Building SystemC waiting state automata

Nesrine HARRATH
UEI, ENSTA, 32 Bd Victor,
75739 Paris cedex 15, France
www.ensta-paristech.fr
nesrine.harrath@ensta-paristech.fr

Bruno Monsuez
UEI, ENSTA, 32 Bd Victor,
75739 Paris cedex 15, France
www.ensta-paristech.fr
bruno.monsuez@ensta-paristech.fr

Joëlle Delacroix
CEDRIC, CNAM, 292 rue Saint Martin,
75141 Paris cedex 03, France
www.cnam.fr
delacroix@cnam.fr

SystemC is becoming a de facto standard for the system-level description of system-on-chip. However most formal verification techniques used for verifying hardware components targets low level design, usually netlist or RTL, but time-to-market requirements have rushed the industry towards design paradigms that offer a very high level of abstraction. In previous works, we proposed a verification methodology based on SystemC waiting-state automata, an abstract formal model for verifying properties of SystemC at the transaction level within a delta-cycle. The main drawback of this model is that it should be provided manually. In this paper, we propose a method to automatically build the SystemC waiting-state automata from the SystemC code. It is based on an extended symbolic execution of the SystemC design that takes care of synchronous as well as asynchronous communications and that preserves the semantics of SystemC up to a delta-cycle.

SystemC, automata, operational semantics, symbolic execution, compostional verification

1. INTRODUCTION

Nowadays, with the ever increasing complexity of embedded systems, the need of a language that can describe both hardware and software parts did emerge. Besides, embedded electronic devices are often used for highly safety critical applications. Designing those systems is both very error-prone and time-consuming. SystemC language has been developed to target these issues. Using SystemC, developers can easily create a working model of the system at a functional level where more details can then be added in order to refine this model. Despite the efficient implementation of the simulator, it is not feasible to find all corner cases necessary to catch all errors, particularly in critical parts of the design. Therefore, the use of formal methods allow to proove the correctness of the design of the software development.

Over the last years, research activities were mainly focused on exploiting modeling flexibility and exploring different levels of communication and behavior abstraction. More recent work concentrates on formalization and verification. The aim of our work is to propose an executable semantics of the SystemC language in order to automatically build the waiting state automata: an abstract model previously presented by Zhan et al. (2007); Harrath and Monsuez (2009) and that conforms to the SystemC scheduler up to the delta-cycles. Although, the model presents many advantages like its refinability and modularity, there is a necessity to automatically build the SystemC waiting-state automata from the control flow graph as mentioned by Herber et al. (2010).

In this paper, we present an effective methodology to automatically generate the automata from the SystemC code. We use the SystemC waiting-state automata first to model the SystemC components and then to proceed to formal verification. However, up to this stage we can not garantee that we detect all software conception errors. Therefore, we need further methods to refine our verification methodology: we may resort to techniques like equivalence checking as introduced by Somenzi and Kuehlmann (2006) to prove the equivalence of two system descriptions or abstraction techniques, often based on abstract interpretation, to provide a method for symbolically executing systems using the abstract instead of the concrete domain. In this present work, we adopt a new technique that extends the symbolic execution in order to refine and to automatically build the waiting-state automata.

We define first a structural operational semantics of the SystemC language and then we statically execute the SystemC code for each process using symbolic execution. Symbolic execution uses symbolic values of inputs instead of real values

and generates all the traces between the waiting states: the synchronizing points between the communicating threads of the program. To reduce the complexity of the generated automata, we have to approximate the symbolic relations inferred by the symbolic execution. We perform those approximations by explicit *abstraction* operations, which introduces arbitrary, logical relations between predicates defined over the variables of the program. The inferred relations are constructed from those predicates. Thus, the problem of building the SystemC waiting-state automata can be transformed to the simpler problem of inferring relations between predicates and their associated functions, which can be done heuristically or, when necessary, manualy by the user.

The main contributions of this work are: (i) To define new semantics of a subset of SystemC that extends the semantics of Shyamasundar et al. (2007) and that expresses more details about the system behaviour. (ii) To symbolically execute the SystemC program in order to automatically build the waiting-state automata.

The paper is organized as follows. In Section 2, we review and discuss related works. In Section 3 we briefly present a subset of SystemC as well as the SystemC scheduler. Section 4 introduces the SystemC waiting-state automata. Section 5 is devoted to the low-level structural operational semantics of SystemC. Section 6 presents the main steps to automatically build the SystemC WSA from SystemC code using the symbolic execution. In this section, we briefly discuss the use of predicate abstraction to reduce the complexity of the inferred symbolic relations. We finally conclude in Section 7.

2. RELATED WORK

In this section we present related work on formalizing semantics of SystemC, then we compare them to our approach. Salem (2003) presented formal semantics of a synchronous subset of SystemC using denotational semantics. The delta cycle was formulated using function domains. Physical time was modeled on the clock period level. A description style based on defining two types of processes: Synchronous and Combinational was also proposed. The semantics of the SystemC methods and threads limited to this description style were defined. The evaluate and update phases of SystemC scheduler have been formulated for both timed and immediate notifications. This work provides the description of the above parts only using general syntactic rules. It does not provide any specific definitions for basic SystemC components and processes; like wait or notify. The provided semantics, as all denotational semantics, were compact, precise, and providing a rigorous way to understand the SystemC as programming or modeling language. However, it does not stress on the behavior of the language, the implementation and interaction between different components, and how synchronization is maintained during simulation time, which was well captured in our semantics.

Later several works formalize SystemC semantics using automata. For instance, Gawanmeh (2004) and Mueller et al. (2003) translate SystemC program simulation using abstract state machines (ASM). ASM have been extensively used in the definition of different modeling and hardware description languages, but it is still not efficient since it does not capture the synchronization between processes at the waiting states. Besides, their model is not adapted for new techniques for programs analysis like model checking Clark et al. (1999) or abstract interpretation Cousot and Cousot (1977). Besides, works of Nieman and Haubelt (2007); Karlsson et al. (2006) propose a formal representation of SystemC models at a high level of abstraction using respectively communicating automata and petri-nets, but none of them is adequate to verify properties like concurrency and interactions between processes at the delta-cycle level and properties like simulation anomalies and time. The work in Man (2004) uses a process algebra but ignores the need of distinctions between synchronous and asynchronous simulation, ignores delta cycles etc.

Recently, there has been two interesting approaches based on the notation of Structural Operational Semantics (SOS) of Plotkin (2004) that capture at the same time formal semantics of SystemC structures and synchronous and asynchronous interactions between concurrent threads at the delta cycle: works of Shyamasundar et al. (2007) and Xiaoqing et al. (2006). Our operational semantics are based on these two works.

3. SUBSET LANGUAGE

3.1. Introduction

SystemC is a System-level modeling language based on C++ that is intended to enable system level design in response to the need of a very fast executable specification to validate and verify system concepts.
Using the SystemC library, a system can be specified at various levels of abstraction. For hardware implementation, models can be written either in a functional style or in a register-transfer level style. The software part of a system can be naturally described in C or C++.

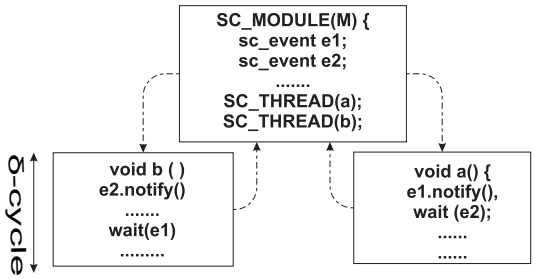


Figure 1: *An example of SystemC program*

The base layer of SystemC semantics as presented by Mueller et al. (2001) provides an event-driven simulation kernel. This kernel operates at the event level and switches execution between processes.

3.2. Syntax

For simplicity, we omit the syntactic elements for representing the architecture of a SystemC program. Syntactically, a SystemC program consists of a set of modules, a module contains one or more processes to describe the parallel aspect of the design. Processes inside a module are of three types: *Method*, *Thread* and *Clocked Thread*. However, methods and clocked threads can be modelled as threads without loss of generality. Therefore, only *threads* are considered throughout the rest of this paper.

Similar to VHDL or Verilog, a process has a list of events that activate it. This list of events is called the *sensitivity list* of the process. As soon as the event occurs, the process is activated and executes until the process terminates or suspends its execution by means of the **wait()** statement. The SystemC methods are special cases of processes that do not call **wait()**. Events may either be generated explicitly by a thread (using the **notify()** statement or method), or implicitly by changing signal values.

We illustrate these concepts by an example in Figure 1: The example shows a module *m* with two input ports and two threads *a* and *b*. Both threads communicate using events e_1 and e_2: Thread *a* notifies the event e_1 that activates the thread *b* and it is waiting for the event e_2 to be notified by *b*. SystemC specification distinguishes three states of a thread: *running*, *waiting* and *runnable*. A running thread may generate events that activate other threads sensitive to the event and change their states to runnable. A single event may trigger the execution of multiple threads. A running thread may become a waiting thread by executing the wait statement. The scheduler chooses a thread among the runnable threads to resume execution. As in Verilog, the ordering in which the runnable threads are activated is chosen non-deterministically. It is important to note that no interleavings are done between the threads unless a wait() statement is executed. This is a major difference between SystemC and other system-level modeling languages such as SpecC, which allows arbitrary interleavings between the threads. It is important to note that the synchronization does not happen upon the generation of the event, but only upon calling wait.

3.3. SystemC scheduler

The simulation of SystemC models is managed by the SystemC scheduler, which can be seen as a total event-driven model: communications through ports and channels, clocks, and actions of modules, are all triggered by (different types of) events. The basic unit of the simulation is the so-called *delta-cycle* and a complete simulation procedure is just a sequence of delta-cycles.

Like VHDL and Verilog, the SystemC kernel supports the notion of a *delta-cycle*. It is a micro time and one simulation time unit that includes many delta-cycles. The delta-cycle semantics guarantee that the combinational logic behavior can be simulated even if there are combinational feedback loops within the circuit. The simulation clock does not advance during a delta-cycle, and as a result all processes that execute during the delta-cycle appear to be executing simultaneously. In order to maintain the appearance of parallel execution, it is also necessary to postpone the effect of all channel and signal updates and event notifications. To that end, during a delta-cycle the kernel switches from *evaluation phase* to *update phase* to *delta-notification phase*. Below, we define steps for SystemC simulation:

1. *Initialization*.
2. *Evaluation Phase*. Select a ready process to execute. The order of the selection is nondeterministic. The execution of a process may cause immediate event notifications to occur, possibly resulting in additional processes becoming ready to run in the same evaluation phase. The execution of a process may generate pending requests to update channels in the following update phase. Repeat this step for any other processes that are ready to run.
3. *Update Phase*. Carry out all pending channel update requests generated in last evaluation phase, which may generate some events. Then go to **step 2**.
4. *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 2**.

5. Simulation 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 simulation 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 there exist processes ready to run, then go to **step 2**, otherwise advance the current simulation time by one time unit again.

6. If there are no more timed notifications, simulation is finished.

4. SYSTEMC WAITING STATE AUTOMATA

In this section, we introduce an overview about an abstract formal model: the SystemC waiting-state automaton (WSA). It is an abstract model that describes SystemC components at the delta-cycle level. It was first introduced by Zhan et al. (2007) and then extended in a second work by Harrath and Monsuez (2009).

4.1. Introduction

Several attempts have been made to apply formal verification methods to SystemC at different levels of abstraction. For example the work in Drechsler and Große (2002), presents the formal semantics of SystemC at the gate level, later many works stress on the correctness of SystemC at the transactional level like Nieman and Haubelt (2007); Karlsson et al. (2006). But none of them studies the correctness of SystemC modules at the delta-cycle level, which is the main contribution of the waiting-state automata. the WSA is a formal model for verifying SystemC components that is based on the waiting states of processes within a delta cycle and that conforms to the SystemC scheduler up to delta cycle. The basic idea of Waiting-State Automata is to consider only the states where processes are hung up, i.e. waiting for some events. In fact, a delta-cycle always starts from a state where all the processes are hung up and the whole cycle can be seen as a sequence of transitions between these waiting states.
Our goal using waiting state automata is to represent a concrete SystemC design as an abstract model that can be used to verify some design properties, where we preserve the behavior of the original model with regard to the waiting states of its processes and the set of the events that activate them. What makes the waiting-state automata a different model compared to existing approaches is:

- It is a compositional model, used to capture the behavioral semantics of each thread independently and is abstracted by an automaton. Then all the automata are composed together to build a global model that describes the whole system bahaviour. Zhan et al. (2007) and Harrath and Monsuez (2009) define two algorithms for composing and reducing minimal automata.

- At the delta cycle level we may verify properties regarding the compositional behaviour of SystemC designs, i.e interactions and interleavings between concurrent processes and verify properties like *causality* and *determinism*.

In Figure 2, we briefly present the difference between the WSA and the *control flow graph* (CFG). In the CFG, the nodes represent the basic commands and guard expressions of the program and the edges stand for flow of control between the nodes. However, nodes in the WSA represent only the *wait* statements: the synchronizing points between processes. Besides, transitions define elements that activate and suspend the process and consequences of that.

4.2. Formalism

Formally, a SystemC waiting-state automaton over a set V of variables is a triple $A = (S, E, \mathcal{T})$, where S is a finite set of states, E is a finite set of events, T is a finite set of transitions where every transition is a 6-tuples $(s, e_{in}, p, e_{out}, f, s')$:

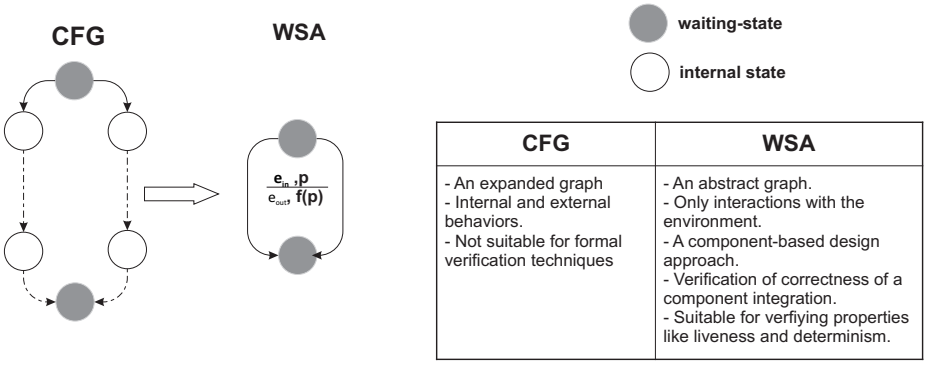
- s and s' are two states in S, representing respectively the initial state and the final state;

- e_{in} and e_{out} are two sets of events : $e_{in} \subseteq E; e_{out} \subseteq E$;

- p is a predicate defined over variables in V, i.e., $FV(p) \subseteq \mathcal{V}$, where ***FV(p)*** denotes the set of free variables in the predicate *p*;

- f is an effect function over $\mathcal{V}$;

We often write $s \xrightarrow[e_{out},f]{e_{in},p} s'$ for the transition $(s, e_{in}, p, e_{out}, f, s')$. Intuitively, e_{in} and the predicate p act as a *guard condition*: the transition is triggered if and only if all the events in e_{in} are present and the predicate p holds; e_{out} and the function f represent an effect: the transition will generate all the events in e_{out} and f will be applied to the current instantiation of V.



CFG	WSA
- An expanded graph - Internal and external behaviors. - Not suitable for formal verification techniques	- An abstract graph. - Only interactions with the environment. - A component-based design approach. - Verification of correctness of a component integration. - Suitable for verfiying properties like liveness and determinism.

Figure 2: *CFG to WSA*

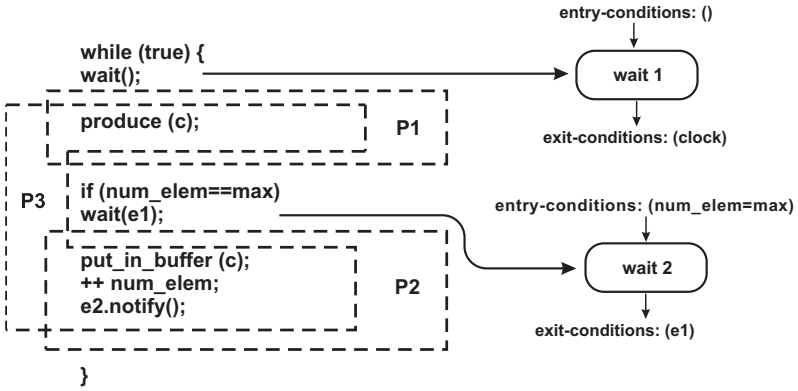


Figure 3: *The generation of the SystemC waiting-state automaton for the producer module*

4.3. Example

To illustrate the formalism of the SystemC waiting-state automata, we take the example of the producer module in the FIFO. The producer is continuously putting data into a buffer at each clock edge and continues producing until the buffer is full, then it waits for an event e_1 which is triggered when the buffer is emptied to resume execution. As shown in Figure 3, the two *wait* statements define the two waiting-states of the automaton, they devide the automaton into three parts (P_1, P_2, P_3) according to the execution trace. Each piece of P_1, P_2 and P_3 executes in an instant, and are seen actually as transitions between the waiting-states.

The objective is then to represent formally how a process controls the transitions between the waiting-states. As shown in Figure 3, we calculate, for every waiting-state, the entry-condition and the exit-condition. For instance, the condition for entering the *wait 2* state is $(num\_elem = max)$ and the condition for exiting the *wait 1* state is a *clock* signal, so the guard condition for the transition from *wait 1* to *wait 2* is an event *clock* plus a predicate $(num\_elem = max)$, defined on the variable num_elem. As a transition is actually a piece of program, it can also generate new events and modify the values of variables. For instance, both P_2 and P_3 generate an event *e2* and increments the variable num_elem by 1. We write *inc* to denote that the variable has increased by 1 and *id* if the variable remains unchanged.

The SystemC waiting-state automaton for the producer process is shown in Figure 4.

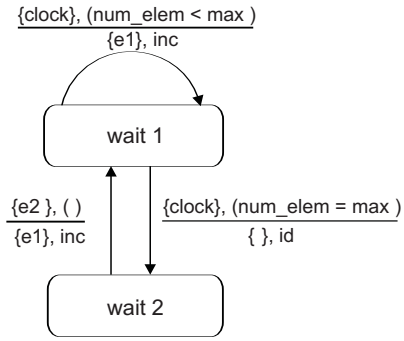


Figure 4: *The SystemC waiting state automata for the producer*

5. SEMANTIC FRAMEWORK

In order to analyse statically SystemC designs and extract the required information to generate the WSA representation, we define a low-level semantics of SystemC using the notation of the

Structural Operational Semantics (SOS) defined by Plotkin (2004). We use a small-step semantics to specify the operation of a program one step at a time. Our semantics are similar to the semantics of Shyamasundar et al. (2007) where a complete behavioral semantics of SystemC is proposed. Here, we stress specifically on three main points:

- capture all reactive features of SystemC.
- specify a network of synchronous and asynchronous components computing either high-level transactions or low-level event communications.
- specify two time scales: the delta cycle and the simulation time.

5.1. Sequential constructs

We briefly present semantics for sequential processes including assignments, channel statements, event statements, guarded statements and wait statements, then we present semantics for parallel composition of processes.

The main difference between our semantics and the semantics of Shyamasundar et al. is that, in the parallel composition, we distinguish the four phases for the SystemC simulation semantics: the *evaluation phase*, the *update phase*, the *delta cycle advancing phase* and finally the *timed simulation advancing phase*.

To describe how a statement changes the configurations of the environment components, we write the transitions rules for sequential processes as follow: $\langle stmt, \sigma\rangle \xrightarrow[E]{E_o} \langle stmt', \sigma'\rangle$ where:

- *stmt* and *stmt'* are the statement before and after the transition respectively. The notation ϵ stands for a terminated process.
- σ and σ' represent respectively the current and the next state of the program. Each state is a function over local and global variables and channels.
- E is the environment while the transition is executed, E_o is the output environnement emitted during the transition. An environment is a 5-tuple $E = (E^I, E^\delta, E^T, \mathcal{V}, \mathcal{RQ})$: E^I is the set of immediate events,E^δ is the set of delta events, E^T is the set of timed events, $\mathcal{V}$ is the set of next delta updates for variables and $\mathcal{RQ}$ represents the set of request updates for channels. A request is a pair $(ch, exp(\sigma))$, where: *ch* is the channel name and *exp* denotes the channel value.

5.1.1. Assignment

The execution of an assignment $v := exp$ is instantaneous. It assigns the value of the expression *exp* to the variable *v* and keeps unchangeable the other variables and channels.

$$\frac{v \in \mathcal{V}}{\langle v := exp, \sigma\rangle \xrightarrow[E]{\cdot} \langle \epsilon, \sigma[v/exp]\rangle}$$

Due to space limitation, we omit transition rules for the statements: skip, stop, var v as well as conditions and iterations. These rules ar similar to the rules presented in Zhu (2005).

5.1.2. Channel statements

The execution of channel statements involves two cases:

- Executing an output statement that we note $ch!!exp$ generates a request to update the channel *ch* with the expression *exp* and leaves other variables and channels unchanged. All pending requests are carried out in the following update phase. The newly generated request will remove the existing one from the request queue. We use operator $\setminus$ to represent removing all elements from the request queue. (1) shows the transition rule of the channel output statement.
- Execution an input statement that we note $ch??v$ assigns the current value of channel *ch* to the variable *v* and leaves other variables and channels unchanged. (2) shows the transition rule of the channel input statement.

$$(1)\ \frac{ch \in Channels \wedge \sigma(ch) \neq exp(\sigma)}{\langle ch!!exp, \sigma\rangle \xrightarrow[E^I, E^\delta, E^T, \mathcal{V}, \mathcal{RQ}]{E^I, E^\delta, E^T, \mathcal{V}, \mathcal{RQ}\setminus(ch, exp(\sigma))} \langle \epsilon, \sigma[v/exp]\rangle}$$

$$(2)\ \frac{ch \in Channels, v \in \mathcal{V}}{\langle ch??exp, \sigma\rangle \xrightarrow[E]{\cdot} \langle \epsilon, \sigma[v/ch]\rangle}$$

5.1.3. Event statements

The event notification statement immediately emits an event *e* in the next environment, and terminates. The processes waiting on these events will unblock in either the synchronization with the next environment or the synchronization with the next delta environment respectively. According to the way an event is notified, there are three kinds of event notifications: immediate notifications $notify()$, delta notifications $notify\_\delta()$ and timed notifications $notify\_t()$.
The execution of *notify()* triggers event *e* immediately, which will activate all processes that are waiting for it. The immediate event notification also overrides the delayed notifications on the same event

if it will be notified in later delta cycles. The execution of $notify\_\delta()$ is instantaneous, which results in some changes in E^{δ} and E^{T} , not only adding a delayed notification to some sets, but also overriding a delayed notification. A notification scheduled to occur earlier will always override the one scheduled to occur later. Transition rules for events notifications are as below:

$$\langle e.notify(), \sigma\rangle \xrightarrow[E]{e,e,\emptyset,\emptyset,\emptyset} \langle \epsilon, \sigma\rangle$$

$$\langle e.notify\_\delta(), \sigma\rangle \xrightarrow[E]{\emptyset,e,e,\emptyset,\emptyset} \langle \epsilon, \sigma\rangle$$

$$\langle e.notify\_\bot(), \sigma\rangle \xrightarrow[E]{\emptyset,\emptyset,e,\emptyset,\emptyset} \langle \epsilon, \sigma\rangle$$

5.1.4. wait statements
The behavior of the *wait* statement is to wait for an event *e* to be in the environment or for a timeout. It works as a synchronization between parallel processes. Syntactically, rules for *wait* statement must be defined as follows:

- Rule 1 defines that if an event *e* is not in the environment, the process continues to wait without doing anything.
- Rule 2 defines that if an event *e* is present in the environment, the *wait* statement terminates and reduces to nothing.

$$\frac{e \notin E}{\langle wait(e), \sigma\rangle \xrightarrow[E]{\cdot} \langle wait(e), \sigma\rangle}$$

$$\frac{e \in E)}{\langle wait(e), \sigma\rangle \xrightarrow[E]{\cdot} \langle \epsilon, \sigma\rangle}$$

5.2. Sequential composition

If we consider two sequential processes P_1 and P_2, there are two cases for sequential composition. If process P_1 does not terminate in the current instant, then P_2 cannot start. If P_1 terminates then P_2 starts in the environment in which P_1 terminates. Transitions rules are defined as follows:

$$\frac{\langle P_1, \sigma\rangle \xrightarrow[E]{E_1,E_1^{\delta},E_1^{T},\mathcal{V}_1,\mathcal{RQ}_1} \langle P_1', \sigma'\rangle}{\langle P_1; P_2, \sigma\rangle \xrightarrow[E]{E_1,E_1^{\delta},E_1^{T},\mathcal{V}_1,\mathcal{RQ}_1} \langle P_1'; P_2, \sigma'\rangle}$$

$$\frac{\langle P_1, \sigma\rangle \xrightarrow[E]{E_1,E_1^{\delta},E_1^{T},\mathcal{V}_1,\mathcal{RQ}_1} \langle \epsilon, \sigma'\rangle}{\langle P_1; P_2, \sigma\rangle \xrightarrow[E]{E_1,E_1^{\delta},E_1^{T},\mathcal{V}_1,\mathcal{RQ}_1} \langle P_2, \sigma'\rangle}$$

5.3. Parallel composition

In this section, we consider transition rules for parallel composition (Table 1). There are two kinds of configurations for parallel processes, one representing executing processes (processes that have been selected by the scheduler) and the other one representing processes that are not executing (either waiting or runnable). In this section, we distinguish between the three phases in the simulation process of the SystemC scheduler.

5.3.1. The Evaluation Phase
The evaluation phase starts from a non-empty table of runnable processes (i:e, processes that are waiting to be selected by the scheduler). Here we have two scenarios:

1. Immediate notifications of a set of events with processes waiting for them.
2. No immediate notifications but there is a non-empty set of runnable processes.

Transition rules for the evaluation phase are presented in Table 1. Rule (1) is for immediate composition, it is defined to unblock all processes that are waiting for events present in the environment. We use the expression *waiting(P,e)* to denote that the process *P* is waiting for the event *e*. In other words, we may write the process *P* in a sequential form *wait;P'*. It is a synchronous composition, but only for the wait statements.
Rule (2) is used when all current processes are hung up *waiting* for events. The scheduler selects a process from the set of *ready* processes. This process runs until it reaches the next *wait* state. The function *add* guarantees that next-delta events and next-delta modifications of variables is taken into account in the process of scheduling the concurrent processes so that non deterministic bahaviour can be detected, i.e, non-deterministic behaviour is possible when two or more different values can be written to a signal at the same evaluation phase, it must guarantees that the assigned value to the signal is exactly the last value taken by the signal.

5.3.2. Update phase
If there is no runnable processes, the scheduler updates channels (*ch*) with new data values (*v*). Rules for the update phase are defined in Table 1.

5.3.3. Delta cycle advancing phase
The rule proceeds only when there is no immediate events and there exists some delta events. The transition makes the delta events in E^{δ} become the immediate events in the next instant, and updates the state of variables. Here, we resume rules for the evaluation and the update phases since we are dealing with a new delta cycle.

Table 1: *Semantics of the parallel composition*

(Evaluation phase)
(1)

$$\frac{\forall i\in\{1..n\},\exists e\in E^I, waiting(P_i,e)\ \wedge\forall j\in\{n+1..m\},\forall e\in E^I,\neg waiting(P_j,e)}{\langle P_1\|\cdots\|P_n\|\cdots\|P_m,\sigma\rangle \xrightarrow[E]{(\emptyset,E^\delta,E^T,V^\delta,\mathcal{RQ})} \langle P_1'\|...P_n'\|\cdots\|P_m,\sigma'\rangle}$$

(2)

$$\frac{\begin{array}{c}\forall i\in\{1\ldots n\},\ waiting(P_i)\ \ \forall j\in\{n+1\ldots m\}, ready(P_j)\\ select\ p\in\{n+1\ldots m\}, \langle P_p,\sigma\rangle \xrightarrow[E]{(E_p^I,E_p^\delta,E_p^T,V_p^\delta,\mathcal{RQ}_p)} \langle P_p',\sigma'\rangle\\ add(\langle E_p^\delta,E^\delta\rangle,\langle E_p^T,E^T\rangle,\langle V_p^\delta,V^\delta\rangle)\end{array}}{\langle P_1\|\cdots\|P_n\|\cdots\|P_p\|\cdots\|P_m,\sigma\rangle \xrightarrow[(E^I,E^\delta,E^T,V^\delta,\mathcal{RQ})]{(E_p^I,E_p^\delta\cup E^\delta,E_p^T\cup E^T,V_p^\delta\cup V^\delta,\mathcal{RQ}_p\cup\mathcal{RQ})} \langle P_1'\|\cdots\|P_n'\|\cdots\|P_p'\|\cdots\|P_m,\sigma'\rangle}$$

(Update phase)

$$\langle P_1\parallel\cdots\parallel P_n,\sigma\rangle \xrightarrow[(E^I,E^\delta,E^T,V^\delta,\mathcal{RQ})]{(E^I,E^\delta,E^T,V^\delta,\emptyset)} \langle P_1\parallel\cdots\parallel P_n,\sigma[v/ch]\rangle$$

(Delta advancing phase)

$$\frac{\forall i\in\{1..n\}, waiting(P_i)}{\langle P_1\|\cdots\|P_n,\sigma\rangle \xrightarrow[(\emptyset,E^\delta,E^T,V^\delta,\mathcal{RQ})]{(E^\delta,\emptyset,\emptyset,\emptyset,\mathcal{RQ})} \langle P_1\|\cdots\|P_n,\sigma[V^\delta/V]\rangle}$$

(Timed advancing phase)

$$\frac{\forall i\in\{1..n\}, waiting(P_i)}{\langle P_1\|\cdots\|P_n,\sigma\rangle \xrightarrow[(\emptyset,\emptyset,\emptyset,\mathcal{RQ})]{(E^T,\emptyset,\emptyset,\emptyset,\mathcal{RQ})} \langle P_1\|\cdots\|P_n,\sigma[V^\delta/V]\rangle}$$

5.3.4. Simulation timed advancing phase
Here, we define rules for the synchronization on timed events which builds the next environment from time events and advance macro-time (time simulation). It is effective when all processes are blocked, where there are no immediate events nor delta events. Timed events are posted by wait(time) statements, timers and clocks. We define here the same rules for the evaluation and the update phases.

6. BUILDING THE SYSTEMC WAITING STATE AUTOMATA USING SYPMBOLIC EXECUTION

Our approach is embedded in the overall process of building WSA. We suppose that the waiting-state automata conform to SystemC semantics at the delta cycle level (Section 3). The basic idea is to combine two static methods for programs analysis: first writing the operational semantics for the basic statements and then proceeding to the symbolic execution of the program. The operational semantics were introduced in Section 5, later we define the symbolic execution and how to use it to build the waiting state automata.

6.1. Background

The *symbolic execution* (SE) as first introduced by King (1976); Darringer (1988) is a natural extension of normal execution providing normal computation as a special case. The main idea behind SE is the use of symbolic values instead of the real ones in order to generate the set of all possible executions for all the values of the input variables. The semantics and rules of the symbolically executed program remain the same and need just to be extended in order to deal with the symbolic values. Therefore the assignment operation in fairly obvious, the assigned variable changes its interpretation by evaluating the expression to the right that consists in replacing all the symbolic values that it contains with their corresponding symbolic expressions. As for the conditional instructions a choice has to be made in order to decide what branch should be taken. Therefore a path condition *(PC)* is also included in the state that will keep track of all the decisions made along the execution, working as an accumulation of assertions made on that symbolic variables, refining their values domains and helping decide which of the *then* or the *else* branches should be taken. We can see that, by construction, the SE only generates feasible paths.

6.2. Extension of the Symbolic execution

In the following we will describe the main idea behind SE throughout the example of the producer module. Our technique consists in symbolically executing the SystemC program in order to generate all the reachable states and consequently the set of the executing traces.

The state of a basic SE is composed of the values of the current variables in use and the path condition (*PC*) that represents the history of the choices made up to that point, mostly present to deal with the conditional instructions. Besides, the state is composed of the *input* events, the *output* events and a function *f* that modifies the variables. Formally, we write: $S = (Var, e_{in}, \mathcal{PC}, e_{out})$ where:

- Var is the set of the global variables.
- e_{in} is the set of events that activates the state.
- $\mathcal{PC}$ is the path condition.
- e_{out} is the set of events triggered.

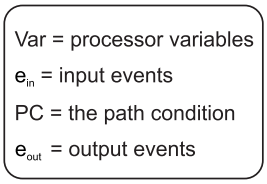


Figure 5: *The symbolic state*

6.3. New semantics of the SystemC Components

In this section, we modify rules in Section 5 to suit the formalism of the waiting-state automata (Section 4), then we proceed to the symbolic execution of the producer example and finally we generate the set of executing traces of the process.

In Table 2, we represent new rules for the *wait* and the *notify* statements, along with the three basic programming constructs in imperative languages: *assignments*, *conditional statements*, and *loops*. In fact each transition has: (i) a set of pre-conditions: an input event e_{in} that triggers the transition and a predicate p over variables and (ii) a set of post-conditions: an output event e_{out} and a function f over predicates.

Table 2: *The modified operational semantics of SystemC statements*

assignment

$$\{x := e, \sigma\} \xrightarrow[\{\},x=e]{\{\},()} \{\sigma[x/e]\}$$

if

$$\frac{\{b,\sigma\}\rightarrow\{true,\sigma\} \quad \{p,\sigma\}\rightarrow\{\sigma'\}}{\{if\ b\ then\ p\ else\ q,\sigma\} \xrightarrow[\{\},.]{\{\},(b=true)} \{\sigma'\}}$$

$$\frac{\{b,\sigma\}\rightarrow\{false,\sigma\} \quad \{q,\sigma\}\rightarrow\{\sigma'\}}{\{if\ b\ then\ p\ else\ q,\sigma\} \xrightarrow[\{\},.]{\{\},(b=false)} \{\sigma'\}}$$

while

$$\frac{\{b,\sigma\}\rightarrow\{true,\sigma\} \quad \{p;\ while\ (b)\ do\ p,\sigma\}\rightarrow\{\sigma'\}}{\{while\ (b)\ do\ p,\sigma\} \xrightarrow[\{\},.]{\{\},(b=true)} \{\sigma'\}}$$

$$\frac{\{b,\sigma\}\rightarrow\{false,\sigma\}}{\{while\ (b)\ do\ p,\sigma\} \xrightarrow[\{\},.]{\{\},(b=false)} \{\sigma\}}$$

wait(e)

$$\frac{e\in E}{\{wait(e),\sigma\} \xrightarrow[\{\},.]{\{e\},()} \{\sigma\}}$$

$$\frac{e\notin E}{\{wait(e),\sigma\} \xrightarrow[\{\},.]{\{\},()} \{wait(e),\sigma\}}$$

e.notify()

$$\frac{add(e,E)}{\{e.notify(),\sigma\} \xrightarrow[\{e\},.]{\{\},()} \{\sigma\}}$$

6.4. Symbolic execution of the producer example

We resume the example of the producer. The symbolic execution of the program as shown in Figure 6, begins with the first statement: *while(true)*, we have to deal with two cases: the loop is not enterred and the loop is unfolded at least once. Technically, we apply the *while* rule (Table 2), in this case we don't have two branches since the guard is usually true.

The next step in the symbolic execution is to deal with the *wait()* statement, we use the transition rule for the wait defined in Table 2. We precise the entry and the exit conditions for this transition, in the top of the transtion the input event $e_{in} = clock$, there is no predicate. As a result, no event is triggered and there is no modification over predicates. This represents the first waiting state in the SystemC waiting-state automaton.

Once we precise rules for the wait statement, the next symbolic execution step is to apply the *if* rule to the statement $if(num\_elem = max)$, which produces two conjuncts in place of one. The first conjunct yields to the next wait statement $wait(e_1)$ which represents the second state in the SystemC waiting state automaton: this branch constitutes the trace from the first waiting state to the second waiting state. The transition rule for the second wait statement is the same as in Table 2 with only one entry condition $(e_{in} = e_1)$ and no exit conditions.
The second conjunct yields directly to the *assignement* num_elem++, we apply the assignment rule. The effect of the assignment manifests itself in the assumption $(num\_elem = num\_elem + 1)$: it represents the affect function that we note *inc* in the SystemC waiting state automaton. We write *id* for the identity function.

The last statement of the producer process is $notify(e_2)$, the symbolic execution of this statement just triggers an output event e_2. Since we have a loop, we turn back to the first wait statement and then we build another trace from the second waiting state to the first waiting state.

Table 3: *Operational semantics for the producer process*

$$\langle wait(), \sigma\rangle \xrightarrow[\{\},id]{\{clock\},()} \langle \epsilon, \sigma\rangle$$

$$\langle if(num\_elem = max), \sigma\rangle \rightarrow \begin{cases} \frac{\{\},(num\_elem=max)}{\{\},id}; \frac{\{e_1\},()}{\{\},id} \\ \frac{\{\},(num\_elem<max)}{\{\},id} \end{cases}$$

$$\langle if, \sigma\rangle \xrightarrow[\{\},id]{\{\},(num\_elem=max)} \langle true, \sigma\rangle; \langle wait(), \sigma\rangle \xrightarrow[\{\},id]{\{e_1\},()}$$

$$\langle if, \sigma\rangle \xrightarrow[\{\},id]{\{\},(num\_elem<max)} \langle false, \sigma\rangle, \langle if, \sigma\rangle \rightarrow \langle \epsilon, \sigma\rangle$$

$$\langle num\_elem++, \sigma\rangle \xrightarrow[\{\},inc]{\{\},()} \langle \epsilon, \sigma\rangle$$

$$\langle e_2.notify(), \sigma\rangle \xrightarrow[\{e_2\},id]{\{\},()} \langle \epsilon, \sigma\rangle$$

$$\langle wait(), \sigma\rangle \rightarrow$$

As it is shown in Figure 6, the set of traces between the waiting states of the producer process are three: one trace from the first wait statement to the second and another trace in reverse and only one loop. They are described as below:

- $\xrightarrow[.,id]{\{clock\},.}; \xrightarrow[\{\},id]{\{\},(num\_elem=max)}$: it represents the transition from the state *wait 1* to the state *wait 2* (see Figure 7).
- $\xrightarrow[.,id]{\{e_1\},.}; \xrightarrow[\{\},inc]{\{\},()}; \xrightarrow[\{e_2\},id]{\{\},()}$: it represents the transition from the state *wait 2* to the state *wait 1* (Figure 7).
- $\xrightarrow[.,id]{\{clock\},.}; \xrightarrow[\{\},id]{\{\},(num\_elem<max)}; \xrightarrow[\{\},inc]{\{\},()}; \xrightarrow[\{e_2\},id]{\{\},()}$: it represents the loop in the automaton in Figure 7.

6.5. Building the SystemC waiting state automata using symbolic execution

In this section, we briefly show how to generate the waiting-state automaton for a SystemC program. In fact, a SystemC program is composed of a set of processes (or threads), each process is either operating locally or communicationg with other processes using the wait statements. First, we build the waiting-state automaton for each process independently, where we apply the semantic rules, that we define in Section 5, together with symbolic execution. We use this description to build a transition system for each process where we consider only the wait statements. For instance, if we consider below the execution trace that we build using our rules:

$$\langle stmt_0, \sigma_0\rangle \xrightarrow[e^1_{out},f^1]{e^1_{in},p^1} \ldots \xrightarrow[e^n_{out},f^n]{e^n_{in},p^n} \langle stmt_n, \sigma_n\rangle$$

During the execution of the process from $stmt_0$ to $stmt_n$ (or between σ_0 and σ_n), the states are observable only from within the process, and no other process in the environment can observe the intermediate states. Hence, from the environment point of view, only the first and the last states are observable. We use the predicate abstraction as introduced first by Graf and Saidi (1997) to infer the relations between the transitions guards and affects. Formally, we apply the following abstraction rule:

$$\frac{\{stmt_1, \sigma_1\} \xrightarrow[e^*_{out},f^*]{e^*_{in},p^*} \{stmt_n, \sigma_n\}}{\{stmt_1, \sigma_1\} \xrightarrow[e^1_{out},f^1]{e^1_{in},p^1} \ldots \xrightarrow[e^n_{out},f^n]{e^n_{in},p^n} \{stmt_n, \sigma_n\}}$$

where $e^*_{in} = \bigvee_{i=1}^{i=n} e^i_{in}, p^* = \bigwedge_{i=1}^{i=n} p^i, e^*_{out} = \bigvee_{i=1}^{i=n} e^i_{out}$ and $f^* = \bigwedge_{i=1}^{i=n} f(p)^i$.

We get as a result the configuration below which conforms to the waiting-state automaton:

$$\langle stmt_0, \sigma_0\rangle \xrightarrow[e'_{out},f']{e'_{in},p'} \langle stmt_n, \sigma_n\rangle$$

We do the same for all execution traces. The waiting state automaton for the producer is presented in Figure 7.

To conclude, the process is first converted to the control-flow graph, and then to the waiting-state automaton. Every transition is labeled with guards (events and conditions over variables), and the outputs (output events and actions over variables). Then, all the automata are composed together in order to build an automaton for the whole program.

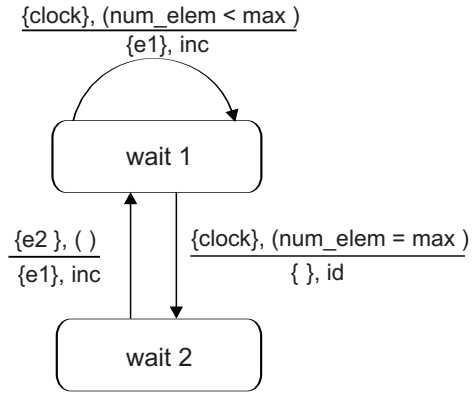


Figure 7: *The waiting-state automaton for the producer*

7. CONCLUSION

In this paper, we have presented a global approach how to automatically build the waiting-state automata: an abstract model that we presented in works Zhan et al. (2007); Harrath and Monsuez (2009). First, we use the stuctural operational semantics of Plotkin to present semantics of a subset

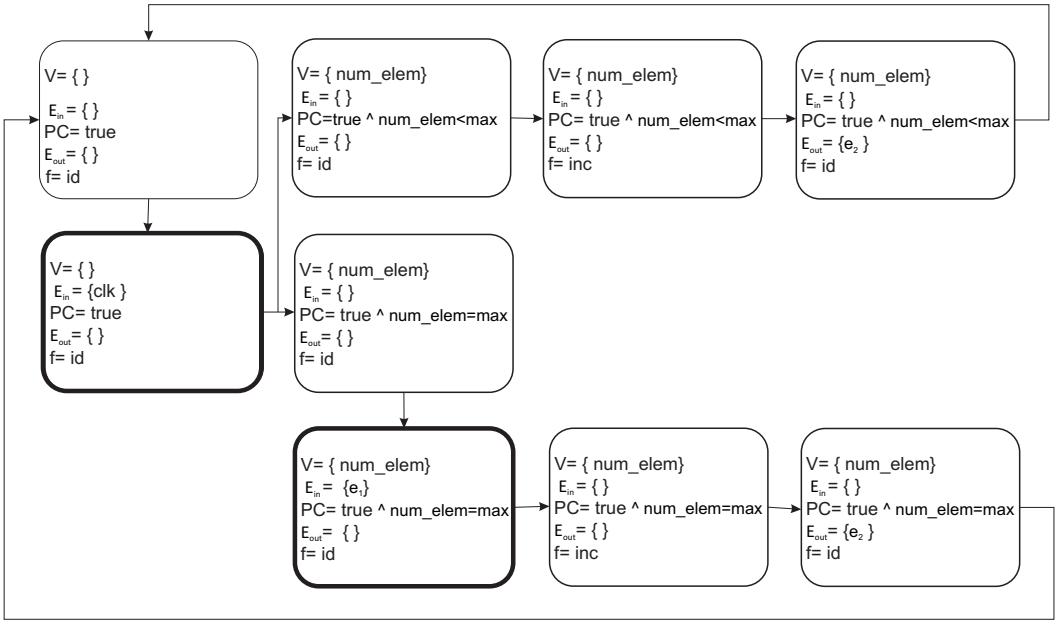


Figure 6: *Symbolic execution of the producer example*

of SystemC. The operational semantics can increase the correct understanding of a language and gives the possibility of formal reasoning. The semantics captured the behavioral semantics of SystemC components, we stress on the communications between threads within a delta cycle. Then, we use symbolic execution to present the effect of executing statements on system variables and events. Finally, we give a brief introduction about relations inference using predicate abstraction as introduced by Graf and Saidi (1997) which is a special case of abstract interpretation Cousot and Cousot (1977). We just introduced how to use predicate abstraction to merge traces between waiting states and then briefly how to infer relations between predicates.

Future Work

As future work, we plan to perform our formal semantics to handle all SystemC constructs. Another line of future work, which is more speculative, concerns predicate abstraction. One could investigate the use of CEGAR (counter example-guided abstraction refinement) techniques Clark et al. (2000) to arrive at useful relations inference between predicates.

8. REFERENCES

J.A. Darringer. *The application of program verification techniques to hardware verification*. Annual ACM IEEE Design Automation Conference, pages 376381, 1988.

J.C. King. *Symbolic execution and program testing*. Communications of the ACM (Association for Computing Machinery), 19(7), 1976.

P.Cousot and R.Cousot. *Abstract interpretation: a unified lattice model for the static analysis of programs by construction or approximation of fixpoints*, In Principles of programming languages. ACM, pages 238-252, 1977.

S. Graf and H. Saidi. *Construction of abstract state graphs with PVS*. In Proc. of the 9th International Computer Aided Verification. Springer Verlag, pages 72-83, 1997.

E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press,december 1999.

E. Clarke, O. Grumberg, S. Jha, Y. Lu and Helmut Veith. *Counterexample-Guided Abstraction Refinement*. In proceedings 12th International Conference on Computer Aided Verification, pp 154-169. Springer, 2000.

Drechsler, R. and Große, D. *Reachability analysis for formal verification of SystemC*. In Euromicro Symposium on Digital Systems Design, pp.337-340, 2002.

A. Gawanmeh, A. Habibi, and S. Tahar. *An executable operational semantics for systemc using abstract state machines*. Technical report, Department of Electrical and Computer Engineering, Concordia UniversityMontreal, March 2004.

W. Mueller, J. Ruf, and W. Rosenstiel. SystemC Methodologies and Applications. Kluwer Academic Publishers. 2003.

K.L. Man. $SystemC^{FL}$*: Formalization of SystemC*. In Proc of 12th IEEE Mediterranean Electrotechnical Conference, 2004.

Y.Zhang, F.Vdrine and B.Monsuez. *SystemC Waiting-State Automata*. On First International Workshop on Verification and Evaluation of Computer and Communication Systems. Algiers, Algeria, pages 5-6, eWiC, BCS, 2007.

N.Harrath, B.Monsuez. *Timed Waiting-State Automata*. On Third International Workshop on Verification and Evaluation of Computer and Communication Systems. Rabat, Morroco, eWiC, BCS, 2009.

D.Kroening and N.Sharygina. *Formal verification of SystemC by automatic hardware/software partitioning*. In the Third ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2005. Formal Methods and Models for Codesign, pages 101-110, 2005.

W.Mueller, J.Ruf, D.Hoffmann, J.Gerlach, T.Kropf, W.Rosenstiehl. *The Simulation Semantics of SystemC*. In Design, Automation and Test in Europe, IEEE CS, 2001.

F.Nielson, H.R.Nielson and C.Hankin. *Principles of Program Analysis*. Springer Verlag, 1999/2005.

G. D. 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 60-61:17139, 2004.

A. Salem. *Formal Semantics of Synchronous SystemC*.Proc. Design, Automation and Test in Europe Conference and Exposition, Munich, Germany, pp. 1037610381. 2003.

Niemann, B. and Haubelt, Ch. *Formalizing TLM with communicating state machines*. In Advances in Design and Specification Languages for Embedded Systems by Sorin A. Huss. pp. 225-242, 2007.

Karlsson, D. Eles, P. and , Peng Z. *Formal verification of SystemC Designs Using a Petri-Net Based Representation*. In Proceeding on the conference on Design, Automation and Test in Europe, pp.1228-1233, 2006.

F. Somenzi and A. Kuehlmann. *Equivalence checking*. In Louis Scheffer, Luciano Lavagno, and Grant Martin, editor, Electronic Design Automation For Integrated Circuits Handbook. CRC Press, 2006.

R.K. Shyamasundar, F. Doucet,R. Gupta, and I.H. Kruger. *Compositional Reactive Semantics of SystemC and Verification in RuleBase*, pages 152-169,2007.

P. Xiaoqing, Z. Huibiao, H. Jifeng and J. Naiyong. An Operational Semantics of an Event-Driven System-Level Simulator. *In proceedings of the 30th Annual IEEE/NASA Software Engineering Workshop*, pages 190-220, 2006.

Zhu, H. Linking the Semantics of a Multithreaded DiscreteEvent Simulation Language. PhD thesis, London South Bank University, UK, February 2005.

Herber, P., Pockrandt, M., Glesner, S. Automated conformance evaluation of SystemC designs using timed automata. *In European Test Symposium(2010)* 188-193