SystemC Waiting-State Automata

Yu Zhang
Project Everest, INRIA Sophia-Antipolis
2004, Route des Lucioles, BP 93, 06902 Sophia-Antipolis, France
Yu.Zhang@sophia.inria.fr

Franck Védrine
CEA, LIST, Boîte 65, Gif-sur-Yvette, F-91191 France
Franck.Vedrine@cea.fr

Bruno Monsuez
UEI, ENSTA, 32 Bd Victor, 75739 Paris cedex 15, France
Bruno.Monsuez@ensta.fr

Abstract

Delta-cycles are basic units of SystemC modeling and they are supposed to provide the guarantee of some critical properties about interactions between concurrent processes, like determinism and liveness, which is the basis for higher-level modeling and analysis. However, uncareful design may cause serious problems at the transaction level, which break the properties that we want to ensure at the level of delta-cycles. We propose a formal model based on SystemC waiting-state automata for verifying properties of SystemC models at the transaction level within a delta-cycle and show that this model conforms to the SystemC scheduler up to delta-cycles.

Keywords: SystemC, compositional verification, automata, model-checking

1. INTRODUCTION

SystemC becomes nowadays a popular language for modeling complex hardware systems. Compared with other hardware description languages, SystemC is more feasible for designing large-scaled complex systems and modeling high level behaviors — it comes with many pre-defined constructs in its syntax, like channels, modules, clocks, etc., which make the compositional system design much easier. SystemC is also equipped with a simulation engine which allows us to simulate a model once it is designed, but pure simulation is not sufficient if we want to ensure that the model satisfies certain properties, especially those safety-related properties. Simulation correctness does not imply logical correctness and verification is still a major issue.

SystemC simulation engine introduces the important notion of *delta-cycle* as the fundamental simulation unit. In fact, a simulation procedure can be seen as a sequence of delta cycles and all the interactions within a delta cycle are abstracted from the modeling perspective. Such an abstraction is supposed to provide a guarantee that all these interactions should work correctly so that higher-level analysis or verification can be done without considering what goes on inside delta-cycles. However, this is probably the most error-prone part of SystemC modeling, as the final process space within a delta-cycle can be very large and the interaction in between might be extremely complicated. A typical problem is the causality waiting cycles among processes, which causes the unexpected halt of the object system. Furthermore, accessing shared resources may cause competitions between processes, and consequently results in non-deterministic behavior at the delta-cycle level, which is certainly not desired in hardware design.

We propose in this paper a formal model for verifying SystemC up to delta-cycle. This is based on the observation that most important properties depends heavily on how processes get into and get out of a waiting state, which is actually controlled by the underlying event-based engine. We propose first a way of constructing small automata for single processes, by analyzing their waiting

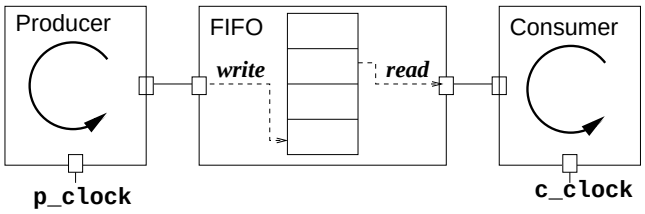

FIGURE 1: A simple FIFO

status. Then we define algorithms for composing the small process automata and reducing it so as to recognize the abstraction at the level of delta-cycle. These algorithms conform to the SystemC scheduler, which defines the executing semantics of SystemC. Verifications can be done at both the process automata and the final composed automata. We also discuss some extensions based on this model, like adding counters into the automata, so that additional properties can also be verified.

The rest of the paper is structured as follows: Section 2 is an introductory section on SystemC, through a simple example — a clocked FIFO model. We describe in particular how processes in a SystemC model communicate with each other and how the SystemC simulation engine schedules these processes. Then we introduce in Section 3 our formal model — the SystemC waiting-state automata and show how to derive them from single processes. In particular, we define the composition and the reduction at the symbolic level, so that the final automata can recognize the abstraction of the whole system at the level of delta-cycles. Section 4 introduces an extension of the waiting-state automata, by adding counters to include more information in the model and make it possible of verifying more properties. We also make some comparison, in Section 5, between our model and other formal models for SystemC and we conclude the paper in Section 6.

2. PRELIMINARY OF SYSTEMC

2.1. A simple FIFO model

Let us start by a simple SystemC model — an implementation of FIFO with clocks. The structure of the model is shown in Figure 1. The model contains a First-In-First-Out buffer and two modules cooperating through the buffer — a producer module which continuously puts data into the buffer and a consumer module which continuously retrieves data from the buffer. The two processes are triggered by two individual clocks: `p_clock` and `c_clock`. When a `p_clock` signal arrives, the producer starts producing and tries to write the product into the buffer. Similarly, the `c_clock` signals control the consumer. The two clocks are independent, hence the producing and the consuming can be at different paces.

It is certainly possible that the producer fills all the slots of the buffer and continues writing, or the consumer retrieves all the elements and still tries to consume. In this case, the producer/consumer must wait for the other to release/fill the buffer and this is done by the SystemC events mechanism. The implementation of the modules in SystemC is given in Figure 2.

2.2. Events

SystemC provides many different types of constructs in its syntax for system design, like modules, channels, ports, signals, clocks, etc. Despite of the diversity in syntax, SystemC is essentially an event-driven model, and all communications in SystemC models are implemented using events and the associated wait/notify mechanism. For instance, the FIFO module in the previous example is actually defined as a channel `sc_channel`, and the mutual access to the buffer is implemented through the two events `r_event` and `w_event`.

SystemC events can be roughly classified into the following three sorts:

```
class fifo_if : virtual public sc_interface{
 public:
  virtual int write(char) = 0;
  virtual int read(char &) = 0;
  virtual int num_available() = 0;
};

class fifo :
public sc_channel, public fifo_if {
 private:
  enum e { max = 2 };
  char buffer[max];
  int num_elem;

 public:
  sc_event w_event, r_event;

  fifo(sc_module_name name) :
    sc_channel(name), num_elem(0) {}

  void write(char c) {
    if (num_elem == max)
      wait(r_event);
    put_in_buffer(c);
    num_elem = num_elem + 1;
    w_event.notify();
  }

  void read(char &c){
    if (num_elem == 0)
      wait(w_event);
    c = get_from_buffer();
    num_elem = num_elem - 1;
    r_event.notify();
  }

  int num_available() { return num_elem;}
};
```

```
class producer : public sc_module {
 public:
  sc_port<fifo_if> fifo;
  sc_in_clk p_clock;

  SC_HAS_PROCESS(producer);
  SC_MODULE(producer) {
    SC_THREAD(main);
    sensitive_pos << p_clock;
  }

  void main() {
    while(true) {
      wait();
      produce(c);
      fifo->write(c);
    }
  }
};
```

```
class consumer : public sc_module {
 public:
  sc_port<fifo_if> fifo;
  sc_in_clk c_clock;

  SC_HAS_PROCESS(consumer);
  SC_MODULE(consumer){
    SC_THREAD(main);
    sensitive_pos << c_clock;
  }

  void main() {
    while(true) {
      wait();
      fifo->read(c);
      consume(c);
    }
  }
};
```

FIGURE 2: The FIFO modules: buffer, producer and consumer

- user-defined events. These are events defined by SystemC programmers in source code. For instance, the `w_event` and the `r_event` as in Figure 2. Such events are usually triggered by the command `notify`;
- channel events. These are pre-defined SystemC events and they are triggered when something occurs on channels. For instance, an event denoting the arrival of a new value will be triggered whenever some value is written to a `sc_buffer` channel. Channel events at different types of channels have different semantics [5, 6];
- clock events[1]. Clock signals are also seen as events and they are usually defined as `sc_clock`'s in the SystemC main program. SystemC core engine is in charge of generating `sc_clock` events at proper time. We never notify a clock event in the program.

We shall not distinguish the first two sorts of events as both of them can be dynamically notified, while the clock events are rather seen as the input events of SystemC models. In fact, we prefer not considering any pre-defined channels or signals in our modeling, but rather taking into account their event-driven implementation. For instance, the FIFO buffer in the previous example is defined as a channel with two actions (`write()` and `read()`) and two internal events (`w_event` and `r_event`), and modules connected to FIFO are allowed to call the channel methods but they are not aware of the existence of the channel events. However, for our analysis within delta-cycles, we shall put the method definitions inline in the codes of processes (see the analysis in the beginning of Section 3).

2.3. SystemC simulation 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. The scheduler maintains several tables, among which we are particularly interested in the table of runnable processes (processes that are ready to execute at the current delta-cycle).

Here is a brief description of a delta-cycle: a delta-cycle starts with a non-empty runnable process table. The scheduler executes these processes one by one, in a pre-defined order; every runnable process executes until it ends or it is pended again (by a `wait` command for instance); if any immediate event (e.g. the `w_event` generated by the producer process in the FIFO example) is notified during the execution of a runnable processes, it will add processes that are currently sensitive to this event into the runnable process table; delta-events and timed events that are generated during the execution of a process are stored in other tables. The process table is emptied when all runnable processes are executed and the procedure of executing all the runnable processes is called a *evaluation phase*. The scheduler then checks those delta-events notified in the evaluation phase: if there are processes that are sensitive to these events, then add them into the process table. This procedure is called a *delta notification phase*. If the process table is non-empty, the scheduler enters the next delta-cycle and executes the evaluation phase again; otherwise, it checks the timed events notified in the evaluation phase and adds processes that sensitive to these timed events into the process table. This is called a *timed notification phase*. The scheduler then advances the simulation clock and enters the next delta-cycle.

Above is what is defined as delta-cycles in SystemC scheduler. A detailed explanation and implementation of delta-cycles can be found in SystemC documents [5, 6]. However, for the sake of clarification, we prefer regarding a delta-cycle as starting from a delta notification phase or a timed notification phase. In other words, a delta-cycle in this paper will start with a set of events which add all sensitive processes into the process table, then continue with an evaluation phase.

Take the simple FIFO as an example. Suppose that at the beginning of a delta-cycle we have both `p_clock` signal and `c_clock` signal, which will be seen as events in SystemC. The `p_clock` event then adds the producer process into the process table if the producer is waiting for the clock

[1]Clocks are actually seen as typical channels in SystemC — `sc_clock` is implemented using `sc_signal`, but from the analysis viewpoint, we would better differentiate them from channel events.

signal. Note that the producer may wait for a `r_event` instead of a clock signal, and in this case, it will not be added into the process table. Similarly, the `c_clock` event will add the consumer into the process table if it is waiting for the `c_clock` signal. The scheduler then runs the processes in a pre-defined order. For instance, if we run the producer first then the consumer, according to the status of the buffer, there are then two cases of what happens within a delta-cycle:

- if the buffer is not full, the producer will succeed in putting a new product into the buffer, get hung up and wait for the next clock signal. The scheduler moves on to run the consumer until it is pended. During this procedure, two extra events are generated — `w_event` and `r_event` and they are added into the event table, but as no process is sensitive to them, nothing is added into the process table and the evaluation phase is over, as well as the present delta-cycle;
- if the buffer is full, the producer will be hung up and waits for a `r_event`. The scheduler continues to run the consumer. This generates a `r_event`, which will add immediately the producer process into the process table again, since it is now sensitive to `r_event`. The scheduler then resumes the execution of the producer, until it is pended for the next clock signal. The evaluation phase, as well as the current delta-cycle, stops here. Note that if there is no `c_clock` at the same cycle, the delta-cycle just stops when the producer is pended for the first time (when it waits for the `r_event`).

2.4. Cycles in SystemC scheduling engine

Cycles, like delta-cycle in the SystemC simulation engine, are important to the verification of SystemC models. But cycles are not explicitly defined in the SystemC syntax, and there are usually multiple levels of cycles in a SystemC model.

We give a rough classification here on what can be seen as a cycle in SystemC models:

- the *minimal cycle*, which is basically the interval between two continuous `wait` within a single process. We call these cycles the *minimal* cycles, because we are not interested in the detailed execution of the program. There are of course smaller cycles, e.g., a cycle inside a loop which does not contain any `wait`, but they might be of little interest for SystemC verification. A minimal cycle is usually used to abstract the transitions between two waiting states within a single process;
- the *delta-cycle*, as defined in the SystemC scheduler. A delta-cycle contains usually multiple minimal-cycles. Abstractions at this level of cycles should provide some fundamental properties about interactions between processes;
- the *gcd-cycle*, i.e., the *greatest common divisor* of all clocks. For instance, in the example of simple FIFO, the gcd-cycle might be defined as $gcd(\texttt{p\_clock}, \texttt{c\_clock})$. With this definition, one can concentrate on behaviors at a higher level, and it is particularly useful for verifying temporal properties, but it remains to clarify how to abstract the behavior of a SystemC model at this level, when a gcd-cycle contains multiple delta-cycles.

The classification here is certainly not exhausted, but it should contain those cycles that appear frequently in most SystemC models. Clearly, these cycles provide different levels of abstractions and can be useful for verifications for particular purposes.

2.5. Liveness and determinism

Abstractions at the level of delta-cycles should guarantee some fundamental properties of interactions between concurrent processes. Among others, we cite two properties here: *liveness* and *determinism*.

The liveness property is probably the most common one in the field of formal verification. It simply states that the implementation must be deadlock-free, and in SystemC modeling, it means that there is no causality waiting cycles between processes. For instance, in the FIFO example, the producer and the consumer should not wait for each other.

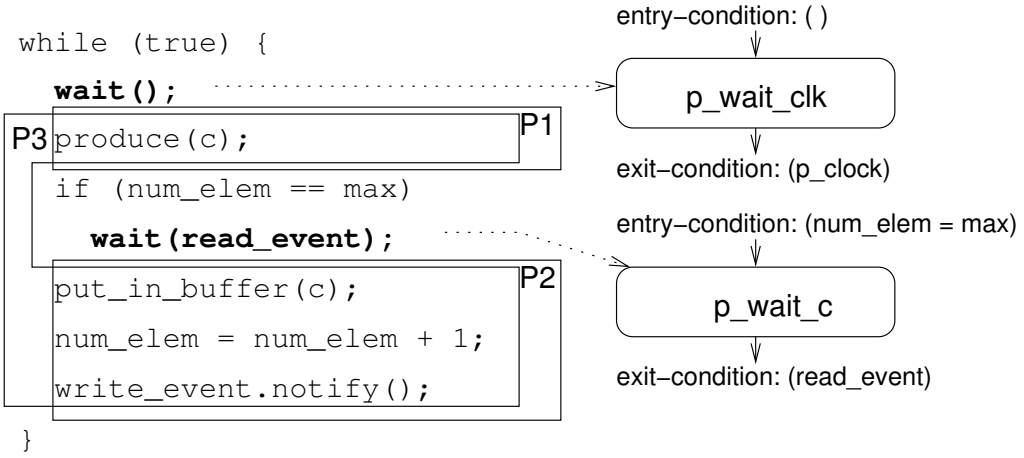

FIGURE 3: Analysis of the producer module

Determinism is a very critical property for hardware design. In SystemC modeling, a deterministic SystemC model should ensure that the behavior is independent of the order of internal process executions. It stresses in particular the determinism at the level of delta-cycles, because non-deterministic behaviors are caused by competitions when multiple processes are accessing shared resources at the same time, and such competitions usually occur within a single delta-cycle.

As an example, consider another implementation of FIFO: the producer/consumer does not wait for the other to release/fill the buffer when it is full/empty, and instead, they just return a write/read failure. It is clear that there might be a competition between the producer and the consumer: if at the beginning of a delta-cycle, both producer and consumer are allowed to operate on the buffer (both `p_clock` and `c_clock` are present), we might have different results. For instance, if the buffer is full, producing first will cause the new product to be discarded, while consuming first will cause a successful writing to buffer. Non-determinism is hard to detect through simulation, because the SystemC scheduler will fix an execution order for process, though semantically, it should not be fixed. For instance, it may always execute consumer first, then the simulation will return the same result with the same input and we cannot see the non-deterministic behavior from the simulation.

3. SYSTEMC WAITING-STATE AUTOMATA

This section introduces a formal model of SystemC, based on the analysis of the "wait/notify" mechanism of SystemC, which plays an important role in the SystemC scheduler and serves as the basic engine of implementing interactions between processes. The basic idea of our model is to consider only those 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.

Let us start by an informal analysis of the producer process, as shown in Figure 3. The two `wait` statements define the two waiting-states of the automaton, and they divide the program into three pieces (P_1, P_2, P_3 in Figure 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. One should not confuse these conditions with pre-conditions/post-conditions of programs: the pre/post-condition assertion says that if a pre-condition of a program P holds before the execution of P, then after the execution, the post-condition holds; the entry/exit-conditions are rather seen as the guard condition for starting/stopping the execution of the `wait` statements. The transition from one state to another is then determined by the condition of exiting the source

state plus the condition of entering the destination state. For instance, the condition for entering the p_wait_c state is num_elem == max and the condition for exiting the p_wait_clk state is a p_clock signal, so the guard condition for the transition from p_wait_clk to p_wait_c is an event p_clock plus a predicate (num_elem == max), defined on the variable num_elem (max is a pre-defined constant here). 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 w_event and make the variable num_elem decreased by 1. These are the effects of transitions.

3.1. SystemC waiting-state automata

Notice that transitions between waiting-states depend not only on the events, but also on variables, especially those variables shared by several processes, e.g., the variable num_elem in the FIFO example. Let $\mathcal{V}$ be a finite set of variables $\{x_1, x_2, \ldots, x_n\}$, where every variable x_i has a domain D_i which contains the values that can be assigned to x_i. $D_1 \times \cdots \times D_n$ is called the domain of $\mathcal{V}$, abbreviated as $\vec{D}$. We call a function $f : \vec{D} \to \vec{D}$, defined over these variables, an *effect function over* $\mathcal{V}$.

Definition 1 *A* SystemC waiting-state automaton*, over a set $\mathcal{V}$ of variables, is a triple $A(\mathcal{V}) = (\mathcal{S}, \mathcal{E}, \mathcal{T})$, where $\mathcal{S}$ is a finite set of states, $\mathcal{E}$ is a finite set of events, $\mathcal{T}$ 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 $\mathcal{S}$, representing respectively the initial state and the end state of the transition;*
- *e_{in} and e_{out} are two sets of events: $e_{in} \subseteq \mathcal{E}, e_{out} \subseteq \mathcal{E}$;*
- *p is a predicate defined over variables in $\mathcal{V}$, 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 $(s, e_{in}, p, e_{out}, f, s')$. The effect function set $\mathcal{F}(A)$ of the automaton $A(\mathcal{V})$ is the set of all effect functions in $A(\mathcal{V})$: $\mathcal{F}(A) = \{f \mid \exists t \in \mathcal{T}$ s.t. $\mathsf{proj}_6^5(t) = f\}$, where proj_6^5 denotes the fifth projection of a 6-tuple. We also use $\mathcal{T}(s)$ to denote the set of transitions from a given state s, i.e., $\mathcal{T}(s) = \{t \mid t \in \mathcal{T}$ and $\mathsf{proj}_6^1(t) = s\}$.

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 represents an effect: the transition will generate all the events in e_{out} and f will be applied to the current instantiation of $\mathcal{V}$.

Naturally, we expect that a SystemC automaton represents faithfully the process from which it is derived. However, in a SystemC process, the transition from a waiting-state to another is only triggered by the events and the predicates determine which state the process will enter after being wake up, which means that transition from the same state must have the same set of incoming events (e_{in}). We say that a SystemC waiting-state automaton $A(\mathcal{V}) = (\mathcal{S}, \mathcal{E}, \mathcal{T})$ is *faithful* if for every two transitions t, t',

$$\mathsf{proj}_6^1(t) = \mathsf{proj}_6^1(t') \Rightarrow \mathsf{proj}_6^2(t) = \mathsf{proj}_6^2(t'),$$

and for every state $s \in \mathcal{S}$, $\bigvee_{t \in \mathcal{T}(s)} \mathsf{proj}_6^3(t)$ always holds.

The automata for the processes of the producer and the consumer are shown in Figure 4. where we consider only one variable num_elem and the definitions of the effect functions inc and dec are obvious:

$$\begin{aligned} inc(\texttt{num\_elem}) &= \texttt{num\_elem} + 1, \\ dec(\texttt{num\_elem}) &= \texttt{num\_elem} - 1. \end{aligned}$$

The function id is the identity function and the empty predicate can be seen as the truth constant. Clearly, these two automata are faithful if we consider the domain of the variable num_elem as the interval $[0, \texttt{max}]$.

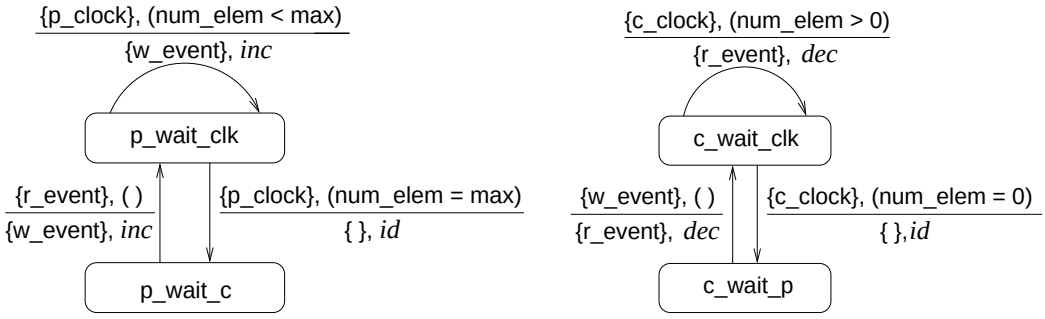

FIGURE 4: The automata for the producer and the consumer

In fact, since the two automata are derived by analyzing the waiting states of processes, as we have seen in the beginning of this section, a transition actually represents the execution within a minimal cycle, i.e., the execution of a process between two continuous `wait`. We call such an automaton a *minimal-step waiting-state automata*. We assume that every mininmal-step automaton is *total*, i.e., every state has some successor. Otherwise, it means that the single process itself may cause a deadlock, which is not the case we study here.

3.2. Symbolic composition and reduction

The global strategy of verifying SystemC models using waiting-state automata is to define first a minimal-step automaton for every process, and compose them together so as to achieve a bigger automaton for the whole SystemC model which can be finally passed to the model-checking procedure.

The symbolic composition follows the parallel composition of labeled Kripke structures [1] derived from CSP. However, our symbolic composition is followed by a reduction procedure, which enables more aggressive abstractions on the result model. Abstraction is a common way to handle more and more complex system. Other typical abstractions include, for instance, replacing internal events of the composition with more abstract notions like counters and constraints on counters. With respect to the hiding of variables [8], such replacements keep functionality properties after the abstraction and to introduce progressively QoS properties.

3.2.1. Symbolic composition

Definition 2 *Given two SystemC waiting-state automata* $A(\mathcal{V}) = (\mathcal{S}, \mathcal{E}, \mathcal{T})$ *and* $A'(\mathcal{V}) = (\mathcal{S}', \mathcal{E}', \mathcal{T}')$*, over the same set* $\mathcal{V}$ *of variables, the* combination of the two SystemC waiting-state automata *is still a SystemC waiting-state automaton* $(\mathcal{S} \times \mathcal{S}', \mathcal{E} \cup \mathcal{E}', \mathcal{T}'')$ *(written as* $A(\mathcal{V}) \times A'(\mathcal{V})$*) where* $\mathcal{T}''$ *is the smallest set of transitions such that:*

- $(s_1, s_1') \xrightarrow[e_{out},\ f]{e_{in},\ p} (s_2, s_1') \in \mathcal{T}''$*, for every transition* $s_1 \xrightarrow[e_{out},\ f]{e_{in},\ p} s_2 \in \mathcal{T}$*, and for every state* $s_1' \in \mathcal{S}'$ *such that for every transition* $s_1' \xrightarrow[e'_{out},\ f']{e'_{in},\ p'} s_2' \in \mathcal{T}'$*, either* $e'_{in} \not\subseteq e_{in}$ *or* $p \not\Rightarrow p'$*;*
- $(s_1, s_1') \xrightarrow[e'_{out},\ f']{e'_{in},\ p'} (s_1, s_2') \in \mathcal{T}''$*, for every transition* $s_1' \xrightarrow[e'_{out},\ f']{e'_{in},\ p'} s_2' \in \mathcal{T}'$*, and for every state* $s_1 \in S$ *such that for every transition* $s_1 \xrightarrow[e_{out},\ f]{e_{in},\ p} s_2 \in \mathcal{T}$*, either* $e_{in} \not\subseteq e'_{in}$ *or* $p' \not\Rightarrow p$*;*
- $(s_1, s_1') \xrightarrow[e_{out} \cup e'_{out},\ f \circ f']{e_{in} \cup e'_{in},\ p \wedge p'} (s_2, s_2') \in \mathcal{T}''$*, for every pair of transitions* $s_1 \xrightarrow[e_{out},\ f]{e_{in},\ p} s_2 \in \mathcal{T}$ *and* $s_1' \xrightarrow[e'_{out},\ f']{e'_{in},\ p'} s_2' \in T'$*.*

The composed automaton of the two minimal-step automata in Figure 4 can be found in [14].

Note that in the last case of defining transitions of the composed automaton, we can replace the effect function $f \circ f'$ of the new transition by $f' \circ f$, but the composed automata might not be equivalent since $f \circ f'$ and $f' \circ f$ are not always equal. This is the case where the composition will result in potential non-deterministic behaviors and we call a transition with an effect function $f \circ f'$ such that $f \circ f' \neq f' \circ f$ a *non-deterministic transition*.

The symbolic composition of SystemC automata can be used to check the determinism of a SystemC model: we define the corresponding minimal-step automaton for every process and compose them together; if the composed automata does not contain any non-deterministic transition, we can then assert that the model is deterministic.

Detecting non-deterministic transitions can be done without doing the composition. Because the above definition includes all composes of effect functions of the two component automata, we can simply check whether $f \circ f' = f' \circ f$, where $f \in \mathcal{F}(A), f' \in \mathcal{F}(A')$ (A, A' are two component automata). However, such a detection might be too strict in the sense that some non-deterministic transitions may never be triggered and does not change the deterministic behavior of the model. This is often because the guard condition of these "impossible transitions" will never be true, e.g., p and p' in the above definition are two contradicting predicates. Actually, such transitions can be removed after the composition as a refinement, and clearly, checking the non-existence of non-deterministic transitions based on the refined composition will increase the precision of the detection of the non-determinism of SystemC models.

SystemC processes running in parallel may cause causality waiting cycles and it is also represented in the composition of SystemC waiting-state automata. This is related the liveness property mentioned in Section 2 and the verification requires in the first place the detection of "unsafe states" containing mutually waiting processes, for further analysis based on the automata, e.g., the reachability analysis.

First, for every state s in a SystemC automaton, write $e_{in}(s)$ for the set $\bigcup_{t \in \mathcal{T}(S)} \mathsf{proj}_6^2(t)$ (the set of events that trigger transitions from this state), and $e_{out}(s)$ for the set $\bigcup_{t \in \mathcal{T}(S)} \mathsf{proj}_6^4(t)$. Consider two minimal-step automata $A_1(\mathcal{V}) = (\mathcal{S}_1, \mathcal{E}_1, \mathcal{T}_1)$ and $A_2(\mathcal{V}) = (\mathcal{S}_2, \mathcal{E}_2, \mathcal{T}_2)$. We say that a state (s_1, s_2) ($s_1 \in S_1, s_2 \in S_2$) in the composed automaton $A_1 \times A_2$ is a *potential unsafe state* if $e_{in}(s_1) \cap e_{out}(s_2) \neq \emptyset$ and $e_{in}(s_2) \cap e_{out}(s_1) \neq \emptyset$. For instance, in the FIFO example, the composition of the producer automaton and the consumer automaton (Figure 4) gives rise to a potential unsafe state $(\texttt{p\_wait\_c}, \texttt{c\_wait\_p})$. Indeed, it is an unsafe state where the two processes are waiting for each other.

Potential unsafe states can be effectively detected when the component automata are given, but this is still a very coarse detection in the sense that potential unsafe states are not necessarily unsafe. A precise detection of unsafe states require probably user interference.

Definition 2 of automata composition can be extended in a regular way to the case of more than two processes, but the definition would be much more complicated, so we rest on the version of composing two automata and the composition of more that two automata can be done by applying repeatedly the composing in Definition 2.

3.2.2. Symbolic reduction

In a SystemC waiting-state automaton, it is possible that the effect of a certain transition will immediately trigger another transition, e.g., the following two transitions

$$s_1 \xrightarrow[e_{out},\, f]{e_{in},\, p} s_2 \xrightarrow[e'_{out},\, f']{e'_{in},\, p'} s_3 \ ,$$

where $e'_{in} \subseteq e_{out}$ and $p \Rightarrow f(p)$ ($f(p)$ stands for the predicate $p[\mathsf{proj}_n^1(f(\vec{x}))/x_1, \ldots, \mathsf{proj}_n^n(f(\vec{x}))/x_n]$). In this case, we can replace the two transitions by a new transition $s_1 \xrightarrow[e'_{out},\, f' \circ f]{e_{in},\, p} s_3$. We call the previous pair of transitions (t_1, t_2) a *reducible*

pair, where $t_1 = (s_1, e_{in}, p, e_{out}, f, s_2)$ and $t_2 = (s_2, e'_{in}, p', e'_{out}, f', s_3)$. The new transition $(s_1, e_{in}, p, e'_{out}, f' \circ f, s_3)$ is called the *contractum* of (t_1, t_2). In the FIFO example, we can find examples of such reducible transitions (see [14] for details). Since the automaton is composed from two small automata representing two processes, such a reduction actually represents an interaction between the two processes within a single delta-cycle.

In general, a minimal-step waiting-state automaton does not contain any reducible transitions, and in our verification strategy, reductions are usually required when we compose all the minimal-step automata together. As we are intend to define a model that represents the behavior at the level of delta-cycles and hides all interactions within a delta-cycle, the reduction algorithm should be consistent with the SystemC scheduler.

Algorithm 1 (Symbolic reduction) *Given a SystemC minimal-step waiting-state automaton $A(\mathcal{V}) = (\mathcal{S}, \mathcal{E}, \mathcal{T})$, where $\mathcal{T}$ has reducible transitions, let $\mathcal{T}_0 := \mathcal{T}$, $\mathcal{T}_{remove} := \{\}$ and $\mathcal{T}_{new} := \{\}$. The following steps define an algorithm of removing the reducible transitions:*

1. *for every reducible pair (t_1, t_2) and its contractum t_3, where $t_1, t_2 \in \mathcal{T}_0$, let $\mathcal{T}_{remove} := \mathcal{T}_{remove} \cup \{t_1\}$ and $\mathcal{T}_{new} := \mathcal{T}_{new} \cup \{t_3\}$;*
2. *repeat the above step until all reducible pairs in $\mathcal{T}_0$ has been manipulated;*
3. *let $\mathcal{T}_0 := (\mathcal{T}_0 / \mathcal{T}_{remove}) \cup \mathcal{T}_{new}$, $\mathcal{T}_{remove} := \{\}$ and $\mathcal{T}_{new} := \{\}$;*
4. *if there are still reducible pairs in $\mathcal{T}_0$, go to the step 1 and repeat the above procedure; otherwise, let $\mathcal{T}' := \mathcal{T}_0$.*

The reduced automaton is $(\mathcal{S}, \mathcal{E}, \mathcal{T}')$.

[14] gives an example of this reduction on the FIFO model.

If we are given a SystemC model with a set of parallel processes $\{P_1, \ldots, P_n\}$, let $A_1(\mathcal{V}), \ldots, A_n(\mathcal{V})$ be the corresponding SystemC minimal-step waiting-state automata of all the processes in the model, and let $A(\mathcal{V}) = (\mathcal{S}, \mathcal{E}, \mathcal{T})$ be the reduced automaton of $A_1 \times \cdots \times A_n$ by Algorithm 1, then every transition $s_1 \xrightarrow[e_{out},\ f]{e_{in},\ p} s_2$ in T represents a whole execution of the SystemC model within a delta-cycle, in the sense that if at the beginning of a delta-cycle the model is in the state s_1 and the predicate p holds, and if all events in e_{in} are provided by the environment, then in the end of the delta-cycle, the model will be in the state s_2.

A formal verification of SystemC models can be done at the level of delta-cycles, using the SystemC waiting-state automata, together with the composition and the reduction algorithms. Note that at this stage, we might divide the events in $\mathcal{E}$ of the final automaton into two sets — the set of environment events and the other of internal events. The environment events are events generated by the SystemC engine, which are typically “time events” such as clock events. By distinguishing between these two sorts of events, we can again reduce the automaton by removing those transitions depending on non-environment events, i.e., transitions whose e_{in} contains events that are not in the set of environment events.

Besides the automaton, the model-checking procedure requires in addition a sequence of event sets, every set of which contains the environment events that the SystemC engine generate at the beginning of the corresponding delta-cycle. Then we pass this sequence and the automaton of the SystemC model to a model-checking tool as in [10, 8], and check whether those unsafe states can be reached.

4. EXTENDING SYSTEMC WAITING-STATE AUTOMATA WITH COUNTERS

By counting how many times a transition is triggered during the execution of an automaton, we obtain more information about the model and consequently, we are able to verify additional relevant properties for system design. This can be done by adding a counter for every transition in the automaton.

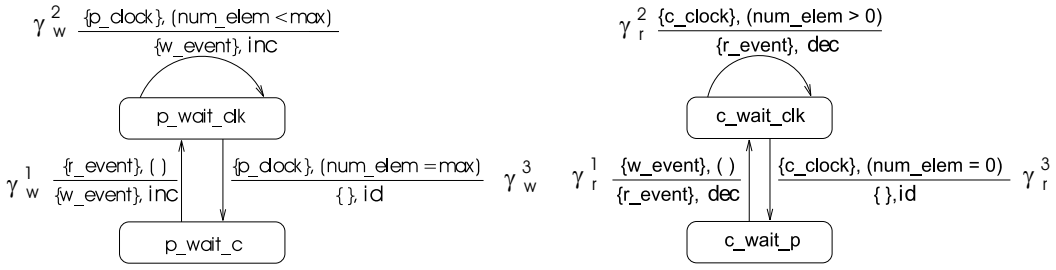

FIGURE 5: The automata for the producer and the consumer

Again, let us take the FIFO example and consider the two small automata for the processes of the producer and the consumer. Figure 5 shows both automata annotated with counters. For instance, $\gamma^3_{\mathtt{w}}$ counts the times of executing the transition from `p_wait_clk` to `p_wait_c`, and it is actually the number of times that the producer enters the waiting-state `p_wait_c`. These counters provide meaningful information about the behavior of the whole system. For instance, a "read" can only occur if a "write" occurs before it, and this property can be represented using counters, i.e., the total count of transitions representing a "read" is always less or equal to the total count of those representing a "write". Formally, it should always hold that:

$$\gamma^1_{\mathtt{r}} + \gamma^2_{\mathtt{r}} \leq \gamma^1_{\mathtt{w}} + \gamma^2_{\mathtt{w}}$$

We can furthermore formulate some properties that may require more precise information, using both counters and system variables, e.g., the following property:

$$(\gamma^1_{\mathtt{w}} + \gamma^2_{\mathtt{w}}) - (\gamma^1_{\mathtt{r}} + \gamma^2_{\mathtt{r}}) \leq \mathtt{num\_elem}$$

Also, the entry-conditions and exit-conditions presented in section 3.1 can be extended by adding conditions on transition counters. For instance, the condition for exiting the `c_wait_clk` state is a `c_clock` signal plus the condition $\gamma^1_{\mathtt{r}} = \gamma^3_{\mathtt{r}} - 1$.

Such relations on counters can be inferred during the construction stage of the waiting-state automaton using for instance techniques based on abstract interpretation as in [12, 13, 11]. Abstract interpretation can also be done after the construction stage. In [9], Lustre descriptions are generated from synchronous SystemC so that the tool NBAC verification and slicing tool can be applied [7].

Definition 3 *An extended SystemC waiting-state automaton over a set $\mathcal{V}$ of variables, is a quadruple $A(\mathcal{V}) = (\mathcal{S}, \mathcal{E}, \mathcal{T}, \mathcal{C})$, where $\mathcal{S}$ is a finite set of states, $\mathcal{E}$ is a finite set of events, $\mathcal{C}$ is a set of transition counters, $\mathcal{T}$ is a finite set of transitions where every transition is a 7-tuples $(s, e_{in}, p, e_{out}, f, s', \gamma)$:*

- *s and s' are two states in $\mathcal{S}$, representing respectively the initial state and the end state of the transition;*
- *e_{in} and e_{out} are two sets of events: $e_{in} \subseteq \mathcal{E}, e_{out} \subseteq \mathcal{E}$;*
- *p is a predicate defined over variables in $\mathcal{V}$ and counters $\mathcal{C}$, i.e., $FV(p) \subseteq \mathcal{V} \cup \mathcal{C}$;*
- *f is an effect function over $\mathcal{V}$.*
- *$\gamma \in \mathcal{C}$ is the counter associated to the transition that increments each time it is executed.*

Our strategy of verifying SystemC models using extended waiting-state automata is similar as before, which requires first building a minimal-step automaton for every process, inferring relations over transitions counters and finally composing them together to build a big automaton that can be passed to a model-checking procedure.

The symbolic composition of two automata $A(\mathcal{V})$ and $A'(\mathcal{V})$ works similarly as in the case of standard waiting-state automata. What makes the composition more complicated is that counters of the automata $A(\mathcal{V})$ and $A'(\mathcal{V})$ do not correspond to the counters of the composed automaton

$A(\mathcal{V}) \times A'(\mathcal{V})$. A transition t of the automata $A(\mathcal{V})$ may be composed with many transitions $\{t'_1, \ldots, t'_n\}$ of the automaton $A'(\mathcal{V})$, and the value of γ_t the transition t in $A(\mathcal{V})$ should be represented in the values of counters for $(t \times t'_1), \ldots, (t \times t'_n)$. If $\gamma_t^{t'_k}$ denotes the counter associated to the transitions $(t \times t'_k)$ in the composed automaton, then $\gamma_t = \sum_{k=1}^{n} \gamma_t^{t_k}$. As the transition predicates (i.e., guard conditions) in extended waiting-state automata are defined over the counters and system variables, the composition should ensure that these predicates of component automata are properly translated in the composed automaton. In particular, we must replace all the occurrence of a transition counter γ_t from component automata, with the sum of the associated transition counters $\sum_{k=1}^{n} \gamma_t^{t'_k}$ in the composed automaton.

Definition 4 *Given two SystemC waiting-state automata $A(\mathcal{V}) = (\mathcal{S}, \mathcal{E}, \mathcal{T}, \mathcal{C})$ and $A'(\mathcal{V}) = (\mathcal{S}', \mathcal{E}', \mathcal{T}', \mathcal{C}')$, over the same set $\mathcal{V}$ of variables, the* combination of the two SystemC waiting-state automata *is still a SystemC waiting-state automaton $(\mathcal{S} \times \mathcal{S}', \mathcal{E} \cup \mathcal{E}', \mathcal{T}'', \mathcal{C}''))$ (written as $A(\mathcal{V}) \times A'(\mathcal{V})$) where $\mathcal{T}''$ is the smallest set of transitions, $\mathcal{C}''$ is the associated set of counters, $\Gamma(s, e_{in}, p, e_{out}, f, s', \gamma)$ is the set of counters of $\mathcal{C}''$ associated to a transition in $\mathcal{T} \cup \mathcal{T}''$ and $\mathcal{M}$ a morphism that maps the counters in $\mathcal{C} \cup \mathcal{C}'$ to $\mathcal{C}''$ such that:*

- $\Gamma(s_1, e_{in}, p, e_{out}, f, s_2, \gamma) = \{\gamma^*\} \cup \Gamma(s_1, e_{in}, p, e_{out}, f, s_2, \gamma)$ *and* $(s_1, s'_1) \xrightarrow[e_{out},\ f]{e_{in},\ \mathcal{M}(p),\ \gamma^*} (s_2, s'_1) \in \mathcal{T}''$, *for every transition* $s_1 \xrightarrow[e_{out},\ f]{e_{in},\ p, \gamma} s_2 \in \mathcal{T}$, *and for every state* $s'_1 \in \mathcal{S}'$ *such that for every transition* $s'_1 \xrightarrow[e_{out},\ f']{e_{in},\ p', \gamma'} s'_2 \in \mathcal{T}'$, *either* $e'_{in} \not\subseteq e_{in}$ *or* $p \not\Rightarrow p'$;
- $\Gamma(s'_1, e_{in}, p, e_{out}, f, s'_2, \gamma') = \{\gamma^*\} \cup \Gamma(s'_1, e_{in}, p, e_{out}, f, s'_2, \gamma')$ *and* $(s_1, s'_1) \xrightarrow[e_{out},\ f]{e_{in}\ \mathcal{M}(p)\ \gamma^*} (s_1, s'_2) \in \mathcal{T}''$, *for every transition* $s'_1 \xrightarrow[e_{out},\ f]{e_{in},\ p, \gamma'} s'_2 \in \mathcal{T}'$, *and for every state* $s_1 \in \mathcal{S}'$ *such that for every transition* $s_1 \xrightarrow[e_{out},\ f]{e_{in},\ p, \gamma} s_2 \in \mathcal{T}$, *either* $e_{in} \not\subseteq e'_{in}$ *or* $p \not\Rightarrow p'$;
- $\Gamma(s_1, e_{in}, p, e_{out}, f, s_2, \gamma) = \{\gamma^*\} \cup \Gamma(s_1, e_{in}, p, e_{out}, f, s_2, \gamma)$ *and* $\Gamma(s'_1, e_{in}, p, e_{out}, f, s'_2, \gamma') = \{\gamma^*\} \cup \Gamma(s'_1, e_{in}, p, e_{out}, f, s'_2, \gamma')$ *and* $(s_1, s'_1) \xrightarrow[e_{out} \cup e'_{out},\ f \circ f']{e_{in} \cup e'_{in},\ \mathcal{M}(p \wedge p'),\ \gamma^*} (s_2, s'_2) \in \mathcal{T}''$, *for every pair of transitions* $s_1 \xrightarrow[e_{out},\ f]{e_{in},\ p, \gamma} s_2 \in \mathcal{T}$ *and* $s'_1 \xrightarrow[e_{out},\ f]{e_{in},\ p, \gamma'} s'_2 \in \mathcal{T}'$
- *for each counter γ associated to the transition $(s_1, e_{in}, p, e_{out}, f, s_2, \gamma_{s_1}^{s_2})$, the morphism $\mathcal{M}$ maps the counter γ to the sum of transition counters defined in $\Gamma(s_1, e_{in}, p, e_{out}, f, s_2, \gamma)$*

$$\mathcal{M}(\gamma) \longrightarrow \sum_{\gamma^* \in \Gamma(s_1, e_{in}, p, e_{out}, f, s_2, \gamma_{s_1}^{s_2})} \gamma^*$$

The algorithm that performs symbolic reduction on extended waiting-state automata simply merges those counters whose associated transitions are reduced.

Algorithm 2 (Symbolic reduction) *Given a SystemC minimal-step waiting-state automaton $A(\mathcal{V}) = (\mathcal{S}, \mathcal{E}, \mathcal{T}, \mathcal{C})$, where $\mathcal{T}$ has reducible transitions, let $\mathcal{T}_0 := \mathcal{T}$, $\mathcal{C}_0 := \mathcal{C}$, let $\mathcal{T}_{remove}, \mathcal{T}_{new}, \mathcal{C}_{remove}, \mathcal{C}_{new} := \{\}$. The following steps define an algorithm of removing the reducible transitions:*

1. *for every reducible pair (t_1, t_2) and its contractum t_3, where $t_1, t_2 \in \mathcal{T}_0$, let $\mathcal{T}_{remove} := \mathcal{T}_{remove} \cup \{t_1\}$ and $\mathcal{T}_{new} := \mathcal{T}_{new} \cup \{t_3\}$;*
   let $\mathcal{C}_{remove} = \mathcal{C}_{remove} \cup \{$ the counters associated to t_1 and $t_2\}$, $\mathcal{C}_{new} = \mathcal{C}_{new} \cup \{$the counter associated to $t_3\}$;
   replaces the counters associated to the removed transitions t_1 and t_2 that appear in all the pre-conditions and post-conditions defined in $\mathcal{T}_0$ with the new counter associated to the transition t_3.
2. *repeat the above step until all reducible pairs in $\mathcal{T}_0$ have been manipulated;*
3. *let $\mathcal{T}_0 := (\mathcal{T}_0 / \mathcal{T}_{remove}) \cup \mathcal{T}_{new}$, $\mathcal{C}_0 := (\mathcal{C}_0 / \mathcal{C}_{remove}) \cup \mathcal{C}_{new}$, let $\mathcal{T}_{remove}, \mathcal{C}_{remove}, \mathcal{T}_{new}, \mathcal{C}_{new} := \{\}$*

4. *if there are still reducible pairs in $\mathcal{T}_0$, go to the step 1 and repeat the above procedure; otherwise, let $\mathcal{T}' := \mathcal{T}_0, \mathcal{C}' := \mathcal{C}_0$,.*

The reduced automaton is $(\mathcal{S}, \mathcal{E}, \mathcal{T}', \mathcal{C}')$.

5. RELATED WORK

Several attempts have been made to apply formal verification methods to SystemC. Early work in this direction, notably by Drechsler and Große, focused on verifications at the gate level [2, 3] and were somehow limited to it. In recent years, several formal models are proposed aiming at verifying properties at the transaction level [8, 9, 10], but to the authors' knowledge, none of them stresses specifically the correctness at the level of delta-cycles, which is mainly the contribution of our model presented in this paper.

Among others, we have mentioned the work by Kroening et al. [8], pointing out that our symbolic composition is similar to their parallel composition but we do more. In fact, the model in [8] is based on the state/event analysis and it makes use of the formalization of labeled Kripke structures [1]. Having separated labels on states and transitions in the model provides a syntactic way of partitioning a SystemC model into a hardware and a software part. But states in their model include all possible intermediate states within a process, not just those waiting-states as in our model, and they also make a classification of processes as runnable processes, waiting processes, etc., which is basically the implementation idea of SystemC scheduler and is avoided in our model. On the other side, their labels on states allow effective manipulation of program data, which can be a nice mechanism to be introduced into our model so that more properties can be verified.

As for the generation of automata from SystemC models, two algorithms of generating finite state machines are proposed in [4]. These are generation algorithms for traditional model-checking adapted for SystemC and their model generates the states for the whole system from the very beginning, then the algorithms focus on solving the state exploration problem using the grouping technique. Instead, our approach has rather the component assembly nature, where predefined abstractions are done during composition. State exploration is not a big problem in our model, at least at the symbolic level, but these algorithms might be adapted in generating some extended waiting-state automata where more information required to represented in states for further verifications.

6. CONCLUSION

We propose in this paper a formal model based on traditional model-checking techniques. We define a new form of automata — SystemC waiting-state automata, based on the analysis of the waiting status of SystemC processes. The main goal of this model is to provide the capacity of verifying SystemC models at the level of delta-cycles, but not limited to it. Further extensions of the model are surely feasible for verifications at higher levels.

The procedure of the verification using SystemC waiting-state automata is clear: translate first SystemC processes into SystemC minimal-step waiting-state automata, eventually infer affine relations regarding transition counts of those small automata, combine these small automata together and reduce the resulted automaton to the final one which conforms to the SystemC scheduler, and pass the final automaton to the model-checking procedure. While the composition and the reduction are clearly defined in the paper, the compilation is remained unclarified. A possible approach of compiling SystemC code to waiting-state automata, is to integrate the waiting states into the control flow graphs of processes, with every arrow getting out of a state being labeled by a set of triggering events (e_{in} in SystemC waiting-state automata), and every arrow getting into a state being labeled by a set of effect events (e_{out} in SystemC waiting-state automata). Such an intermediate format can be translated into pure SystemC waiting-state automata by abstracting the control information using predicates and symbolic functions.

Future work include case studies of large examples besides the long-term compilation work. Also, the model itself demands probably further refinement so as to fit well in real cases, and proper extensions might be necessary for verifying particular properties.

REFERENCES

[1] Sagar Chaki, Edmund M. Clarke, Joël Ouaknine, Natasha Sharygina, and Nishant Sinha. State/event-based software model checking. In Eerke A. Boiten, John Derrick, and Graeme Smith, editors, *Proceedings of the 4th International Conference on Integrated Formal Methods (IFM '04)*, volume 2999 of *Lecture Notes in Computer Science*, pages 128–147. Springer-Verlag, April 2004.

[2] Rolf Drechsler and Daniel Große. Reachability analysis for formal verification of systemc. In *Euromicro Symposium on Digital Systems Design (DSD 2002)*, pages 337–340. IEEE Computer Society, 2002.

[3] Daniel Große and Rolf Drechsler. Formal verification of LTL formulas for systemc designs. In *IEEE International Symposium on Circuits and Systems (ISCAS 2003)*, volume 5, pages 245–248, 2003.

[4] Ali Habibi, Haja Moinudeen, and Sofiène Tahar. Generating finite state machines from systemc. In Georges G. E. Gielen, editor, *DATE Designers' Forum*, pages 76–81. European Design and Automation Association, Leuven, Belgium, 2006.

[5] Open SystemC Initiative. Systemc v2.0 functional specification, 2002.

[6] Open SystemC Initiative. Systemc v2.1 language reference manual, 2005.

[7] B. Jeannet, N. Halbwachs, and P. Raymond. Dynamic partitioning in analyses of numerical properties. In *Proceedings of the Sixth International Symposium on Static Analysis (SAS'99)*, 1999.

[8] D. Kroening and N. Sharygina. Formal verification of systemc by automatic hardware/software partitioning. In *Proceedings of the Third ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2005. MEMOCODE '05*, pages 101–110, 2005.

[9] L. Maillet-Contoz M. Moy, F. Maraninchi. Lussy: a toolbox for the analysis of systems-on-a-chip at the transactional level. In *Proceedings of Fifth International Conference on Application of Concurrency to System Design (ACSD 2005).*, 2005.

[10] K.L. Man, Andrea Fedeli, Michele Mercaldi, and M.P. Schellekens. Systemc^{fl}: An infrastructure for a tlm formal verification proposal (with an overview on a tool set for practical formal verification of systemc descriptions). In *Proceedings of the IEEE East-West Design & Test Workshop EWDTW*, 2006.

[11] N. Mercouroff. An algorithm for analyzing communicating processes. In *Mathematical Foundations of Programming Semantics, 7th International Conference, Pittsburgh, PA, USA, March 25-28, 1991, Proceedings*, volume 598 of *Lecture Notes in Computer Science*. Springer, 1992.

[12] A. Venet. Automatic determination of communication topologies in mobile systems. In *Proceedings of the Fifth International Symposium on Static Analysis (SAS'98)*, 1998.

[13] Arnaud Venet. Abstract interpretation of the pi-calculus. In *Analysis and Verification of Multiple-Agent Languages, 5th LOMAPS Workshop, Stockholm, Sweden.*, volume 1192 of *Lecture Notes in Computer Science*. Springer, 1997.

[14] Yu Zhang, Franck Védrine, and Bruno Monsuez. Systemc waiting-state automata, 2007. http://www-sop.inria.fr/everest/Yu.Zhang/docs/ZVM-systemc-long.pdf.