Compositional Reactive Semantics of SystemC and Verification with RuleBase

Rudrapatna K. Shyamasundar, Frederic Doucet, Rajesh K. Gupta, and Ingolf H. Krüger

Abstract We present a behavioral semantics of SystemC that succinctly captures its reactive features, clock and time references, macro- and micro-time model, and allows the specification of a network of synchronous and asynchronous components communicating through either high-level transactions or low-level signal and event communications. The proposed semantic framework demonstrates the anomalies introduced by the simulation kernel, in spite of the macro- and micro-time scales. The framework further relates the simulation and logical correctness and provides a technique for scaling up the verification while keeping the correctness intact. Furthermore, we translate SystemC components to RuleBase using our semantic characterization that permits testing and verification of heterogenous designs. We illustrate the verification of a Central Locking System (CLS) designed in SystemC.

Keywords: SystemC, semantics, verification, model checking

1 Introduction

System-level modeling using SystemC facilitates the use of various features and concepts such as perfect synchrony, asynchrony, reactive, and time specifications through a C++ class library. SystemC provides a bridge between hardware and software design and thus, provides a unifying framework for hardware/software design. SystemC consists of C++ libraries and a simulation kernel for creating behavioral and register-transfer level (RTL) designs. It provides a common development environment needed to support software engineers working in C/C++, and hardware engineers working in HDLs such as VHDL, Verilog, etc., particularly system-on-a-chip designs. While the simulation behavior of a SystemC description is well understood by engineers, existing frameworks have not catered to a comparative evaluation of simulation and verification particularly to the various perfect-synchrony features as well as macro- and micro-time scales. We also show that the two time scales, while intended to avoid some of the anomalous behaviors possible in perfectly synchronous languages like Esterel, cannot indeed be avoided. In fact, it also throws open the question whether the δ-cycle can indeed be avoided to speed up simulation without foregoing correctness.

S. Ramesh and P. Sampath (eds.), *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, 227–243.

© Springer 2007

In this paper, we describe a compositional semantics using the rewrite framework of Esterel. A sound semantic model provides the ability to reason regarding issues with composition of SystemC models without adding global restrictions other than those imposed by SystemC language itself. With the ever increasing of complexity of systems, it is important to exploit the notion of compositionality that is deeply embedded in the rationale of SystemC. A clean separation of process reaction, and computing the next environment provides a basis for simulation and verification without flattening the composed model into a single uniform model. The main contributions of the paper are:

1. a semantic foundation that captures (i) the synchronous and asynchronous process composition, (ii) all levels of abstractions for communications, and (iii) relation between simulation correctness and logical correctness;
2. a way to scale up the verification while ensuring the simulation and logical correctness and equivalence are intact; and
3. an automatic translation to the model checkers for verification of SystemC components, providing a powerful workbench for testing and verification.

2 Overview of SystemC

SystemC is essentially a C++ library that provides macros to model hardware and software systems. The difference between a system-level modeling language rooted in C++ such as SystemC, and C++ itself, is that the system-level modeling macros are used to model a system, but not to implement it. Figure 1 shows an abstract syntax for SystemC, where we consider only the statements specific to modeling with SystemC and not the general statements in the C++ language. A SystemC program is a set of interconnected modules communicating through channels, signals, events, and shared variables. A module is composed of a set of ports, variables, a process and a set of methods. A process is sensitive to a set of events, and optionally can have

```
program       := { modules, channels, signals, events, variables }
module        := { ports, variables, process-decl, process-body, methods }
process-decl  := <process name> <sensitivity> <reset-condition>
process-body  := <event-comm | signal-comm | chan-comm | control-flow |
                          arithmetic>*
event-comm    := wait(event) | wait(event,time) |
                 wait(time) | wait(event_list)
                 notify(event) | notify-delayed(event) |
signal-comm   := signal.read | signal.write |
chan-comm     := tlm_port->put(value) | tlm_port->get(var) |
                 tlm_port->method(parameters)
control-flow  := <C++ control flow>
arithmetic    := <C++ arithmetic>
```

Fig. 1 Simplified abstract syntax for SystemC.

a reset condition. Some of the distinctive characteristics are informally summarized below:

- A process is in the ready state when either the SystemC program starts or there is an event that the process is waiting for. A process is in the waiting state when it is waiting for an event. A process is unblocked when it is notified by an event. Events can be notified immediately, or the notification can be delayed until all processes are waiting.
- Time is modeled through macro-time in some pre-defined quantifiable unit; a process waits for a given amount of time, expiration of which is notified through an event.
- Between processes, the basic communication is by reading and writing signals. During the execution of a SystemC program, all signal values are stable until all processes reach the waiting state. When all processes are waiting, signals are updated with the new values.
- Transaction-level communications are through channels, which are accessed using an interface defined by a set of methods. The transaction-level model (TLM) interface can be put and get methods, to connect to channels like FIFO buffers, etc., or a custom set of methods to connect to specific channels. In the body of the methods, communication is done by using the shared variables, events, signals or other channels defined in the channel.

When executing a SystemC program, the illusion of concurrency is provided to the user by a simulation kernel implementing a discrete event simulation loop. The simulation loop divides time into macro-time and micro-time, where micro-time is used for creating a partial ordering of the events that can occur in a macro-time unit. Micro-time events are called delta events, processed into the simulation queue, advancing micro-time as needed without advancing macro-time (to simulate synchronous reactions). Micro-time events are not observable in a macro-time scale because they are used only to simulate a synchronous concurrent reaction on a sequential computer.

The discrete-event simulation loop is used to compute the next environment which contains the events to which the processes synchronize. The simulation kernel first synchronizes all the processes with immediate events, and then picks one process to react. This reaction loop is repeated until there are no more immediate events and all processes are waiting. Then, the processes are synchronized with micro-time events, followed by a new reaction loop. When there are no more immediate nor micro-time events, the processes are synchronized with the macro-time events, leading to another reaction loop.

Note that simulation cannot guarantee correctness due to:

1. the inability of simulation to produce all possible behaviors, and
2. the simulation loop can introduce anomalous behaviors that cannot happen logically. For example, due to the underlying scheduler (as we discuss in Section 4), the simulation kernel can introduce nondeterminism and causality cycles in a design description.

3 Semantic Framework

We now define the behavioral semantics of SystemC compositionally to capture all possible behaviors that can be computed by a SystemC program by composing the semantics of its components. First, we divide the observables as controllable variables and environment variables. For a given SystemC module, the controllable variables are output signals, internal variables, output channels, output events, and the program counter for the process. The environment variables are input signals, input events, input channels, and global variables.

At any point during the program, there is at most one process that is reacting to the environment. One can locally visualize instants during which reactions occur by observing the state (C++ variables and program counters for each processes) of the program, denoted σ, or the modeling environment (events, channels, signals, processes, etc.), denoted E. An environment only lasts an instant; i.e., it is not persistent like the state and an event occurs only right after the instant it is emitted. For describing, how a statement changes the configurations of the observables, we use rewrite rules of the form $(\langle stmt \rangle, \sigma) \xrightarrow[E]{E_O, b} (\langle stmt' \rangle, \sigma')$ where:

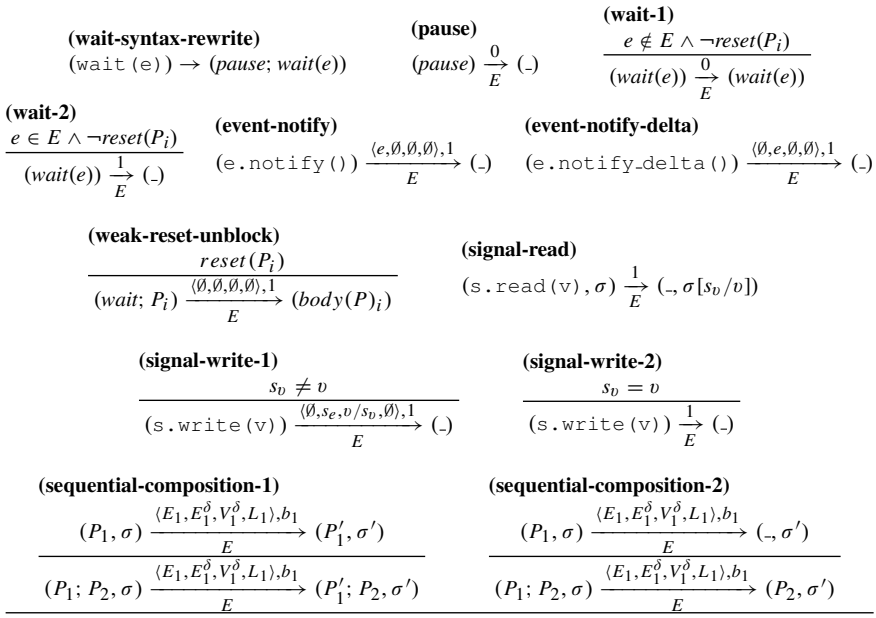
- $stmt$ is a SystemC program text with the location of the program counter, before the reaction, and $stmt'$ is the program text with the location of the program counter after the transition,
- σ and σ' are the states before and after the reaction respectively,
- E is the environment while taking the transition, E_O is the output emitted during the transition; in general, an environment is a 4-tuple $E = \langle E^I, E^\delta, V^\delta, L \rangle$ where:
  - E^I is the set of immediate events,
  - E^δ is the set of next delta events,
  - V^δ is the set of next delta updates for variable,
  - L is a set of pending transactions or pending asynchronous tasks,
- b is a Boolean flag indicating if the process completed in the instant or not.

Keeping in view the simulation explained above, a SystemC model behaves in an alternating sequence of synchronizations and reactions $(\rightarrow_{sync} \rightarrow_{react})^*$), observable as a sequence of environments and states $(E_0\sigma_0)(E_1\sigma_1)(E_2\sigma_2)\ldots$.

3.1 Reactive Statements

The reactive semantics forms the crux of the simulation. Some of the rules are given Table 1. The wait-syntactic rule defines that a `wait` statement is syntactically reduced to the sequence of a *pause* statement followed by a semantic *wait* statement. The wait argument `e` is passed verbatim through the reduction. A *pause* statement pauses a reaction until the next environment – it does not terminate in the current instant and reduces to nothing. The behavior of the semantic *wait* statement is to wait

Table 1 Semantics of reactive statements.



for an event e to be in the environment. Rule wait-1 defines that when event e is not in the environment and the reset condition specific to the current process (defined in by variable P_i) is false, the process continues to wait without doing anything. In rule wait-2, when event e is present in the environment and the reset is not asserted, the wait statement terminates and reduces to nothing.

The event notification statement immediately emits an event e in the next environment, and terminates. The delayed notification statement emits event e to be in the next delta environment. The processes waiting on these events will unblock in either the synchronization with the next environment and the synchronization with the next delta environment respectively. The weak-reset-unblock rule shows that when a process is waiting for some event and the reset variable is asserted, the process resets to the initial value for its program counter.

Now let us look at statements concerned with signal communications. A SystemC signal s is persistent and is associated with a variable s_v which holds the data value, and to an event s_e to notify signal value transitions. For a signal write operation, if the value v being written to a signal is different than the current value, the statement terminates, reduces to nothing, and emits event s_e in E^δ and v/s_v in V^δ. Otherwise the statement terminates without doing anything. Finally, there are two cases for sequential composition. If statement 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.

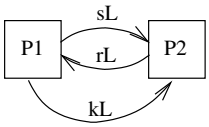


Fig. 2 Asynchrony interface.

Table 2 Semantics of timed statements.

(wait-timeout)

$$(\texttt{wait}(t)) \xrightarrow[E]{\langle sL_{P_i}(t),\emptyset,\emptyset,L_{P_i}\rangle,0} (wait(rL_{P_i}))$$

(wait-event-timeout)

$$(\texttt{wait}(t,e)) \xrightarrow[E]{\langle sL_{P_i}(t),\emptyset,\emptyset,L_{P_i}\rangle,0} (wait(rL_{P_i}|e))$$

(wait-timeout-event-1)

$$\frac{(rL_{P_i} \in E) \wedge \neg reset(P_i)}{(wait(rL_{P_i}|e)) \xrightarrow[E]{1} (\_)}$$

(wait-timeout-event-2)

$$\frac{(rL_{P_i} \notin E) \wedge (e \in E) \wedge \neg reset(P_i)}{(wait(rL_{P_i}|e)) \xrightarrow[E]{\langle kL_{P_i},\emptyset,\emptyset,\emptyset\rangle,1} (\_)}$$

3.2 Statements for Time

Wait-timeout statements are used to request a notification at a later macro-time. For this purpose, we shall assume the presence of an asynchronous timer process in the environment as in CRP [2], that can be called from any process in need of setting an alarm.

Figure 2 shows the asynchronous gateway interface. The timer is an asynchronous task, which is started and controlled, indirectly, by the process. The timer has the general interface of the asynchronous task which is described as follows. The asynchronous task is started with an event sL, and the completion of the asynchronous task is notified with event rL. Event kL is used to kill an asynchronous task. The set L contains the labels of all the currently active asynchronous tasks.

The semantic functions for the timed statements are given in Table 2. The wait-timeout statement requests an alarm after t units of time by sending t on signal sL_{P_i} to a timer L_{P_i} in the environment, with, P_i being the label for the current process. The process then proceeds to wait for an event rL_{P_i} which is to be sent by the timer after t time units. A process can also wait for an event e, with a timeout t, as showed in rule wait-event-timeout. If event e occurs before the time out (before receiving event rL_{P_i}), the process will resume and kill the pending timer by notifying event kL_{P_i}. It is necessary to kill the pending timer so that, after time t the process will not receive any unnecessary timeout event.

3.3 Rules for Parallel Composition

In SystemC, the parallel composition of the processes is defined as each module is instantiated. All modules are to be executed concurrently once the simulation is

Table 3 Semantics of parallel composition.

(sync-imm)

$$\frac{\forall i \in \{1..l\} : \exists e \in E^I : waiting(P_i, e) \qquad \forall j \in \{l+1..m\} : \forall e \in E^I : \neg waiting(P_j, e)}{(P_1 \| ... \| P_l \| P_{l+1} \| ... \| P_m) \xrightarrow[\langle E^I, E^\delta, V^\delta, L\rangle]{\langle \emptyset, E^\delta, V^\delta, L\rangle, 1}_I (P_1' \| ... \| P_l' \| P_{l+1} \| ... \| P_m)}$$

(async-react)

$$\frac{\begin{array}{c}\forall i \in \{1...l\} : waiting(P_i) \qquad \forall j \in \{l+1..m\} : ready(P_j) \\ select\ x \in \{l+1..m\} : (P_x, \sigma) \xrightarrow[E]{\langle E_x, E_x^\delta, V_x^\delta, L_x\rangle, 0} (P_x', \sigma') \\ merge(\langle E_x^\delta, E^\delta\rangle, \langle V_x^\delta, V^\delta\rangle, 1)\end{array}}{(P_1 \| ... \| P_l \| P_{l+1} \| ... \| P_m) \xrightarrow[\langle E^I, E^\delta, V^\delta, L\rangle]{\langle E_x, E_x^\delta \cup E^\delta, V_x^\delta \cup V^\delta, L_s \cup L\rangle, 1} (P_1' \| ... \| P_l' \| P_{l+1} \| ... \| P_x' \| ... \| P_m)}$$

(sync-micro)

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

(sync-macro)

$$\frac{\forall i \in \{1..n\} : waiting(P_i) \qquad e_t = next\_time()}{(P_1 \| ... \| P_n, \sigma) \xrightarrow[\langle \emptyset, \emptyset, \emptyset, L\rangle]{\langle e_t, \emptyset, \emptyset, L\rangle, 1}_T (P_1 \| ... \| P_n, \sigma[V^\delta / V])}$$

started. The various booking operations for building the environment can be partitioned as:

1. synchronizing processes with the events in the environment (denoted $\rightarrow_I$),
2. reaction of the selected process (denoted $\rightarrow$),
3. building next micro-environment (denoted $\rightarrow_\delta$), and
4. building next macro-environment (denoted $\rightarrow_T$).

The complete simulation loop can then be captured as iterative composition of relations given by: $((\rightarrow_I \rightarrow)^* \rightarrow_\delta)^* \rightarrow_T)^*$.

The various semantic rules of composition are given in Table 3. Rule sync-imm is defined to unblock all processes that are waiting for events that are in the environment. We use the notation $waiting(P, e)$ to mean that P is waiting on event e, meaning P is of the form $wait; P'$. In other words, it is a synchronous composition, but only for the wait statements.

Rule asynch-react defines the reactivity. A process which is ready, is selected to run until it reaches the next pause. The merge predicate provides a check on whether or not to allow nondeterministic environments in the composition. Nondeterministic environment are possible when two different values can be written to a signal in the same reaction. The $merge$ predicate checks the feasibility of partially ordering the events in the delta cycle. Setting the third parameter to "1" indicates that the partial

order has to be consistent. One can allow nondeterministic environment by setting the third parameter to -1.

Rule sync-micro defines the synchronization on delta events to build the next micro-environment. 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 variables.

The rule for the synchronization on timed events builds the next environment from time events and advance macro-time. It is effective when all processes are blocked, where there are no immediate event nor delta event. Timed events are posted by `wait(time)` statements, timers and clocks. For simplicity in this rules, we use $next\_time()$ to broadly indicate moving to the next time.

3.4 Statements for Transaction-Level Modeling

For transaction-level method calls, we simply inline the body of the method inside the caller module. The put/get transaction-level channels from the SystemC TLM library, when used as a single place buffer in a point-to-point connection, do not cause nondeterministic behaviors. This is because the state changes in the transaction-level buffers are visible immediately for the calling process, but only at the next delta cycle for the other processes. This behavior is useful to avoid the kind of nondeterministic behavior described in Section 4. Furthermore, using these channels in combination with the step scheduler (described in Section 5) enables efficient transaction-level verification.

Table 4 lists the semantic rules for the TLM statements. We consider only the rules for communication with single place TLM FIFO buffers. Rule tlm-put writes data on the buffer if the buffer is empty. Otherwise, it waits that the data already on the buffer is read. Rule tlm-get works similarly in the complementary fashion. Note that these rules are to be used only with the step scheduler as their generalization for the full scheduler would significantly complicate the semantics and require extra copies of the variables in the verification environment (see Section 5).

Table 4 Semantics of transaction-level statements.

(tlm-put-1)

$$\frac{\neg full(f)}{(\texttt{f->put(v)}, \sigma) \xrightarrow[E]{\langle \emptyset, f_w, \emptyset, L\rangle, 1} (\_, \sigma[v/put(f)])}$$

(tlm-put-2)

$$\frac{full(f)}{(\texttt{f->put(v)}) \xrightarrow[E]{0} (wait(f_r); \texttt{f->put(v)})}$$

(tlm-get-1)

$$\frac{\neg empty(f)}{(\texttt{f->get(v)}, \sigma) \xrightarrow[E]{\langle \emptyset, f_r, \emptyset, L\rangle, 1} (\_, \sigma[get(f)/v])}$$

(tlm-get-2)

$$\frac{empty(f)}{(\texttt{f->get(v)}) \xrightarrow[E]{0} (wait(f_w); \texttt{f->get(v)})}$$

3.5 Computing the Semantics of SystemC Components

We now show how we generate the transition system for a SystemC component by applying the semantic rules. For a process P, the transformation yields the reactive sequences from initial state σ_0 to state σ_n such as:

$$(stmt, \sigma_0) \xrightarrow[E]{E_0,1} \ldots \xrightarrow[E]{E_{n-1},0} (stmt_n, \sigma_n)$$

where from a given state, the process will react until the next wait statement (or up until termination). During the reaction, the states in the sequence between σ_0 and σ_n are observable only from within P, and no other process in the environment can observe the intermediate states. Hence, from another SystemC process, only the first and last states of the reaction are observable:

$$(stmt, \sigma_0) \xrightarrow[E]{E_O,0} (stmt_n, \sigma_n)$$

Therefore, when building the transition system for a process, we can reduce a detailed graph to an observable graph with all the intermediate transitions, form an initial state σ_0 to a state σ_n, collapsed to one single transition.

From the process description, we use the semantic rules to construct the control-flow graph, and add the predicate for the conditions and assignments. Since we do not keep the state implicitly in the semantic structure, we use the weakest precondition on observable paths to convert the control-flow graph to a graph with only the observable reactive steps. We use the standard definitions for constructively computing weakest precondition. For all the pair of observable points (paths from a pause statement to the next pause statement in the graph) we compute the weakest precondition between the points, and add it to the observable graph if the weakest precondition is satisfiable. Note that one cannot constructively compute the weakest precondition for loops that cannot be unrolled. This is a well known limitation of the approach, but we can alleviate it by requiring wait statements inside loop bodies.

Figure 3 shows an example of the semantic translation of a SystemC process. The process is first converted to the control-flow graph, and then to the observable graph. Every transition is labeled with a guard (the conjunction of the labels starting with a G), and the variable assignments should the transition be taken. The process initializes a counter variable to 0, waits on the clock, re-initialize it to 10, and then counts up until it is reset.

4 Anomalous Behaviors

We illustrate various anomalous behaviors such as causality (as in Esterel), non-determinism, etc.

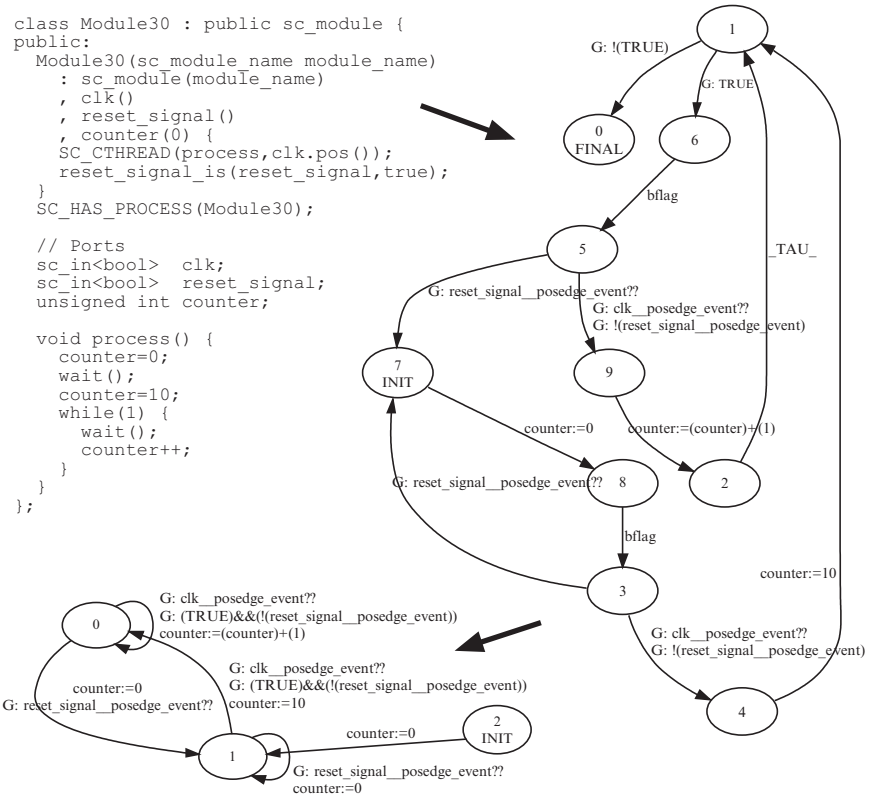


Fig. 3 Example of semantic translation.

4.1 Causality Cycle

A causality cycle is a behavior that triggers an infinite amount of action in a finite amount of time. It is important to look for causality cycles in a SystemC design as causality cycles are not always triggered in simulation. Indeed, a causality cycle can be triggered by a corner case condition in the behavior of the composition of a system of asynchronous components. In a simulation, one can observe a causality cycle when a computation does not stabilize to specific output values in an instant and keeps re-triggering itself. In our semantics rules, a causality cycle occurs when it never gets to the next synchronization with the macro-time events. Indeed, the causal cycle occurs in the reaction $(((\rightarrow_I \rightarrow)^* \rightarrow_\delta)^* \rightarrow_T)^*$ when step $\rightarrow_T$ is never taken.

We now give an example of a SystemC program that has a causality cycle. The component definition for a "watching process" is given in Figure 4(a). The behavior of the watching process is to watch another module, and if the other module has not been triggered, the watching process will trigger it. The component by itself does not

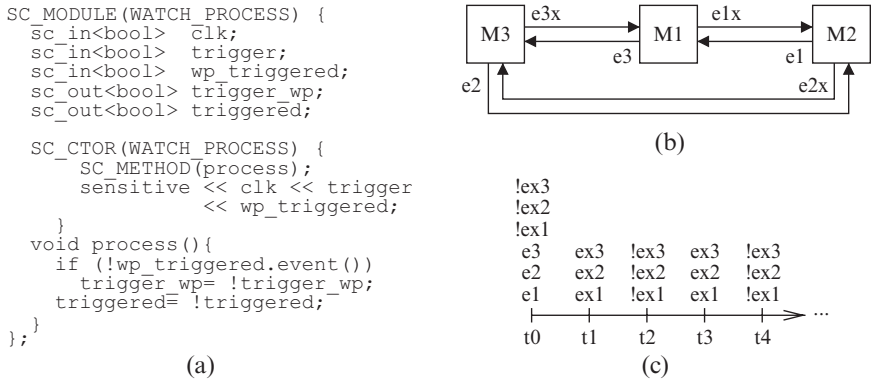


Fig. 4 Example of causality cycle: (a) watching component (b) three such components watching each others, and (c) depiction of the causality cycle.

have a causality cycle. However, if one builds a system where instances of watching processes in are connected in a cycle such as depicted in Figure 4(b), there will be a causality cycle. When executing the program, the system will enter a causality cycle as depicted in Figure 4(c). The reason is that the modules will keep triggering each other.

Note that causal loop can sometimes be desirable when used to produce an oscillating behavior (i.e., generate a clock). For instance, a SystemC untimed model never takes the $\rightarrow_T$ step by construction. We find a causality loop by searching for a loop in the state graph where the next time is never reached. We use the model checker to verify that, for a given design, it is always possible to reach the next clock – thus no divergence.

4.2 Nondeterminism

None of the SystemC syntactic constructs are meant to explicitly produce nondeterministic behavior, in the sense of a nondeterministic choice operator. A program is nondeterministic if, for a given input, it is possible to observe multiple different output behaviors. Causes of nondeterministic behavior in SystemC can be communications with shared variables, immediate event notification, or using uninitialized signals and variables. Nondeterministic behavior may not be observable in the SystemC simulation of a program because the output behavior is decided by the implicit process selection priorities in the scheduler.

We now give an example of a nondeterministic SystemC program, depicted in Figure 5. The program is composed of two modules which communicate through an event e. The first module notifies the second module, and the second module terminates the simulation upon reception of the event notification. To effectively receive

```
sc_event e;

SC_MODULE(M1) {
  SC_CTOR(M1) {
    SC_THREAD(a);
  }
  void p1() {
    e.notify();
  }
};
```

```
SC_MODULE(M2) {
  SC_CTOR(M2) {
    SC_THREAD(b);
  }
  void p2() {
    wait(e);
    sc_stop();
  }
};
```

```
int sc_main(int argc,
  char* argv[]) {
  M1 m1("m1");
  M2 m2("m2");
  //  M2 m2("m2");
  //M1 m1("m1");

  sc_start(10);
  return 1;
};
```

Fig. 5 SystemC code whose behavior is dependant on the scheduler.

```
SC_MODULE(Buffer) {
  bool data_avail;
  int  data_value;

  SC_CTOR(Buffer) {
    data_avail= false;
  }

  void read(int& val) {
    while(!data_avail)
       wait();
    val= data_value;
    data_avail=false;
  }

  void write(int val) {
    while(data_avail)
      wait();
    data_value= val;
    data_avail=true;
  }

  bool peek_data(int& val) {
    if (data_avail) {
      val= data_value;
      return true;
    } else
      return false;
  }
};
```

Fig. 6 Definition of a buffer which can cause nondeterministic behavior.

the event notification, process p2 must wait on the event before the event is notified. If the notification is done before p2 runs, then the event notification will be lost. The behavior of this program is dependent on the scheduler because the scheduler will decide which process will run first. If process p1 is run before process p2, then event *e* will be missed by p2. With the reference implementation of the SystemC simulation kernel, the initial process triggering schedule is determined by the order into which the modules are instantiated. If we permute the order of instantiation (as commented), then the problem is not observable as process p2 will be waiting for the event, and will terminate the simulation. An example of a nondeterministic TLM buffer using shared variables is also illustrated in Figure 6.

5 Verification Framework

For a given SystemC program, we automatically build an SMV description that allows verification of desired properties. We have built the prototype for both IBM RuleBase and NuSMV. We do not build a global transition system explicitly, but rather we only translate the modules and use the model checker to compose the environment model with the transition systems of each module (process). We automate

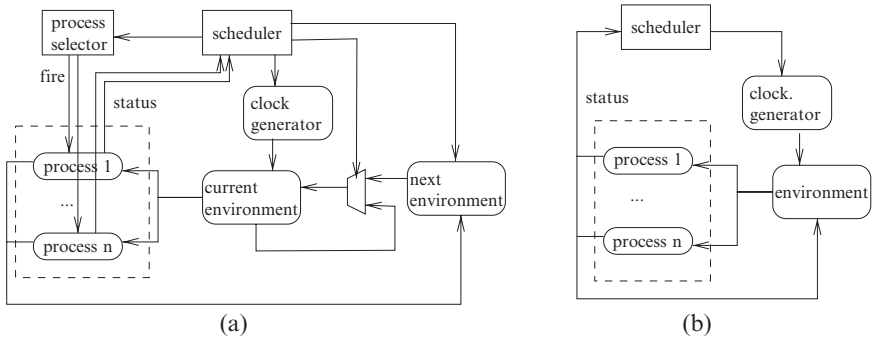


Fig. 7 Example of verification setup: (a) full scheduler and (b) step scheduler.

the composition rules by building an environment that manages the events and fires the processes. For this purpose, it is possible to use two different schedulers:

1. Full scheduler: fires the process one by one and compute all the possible interleavings and synchronizations.
2. Step scheduler: fire all processes synchronously and synchronize them at the same time on the delta events (no immediate events).

Figure 7 depicts the difference between the schedulers. The full scheduler captures the effect of nondeterminism in shared variable communications, while the step scheduler does not. But the good thing is that the verification algorithms are significantly more efficient with the step scheduler because there is no process selection and it does not require duplicating the variables in the environment. However, using the step scheduler introduces the following restrictions on the design. First, nondeterminism, immediate notifications are not allowed. Second, there can be no interference on shared variable communication: this means that, at every cycle, only one process can write to a shared variable.

It happens that a large family of SystemC program satisfy these conditions. This is because these restrictions match the traditional synchronous semantics, to which all RTL models and many TLM models are built. Since the size of the state space is exponential to the number of variables this has the potential for significant performance improvements.

6 An Example: Central Locking System

We illustrate our approach by verifying the transaction-level model of a the design of a design of a Central Locking System (CLS), a system used to lock and unlock the doors, and validate who can perform these operation and when. We consider it to be an interesting example since it is distributed over many components in the car, and

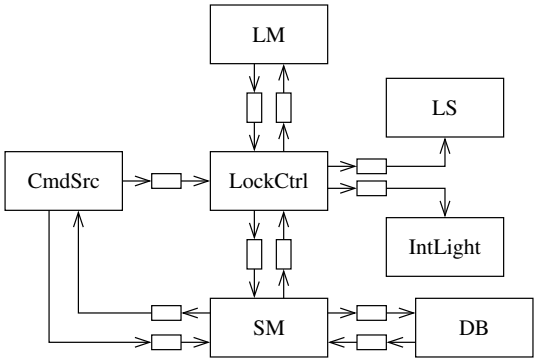


Fig. 8 Structural specification for the Central Locking System.

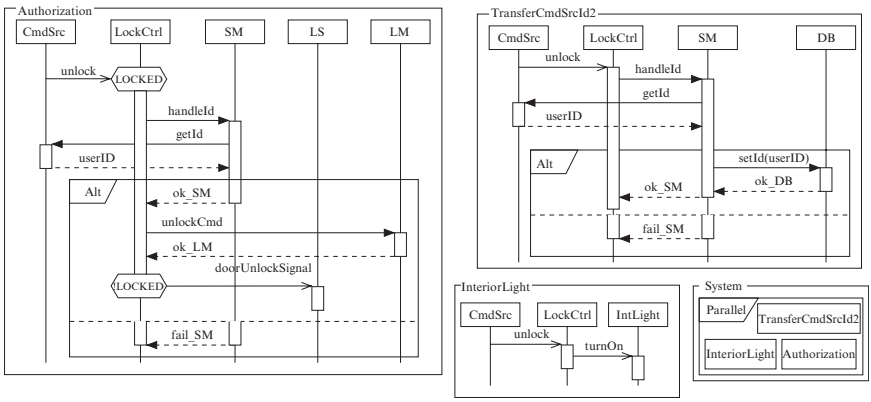


Fig. 9 Behavioral specification for the Central Locking System.

one has to be careful to avoid synchronization or deadlock problems. The structural specification of the CLS is illustrated in Figure 8 and the behavioral specification of the CLS is depicted in Figure 9. The high-level system specification describes the system as being the parallel composition of the three basic services illustrated in the interaction diagrams.

We implemented the CLS with the SystemC language and we replicated the local structure of the system into the component structure. Table 5 shows the verification statistics for the verification of the SystemC implementation of the CLS against the interaction specifications. The data domain for the keys has been reduced from undefined integer range from -1 to 9, which corresponds to the values transmitted between the modules.

Table 5 Verification of SystemC CLS with the services.

	NuSMV 2.4.1		RuleBase	
Property	*Time (s)*	*Memory Used*	*Time (s)*	*Memory Used*
No causal loops	2200.234	47172 K	632	n/a
All services	2031.379	46880 K	218	743 MB
TransferKeyId2 service	418.482	19832K	136	480 MB
IntLight service	185.940	18064 K	178	288 MB
Authorization service	500.835	19456 K	142	447 MB

The verification runs are much faster using RuleBase but require more memory. The first entry is for the verification of the conjunction of all services together, while the next three entries are for the verification of the individual services. All the runs are with the cone of influence reduction and with dynamic BDD variable re-ordering. Using NuSMV, we always verify that the FSM has fair paths and has no deadlocks, increasing the verification times.

7 Related Work

Our approach incorporates the reactive synchronous features of SystemC succinctly and distinguishes from other works and further, it is based on frameworks proposed in [1], [2], and [8].

Recently, there has been two interesting approaches based on synchronous languages. The first one, advocated in [6], is based on deriving the transactional-level models of SystemC into a automata encoded in Lustre, and refining the channel interfaces through a protocol automata. Subsequently, it uses the Lustre model checking tools for analysis. While the approach is nice, it carries over the full delta-cycle details which compromises the scalability of the verification. We improve on their results by defining the TLM abstraction by restricting the design and mapping the semantics to a synchronous transition system ala Esterel, which enables us to improve on the scalability. The approach in [9] describes how to translate the body of a SystemC process into a set of Signal equations, and synchronously compose all the equations into the transition system. The problem with this work is that it ignores the simulation anomalies, timing statements and the TLM statements. The work in [5] uses a process algebra but ignores the need of distinctions between synchronous and asynchronous compositions, ignores δ cycles, etc.

Approaches such as [3, 7] translate SystemC program simulation into ASML units of compiled code. While it appeals as a nice global executional simulation system, the reactive features of the SystemC such as reset (that has priority as well), broadcast to waiting processes, cannot be succinctly captured in synchronous fashion like an Esterel transition system. Indeed, by going to the ASML, one does not have access to the nice results of the synchronous community and it is also unclear how one can

exploit the power of modern model checkers. Our translation is similar to the work of [4] that similarly generates SMV transition relations for SystemC code. Our work fully uses the reactive rules and with the addition of TLM abstractions and uses the step semantics.

8 Summary and Conclusions

In this paper, we have presented a compositional reactive semantics for SystemC that captures both at signal and TLM levels of abstractions. We are able to reduce the design without anomalies to pure synchronous transition systems to exploit the full power of the SMV-based verifiers. Further, the framework provides techniques to detect anomalies and enable relating simulation to logical correctness. It also has brought to light how verification can be speeded up without foregoing correctness. Another interesting question that has been brought to light is the need to arrive at quantitative/qualitative comparisons of δ-cycles of SystemC with respect to the constructive semantics of Esterel. Our system translates a given SystemC program into RuleBase – industrial strength verifier. This allows a variety of design integrations and exploits the power of industrial strength model checkers. Further, our results improve on previous published results and we are now able to verify TLM within some reasonable (scalable) times, while enjoying all the capabilities of the modern model checkers. The work needs to enriched to take into account dynamic memory allocations, exceptions, custom channels, etc. Furthermore, the computation of the weakest precondition can be inefficient and yield unnecessary overhead for medium and large processes.

The focus for our future work is to generalize the methodologies to support custom channels, develop compositional design methodologies, and provide appealing abstractions for application-level transactions and compositions of transactions.

Acknowledgements Thanks go to Mr. Saurabh Joshi, IBM India Research Lab, for various experiments with the system developed for RuleBase.

References

1. G. Berry. *The Foundations of Esterel*. MIT Press, 2000.
2. G. Berry, S. Ramesh, and R.K. Shyamasundar. Communicating Reactive Processes. In *Proc. Symp. on Principles of Programming Languages*, 1993.
3. A. Habibi and S. Tahar. Design and Verification of SystemC Transaction-level Models. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 14(1):57–68, January 2006.
4. D. Kroening and N. Sharygina. Formal Verification of SystemC by Automatic Hardware/ Software Partitioning. In *Proc. of Formal Methods and Models for Codesign*, 2005.
5. K.L. Man. SystemC^{FL}: Formalization of SystemC. In *Proc. of 12th IEEE Mediterranean Electrotechnical Conference*, 2004.

6. M. Moy, F. Maraninchi, and L. Maillet-Contoz. LusSy: A Toolbox for the Analysis of Systems-on-a-Chip at the Transactional Level. In *Proc. of Application of Concurrency to System Design*, 2005.
7. W. Mueller, J. Ruf, D. Hofmann, J. Gerlach, T. Kropf, and W. Rosenstiel. The Simulation Semantics of SystemC. In *Proc. Design Automation and Test in Europe Conf.*, 2001.
8. B. Rajan and R.K. Shyamasundar. Multiclock Esterel: A Reactive Framework for Asynchronous Design. In *Proc. of 13th Intl. Conf. on VLSI Design*, 2000.
9. J.-P. Talpin, D. Berner, P. Le Guernic, A. Gamatie, S. Shukla, and R. Gupta. A Behavioural Type Inference System for Compositional System-on-Chip Design. In *Proc. of Application of Concurrency to System Design*, 2004.