, Dirk W. Hoffmann, Joachim Gerlach, Thomas Kropf, Wolfgang Rosenstiel, and Wolfgang Müller. The simulation semantics of SystemC. In *DATE '01: Proceedings of the conference on Design, automation and test in Europe*, pages 64–70, Piscataway, NJ, USA, March 2001. IEEE Press.

[Sal03] Ashraf Salem. Formal semantics of synchronous SystemC. In *Proc. Date'03: Design, Automation and Test in Europe Conference and Exposition*, pages 10376–10381. IEEE Computer Society, March 2003.

[SCHS10] Adnan Sherif, Ana Cavalcanti, Jifeng He, and Augusto Sampaio. A process algebraic framework for specification and validation of real-time systems. *Formal Aspects of Computing*, 22(2):153–191, 2010.

[Sto77] Joe Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language*. MIT Press, 1977.

[vSH13] Stephan van Staden and Tony Hoare. Algebra unifies operational calculi. In *Proc. UTP 2012: 4th International Symposium, on Unifuying Theories of Programming*, Paris, France, 27-28 August, 2012, volume 7681 of *Lecture Notes in Computer Science*, page 88C104. Springer-Verlag, 2013.

[WC01] Jim Woodcock and Ana Cavalcanti. The steam boiler in unified theory of Z and CSP. In *Proc. APSEC 2001: 8th Asia-Pacific Software Engineering Conference*, pages 291–298. IEEE Computer Society Press, December 2001.

[WC02] Jim Woodcock and Ana Cavalcanti. The semantics of Circus. In *Proc. ZB 2002: 2nd International Conference of B and Z Users*, Grenoble, France, January 23–25, 2002, volume 2272 of *Lecture Notes in Computer Science*, pages 184–203. Springer-Verlag, 2002.

[ZHPJ10] Huibiao Zhu, Jifeng He, Xiaoqing Peng, and Naiyong Jin. Denotational approach to event-driven system level language. In *Proc. UTP 2008: 2nd International Symposium on Unifying Theories of Programming*, Dublin, Ireland, 8-10 September, 2008, volume 5713 of *Lecture Notes in Computer Science*, pages 258–278. Springer, 2010.

[Zhu05] Huibiao Zhu. *Linking the Semantics of a Multithreaded Discrete Event Simulation Language*. PhD thesis, London South Bank University, February 2005.

[ZYH10] Huibiao Zhu, Fan Yang, and Jifeng He. Generating denotational semantics from algebraic semantics for event-driven system-level language. In *Proc. UTP 2010: 3rd International Symposium on Unifying Theories of Programming*, Shanghai, China, 15-16 November, 2010, volume 6445 of *Lecture Notes in Computer Science*, pages 286–308. Springer, 2010.