Formal Verification of SystemC Designs Using a Petri-Net Based Representation

Daniel Karlsson, Petru Eles, Zebo Peng

Department of Computer and Information Science, Linköpings universitet, Sweden
{danka, petel, zebpe}@ida.liu.se

Abstract

This paper presents an effective approach to formally verify SystemC designs. The approach translates SystemC models into a Petri-Net based representation. The Petri-net model is then used for model checking of properties expressed in a timed temporal logic. The approach is particularly suitable for, but not restricted to, models at a high level of abstraction, such as transaction-level. The efficiency of the approach is illustrated by experiments.

1. Introduction

Embedded electronic devices are often highly safety critical, such as in automotive and avionics applications or medical equipment. It is both very error-prone and time-consuming to design these complex systems. At the same time, there is a strong economical incentive to decrease the time-to-market.

SystemC [1] has been developed to target these issues. Using SystemC, developers can easily create a working model of the system at a functional level. More details can then be added in order to refine the model. Each level of refinement can inherently be simulated.

Designing such complex systems, as introduced previously, is a very error-prone process disregarding the methodology and design languages used. With SystemC, typically, simulation is used to trap this kind of design errors. Despite the efficient implementation of the simulator, it is mostly not feasible to find all corner cases necessary to trap all errors. Particularly in critical parts of the design, this is not acceptable. Therefore, there is a need to resort to formal methods.

Very few attempts have been made to apply formal verification methods on SystemC. Most verification methods which do exist are based on simulation [2], [3]. However, Drechsler and Große [4], [5] can prove, using bounded model checking, that the SystemC model satisfies a given property. There are, however, restrictions and limitations in their approach. The models have to be at RTL and cycle-accurate. As a consequence, they can, in particular, not verify models using SystemC channels, necessary for transaction-level modelling (TLM). Nor can they handle continuous time, rather than clock cycles.

Kroening and Sharygina [6] translate SystemC models into labelled Kripke structures (LKS). However, their approach does not take either timing or signal aspects into account. Their work is furthermore more focused on an abstraction-refinement approach based on automatic hardware/software partitioning.

Our approach removes these constraints. Most importantly, we can handle models at levels from the initial functional specification to cycle-accurate RTL, including TLM. Time is treated continuously. Dynamic structures are handled to the extent that an upper bound on the sizes of those structures must be known. Loops may have variable upper bounds and SystemC channels are allowed, and encouraged (core part of TLM).

The approach translates SystemC models, consisting of processes and communication channels, into a Petri-Net based model. This model is then formally verified, using model checking, for properties expressed in a temporal logic.

The paper continues in section 2 with a review on the most important structures of SystemC and the design representation used. Section 3 provides an overview of the approach, while section 4 goes into more detail on how different SystemC constructs are modelled in the design representation. Experimental results are presented in section 5 and section 6 concludes the paper.

2. Preliminaries

2.1. SystemC

SystemC [1] models consist of a collection of *processes*. Each process belongs to one of three types: METHOD, THREAD or CTHREAD. However, METHOD and CTHREAD processes can be modelled as THREADs without loss of generality. Therefore, only processes of type THREAD will be considered throughout the rest of this paper.

A *scheduler* gives control to one process at a time. When a process has received control, it retains control until it explicitly releases it by executing a wait statement. It is important to note that, according to SystemC semantics, only one process may run at a time. The scheduler furthermore divides the execution into *delta cycles*. A delta cycle is finished when there are no more processes ready to run. Between two delta cycles, new processes may become ready and execution can progress. In addition, between delta cycles, time may or may not advance. Time never advances within one delta cycle.

Processes communicate through *channels*. A channel is an object which implements an arbitrarily complex communication protocol. Its methods may, in particular, contain wait statements in order to implement blocking calls. Normally, channels have at least one write method and one read method. *Signals* are a special type of channel. When a new value is written to it, that value is not visible to any reader until the next delta cycle. At the same time, all processes registered to the signal are notified about the change.

Events are a mechanism for one process to notify one or several other processes that something has happened which other processes are interested in. When a process is notified, the scheduler adds that process to its pool of ready processes. The scheduler will then eventually give control to that process. Signals actually use events to notify other processes when their values have changed.

`wait` statements come in a few different variants. The two most important ones include suspending a process during a specified amount of time, and suspending a process until a certain event occurs (with possible time-out). While the process is suspended, other processes may run. However, when the specified amount of time has elapsed or the specified event has been notified, the process is declared ready to run again. Using `wait` statements with timing is the only way to specify time, or make time advance. Other statements are considered to be instantaneous.

2.2. PRES+

The proposed approach uses a design representation called Petri-net based Representation for Embedded Systems (PRES+) [7]. PRES+ is a Petri-net based representation with extensions, some of which are listed below. Figure 1 shows an example of a PRES+ model.

1. Each token has a value and a timestamp associated to it.
2. Each transition has a function and a time delay interval associated to it (time intervals are denoted in square brackets). When a transition fires, the value of the new token is computed by the function, using the values of the tokens which enabled the transition as arguments. The timestamp is increased by an arbitrary value from the time delay interval. In figure 1, the functions are marked on the outgoing edges from the transitions.
3. The PRES+ net is forced to be safe, i.e. one place can at most accommodate one token. A token in an output place of a transition disables the transition.
4. The transitions may have guards (conditions in square brackets on transitions t_2 and t_3 in figure 1). A transition can only be enabled if the value of its guard is true.

Places with no incoming arcs are called in-ports, and places without outgoing arcs are called out-ports.

3. Overview of the approach

The SystemC model to be verified is first translated into a PRES+ model, while maintaining the semantics of the original SystemC model.

The properties to be verified are expressed in computation tree logic (CTL) [8], possibly augmented with time [9]. The properties are originally expressed in terms of SystemC constructs and, consequently, are also translated so that they refer to st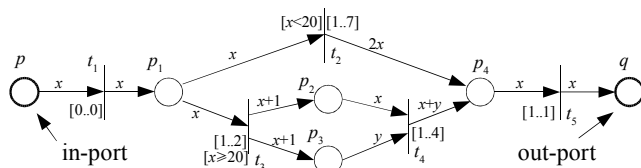ates in PRES+. The PRES+ model is then given to a model checker together with the property to be verified. Diagnostic traces, obtained from the model checking, can straightforwardly be translated back to SystemC from PRES+. The designer consequently never has to bother about PRES+, he/she only sees SystemC.

For verification of the actual models, we used the UPPAAL model checker [10], according to the methodology developed by us for verification of PRES+ models [7]. This paper concentrates on the translation from SystemC to PRES+, in particular the representation of some specific non-trivial SystemC features.

4. Translating SystemC to PRES+

4.1. Basic concepts

In PRES+, each SystemC statement is represented by one place and one transition. The transition performs the actual statement, whereas a token in the place enables the execution of the statement. Variables are also represented by places. There must be a token in the place during the whole life span of that variable. Statements assigning a value to a variable put tokens in the variable's place, and the token is removed when the execution has reached a statement out of its scope. Places for global variables always contain tokens, and fields in objects contain tokens as long as the object exists.

Figure 2 provides an example of this procedure. All transitions have time delay interval [0..0], but the delays are omitted in the figure to avoid clutter.

Statements 1 and 2 introduce and initialise new variables. Transitions t_1 and t_2 reflect this by adding tokens with the initial values in the places corresponding to the variable. At the same time they put a token in the places corresponding to the next statement to be executed, i.e. p_2 and p_3 respectively. Statement 3 updates x, which is straightforwardly reflected in transition t_3. This straightforward translation works well if only one variable is involved in the assignment statement. Statement 4, however, involves two variables. Since PRES+ transitions only can produce one output value, one of the variables must be explicitly fetched and a copy must be temporarily stored in a dedicated place (p_6). Without this procedure, variable x would be erroneously updated to the same value as y.
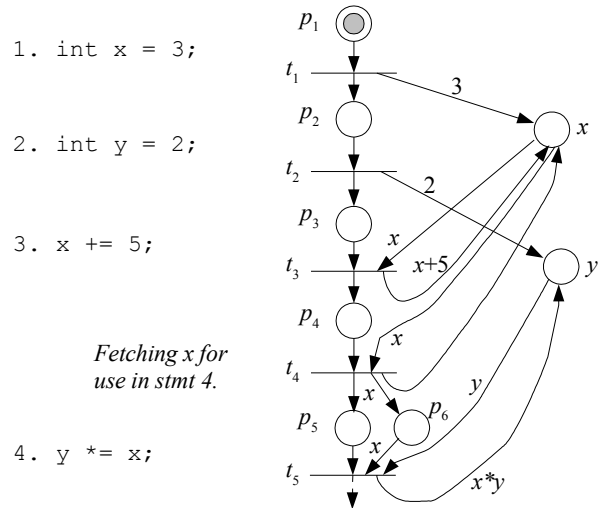

Figure 2. Translation of statements and variables

Figure 1. An example PRES+ model

```
1. int addmult(int v, int& xv) {
2.     int z = xv;
3.     xv *= v;
4.     return z+v;
5. }
6. int x = 0;
7. int y = addmult(2, x);
```
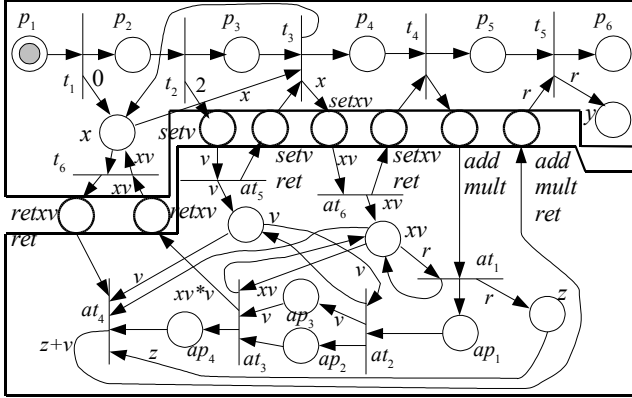
Figure 3. Translation of method calls

4.2. Method calls and interfaces

Calling a method (function) involves three steps: transfer of parameter values, transfer of control and return of control. Each of these steps must be performed explicitly in the PRES+ model. Figure 3 presents the whole scheme. The code in the figure can be divided into two parts. Lines 1 to 5 declare the method `addmult` and lines 6 and 7 introduce code that invokes addmult. Each part is translated to a PRES+ model surrounded by a box. The places between the boxes constitute the interface of the method. These places are called *ports* (see section 2.2). Only by looking at the method header (line 1), it is possible to deduce these ports. Each parameter *par* needs two ports, *setpar* and *setparret*. If *par* is declared as a reference, two additional ports are needed, *retpar* and *retparret*. Finally, the method *addmult* itself needs two ports, *addmult* and *addmultret*.

The method call itself is realised by transitions t_2 to t_5, where t_2 and t_3 transfer the actual parameters 2 and *x* to formal parameters *v* and *xv* respectively. As can be seen in the figure, the transfer uses the *setpar* ports (*setv* and *setxv*) to actually pass the value and the *setparret* ports (*setvret* and *setxvret*) are used to ensure that the transfer is completed before continuing, thereby maintaining the sequential execution semantics inside a process. After the parameter transfer, transition t_4 makes the actual transfer of control to the method. Control is returned back from the method through t_5. The return port, *addmultret*, carries the return value of the method, which is stored in variable *y*.

A closer look at the formal parameters of `addmult` reveals that *xv* is passed by reference, while *v* is not. This means that whenever the value of *xv* is modified, so must the corresponding actual parameter. For this reason, two additional ports (*retxv* and *retxvret*) are added to the interface.

The method call structure, in particular the part that each port must be paired with a return port, occurs in many situations described in this paper. Without this structure, sequential-

ity of a process execution cannot be maintained and the model will not reflect the SystemC semantics correctly.

4.3. Scheduler

4.3.1. SystemC execution mechanism. The main task of the scheduler is to give control to processes ready for execution, according to SystemC semantics. Processes can be declared ready for execution in one of three modes: in the current delta cycle (*immediate*), in the next delta cycle or at a specified time moment.

Another important task of the scheduler is to update the signals between two successive cycles so that values written to signals during the previous cycle are available for reading in the following one.

The scheduler repeatedly performs, a bit simplified, the following steps:
1. Select a process ready for execution and give control to it. New processes may be declared ready for execution during the execution of the process (*immediate* notification). Repeat for each ready process until no more ready processes exist.
2. Update all signals.
3. Let time advance to the ready time of the earliest pending process. If the earliest pending ready time is identical with the current one, a new delta cycle is introduced. Go to step 1.

In step 1, processes ready to run are selected for execution. However, during the execution of one processe, other processes may become ready during the same delta cycle (immediate notification). These processes must also be executed before steps 2 and 3 may be performed.

4.3.2. PRES+ model. The PRES+ scheduler model provides the following services through the ports depicted in figure 4.
1. Give execution control to processes.
2. Receive notice of a process becoming ready.
3. Update signals.

Execution control (service 1) is given to processes through the ports labelled *trigger*. There is one *trigger* port for each process in the system, each dedicated to its specific process. A token in *trigger1* signifies that process 1 may execute. When a process releases control, it puts a token in *yield*, a port common to all processes.

Ports *mkready*, *mkreadyret*, *mkimmready* and *mkimmreadyret* are used for notifying the scheduler about the fact that a process became ready (service 2). There is only one instance of each such port. A process can become ready in two modes: in the current delta cycle (immediate) or in the next delta cycle. A token with a unique process identifier (pid) associated to the process to become ready is placed in *mkimmready* or *mkready* depending on the intended mode. The other two ports are return ports (see section 4.2).
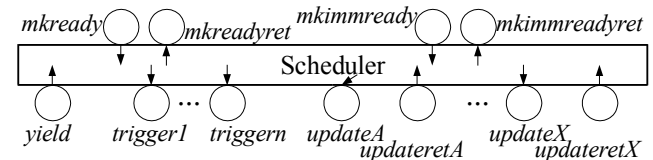
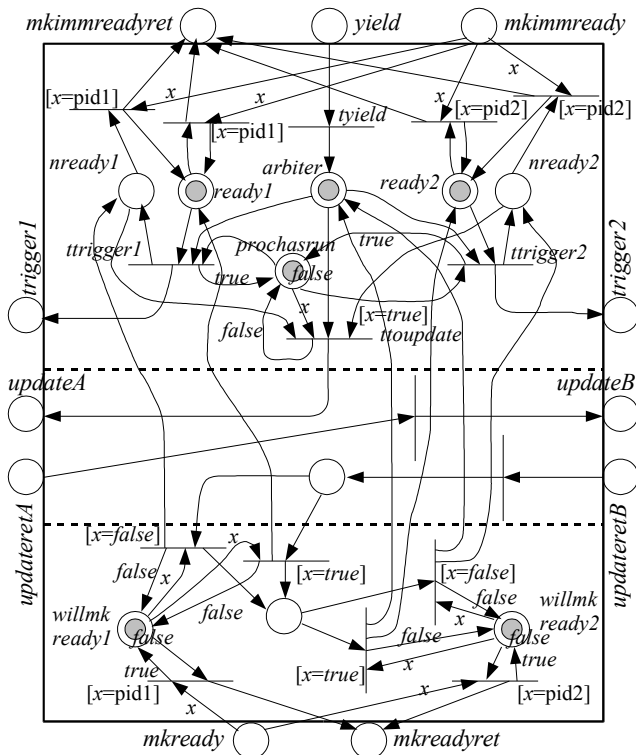**Figure 4. The interface of the scheduler**

Figure 5. A scheduler

Signal updates (service 3) are performed through the ports *update* and *updateret*. There is one pair of these ports for each signal in the system. A token in *update* causes the associated signal to be updated (see section 4.6).

Figure 5 shows a scheduler able to handle two processes (1 and 2) and two signals (A and B). The model can be divided into three parts, separated with dotted lines in the figure.

The upper part decides which process to execute next, selecting one of the ready processes. Places *ready* and *nready* (not ready) record the readiness of a process. These places are updated either through port *mkimmready* or through places *willmkready* (implicitly through port *mkready*) in the lower part.

When there is a token in place *arbiter*, the scheduler is able to fire any of the *ttrigger* transitions associated to a ready process. This will put a token in a *trigger* port and give control to the selected process. When the process finishes, it gives control back to the scheduler, by putting a token in the *yield* port. As result, the *arbiter* place again holds a token and a new process may be selected for execution. When there are no more ready processes, the scheduler enters the middle part by firing transition *ttoupdate*.

In case no process will ever become ready at the current time, the scheduler will loop through the three parts of the scheduler indefinitely. This is due to the fact that all places *nready* will, in this scenario, always contain a token, and when a token arrives in *arbiter*, the only enabled transition is *ttoupdate*. The scheduler therefore has to fire that transition and continue to the middle and lower parts before returning to the upper part. This infinite loop takes zero time, thus disallowing time to advance. Transition *ttoupdate* should therefore only fire if at least one process has executed in the current delta cycle. In order to prevent the infinite loop, the place

prochasrun is introduced. It always contains a token with a boolean value, initially false. The value in this token becomes true (see transition *ttrigger*) only if a process has executed in the current delta cycle. The guard on *ttoupdate* only allows the transition to fire if at least one process has executed. *ttoupdate* furthermore restores the value in *prochasrun* when fired. Using this mechanism, the scheduler will remain in the upper part if there does not exist any ready processes in the beginning of a delta cycle. It can only continue when a process is notified as immediately ready again.

The middle part of the scheduler notifies all signals that the system now enters a new delta cycle. This is performed by putting a token into the ports *update*. Section 4.6 provides a more detailed example of signal updates.

The lower part makes those processes ready that are marked to become ready in the next delta cycle. The truth value of the token in *willmkready* indicates if the associated process should be made ready or not. After that, a token is again placed in place *arbiter*, and the cycle is closed.

4.3.3. Comments. According to the execution mechanism described in section 4.3.1, processes can be made ready in three different modes. The PRES+ model in section 4.3.2 only handles two: in the current delta cycle and in the next delta cycle. It does not make processes ready at arbitrary time moments. The advance of time is handled by the processes themselves, as will be discussed in section 4.5.

4.4. Events

Events provide the service of making processes ready with one method call, `notify`. Each process interested in listening to the event must subscribe to it. Figure 6 presents the PRES+ interface of an event.

Event notifications can be carried out in one of two modes: immediately in the current delta cycle or in the next delta cycle. A notification is invoked by a process by putting a token in either port *immnotify* or *notify*. When that happens, the event object takes all processes subscribed to the event and makes them ready. This is done by notifying the scheduler through the ports *mkready* or *mkimmready*, depending on whether it is an immediate notification or not. The tokens placed in those ports contain the pid of the process in question. Ports *subscr* and *unsubscr* are used to dynamically subscribe and unsubscribe a process to/from the event by placing a token with the pid of that process in the respective place. The internal structure of an event is not shown due to space limitations.

4.5. `wait` statements

There are mainly two types of `wait` statements which can be executed by a process: waiting for an event, or waiting for a certain amount of time. Both types, however, follow the same basic PRES+ structure. In the following, only the latter
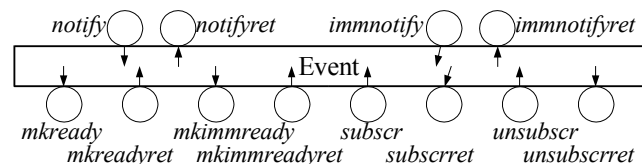

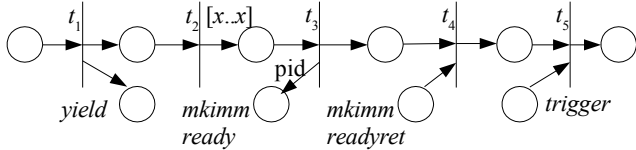**Figure 6. The interface of an event**

Figure 7. Translation of a wait statement

type will be explained (the former can be derived from this and section 4.4). Figure 7 depicts the procedure.

Transition, t_1, hands back the control to the scheduler by putting a token in the port *yield* of the scheduler. While other processes are allowed to execute, transition t_2 measures the specified amount of time, x. When the time has elapsed, t_3 notifies the scheduler by placing a token containing the process identifier in the port *mkimmready* of the scheduler, thereby notifying the scheduler that the process is again ready to execute. Note that at this point, time has advanced according to step 3 in section 4.3.1. Transition t_4 ensures that the scheduler has received the notification and finally, the process has to wait until it actually regains control from the scheduler (port *trigger*) in transition t_5.

4.6. Signals

Signals are a special type of communication channels, which require additional synchronisation with the scheduler after each delta cycle. They have to maintain two variables, *curval* and *newval*. When writing to a signal, *newval* is modified, whereas reading is performed on *curval*. After each delta cycle, *curval* must be updated to the same value as *newval*. At the same time, processes subscribing to value changes on the signal must be notified. This is done through an event, which is located inside the signal. Figure 8 shows the interface of a signal.

Signals have four methods: `read`, `write`, `event` and `update`. The read and write methods read and write from/to the signal respectively. The method event is a boolean method, which returns true if and only if the signal value was changed during the previous delta cycle. Place *isevent* keeps track of this status. Port *update* is used by the scheduler to announce a new delta cycle. The ports *mkready* and *mkreadyret* belong to the event located inside the signal. They are used for making subscribing processes ready upon value changes.

Figure 9 shows the PRES+ model of a signal. The following explanation will focus on the update mechanism. The read, write and event operations are relatively straightforwardly implemented as method calls. They update or retrieve the values of *curval*, *newval* and *isevent* respectively.

Places p_1 and p_9 record whether there has been a write in the current delta cycle. A token in p_1 means that a write has taken place, a token in p_9 has the opposite meaning. Transitions t_2, t_6 and t_{10} update these places to reflect this situation. Depending on p_1 and p_9, either transition t_4 or t_{13} is fired upon an update request from the scheduler. Transition t_{13} is fired if there was no write. It immediately returns from the update
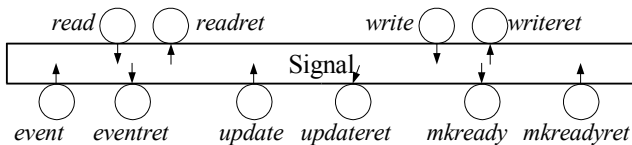

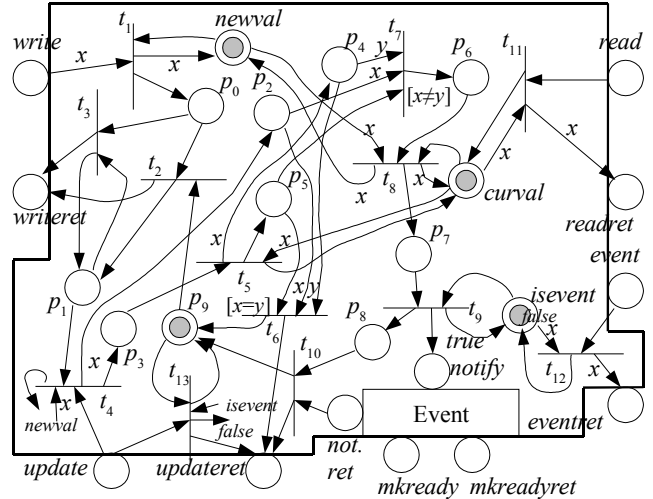**Figure 8. The interface of a signal**

Figure 9. Translation of a signal

operation and sets *isevent* to false. Transition t_4, on the other hand, is the start of a longer chain of transitions. Transitions t_4 and t_5 serve the purpose of fetching the values of *newval* and *curval*, placing copies in p_2 and p_4 respectively, in preparation for comparison by t_6 and t_7. According to the SystemC semantics, processes subscribing to value changes on the signal should not be notified unless *newval* $\neq$ *curval*. Hence, if these two values are equal, then no update and notification should be done (t_6). However, if they are not equal (t_7), *curval* is updated to the same value as *newval* (t_8) and all subscribing processes are notified via an event (t_9). Finally, the signal update returns (t_{10}).

5. Experimental results

The presented verification approach has been tried on several examples. The experiments were run on machines with Intel Xeon 2.2GHz processors and 2GB of primary memory running the Linux operating system.

5.1. Router

This example, modelling a router at transaction level, is taken from the TLM reference implementation [11]. The router forwards messages from one master to one of two slaves.

The model was verified for four different properties.
1. If a request is issued, then a response must come in the future.
2. If a message is sent to slave 1, it will arrive there.
3. If a message is sent to slave 2, it will arrive there.
4. If a message is sent to slave 2, it will not arrive at slave 1.

Table 1 presents the results. All properties were found satisfied within a few seconds' time.

5.2. Packet switch

The packet switch example is a slight modification of the pkt_switch example shipped with the SystemC reference implementation [1]. One or more masters, modelled as in the original example, send messages to one or more slaves. The switch distributes the messages to their right destinations. In order to cope with messages coming in bursts, the switch model contains one FIFO queue for each master and each slave.

Four properties were verified:
1. No deadlock.

Table 1. Results from the Router example

Property	Verification time(s)
1	3.6
2	1.2
3	1.2
4	1.5

Table 2. Results from the Packet Switch example

Property	Verification time(s)			
	1m 1s	1m 2s	2m 1s	2m 2s
1	1.1	58.54	39.55	18080.6
2	0.53	1.64	3.13	9.46
3	0.44	0.9	1.48	3.71
4	0.72	28.74	19.11	15375.0

Table 3. Results from the TL AMBA example

Property	Verification time(s)			
	1m 1s	1m 2s	2m 1s	2m 2s
1	8.95	86.88	81.65	7358.26
2	19.17	182.16	219.94	3281.34
3	1.00	2.58	2.88	8.34
4	13.16	90.95	115.46	3408.00

Table 4. Results from the SL AMBA example

Property	Verification time(s)			
	1m 1s	1m 2s	2m 1s	2m 2s
1	34.54	506.09	129.73	4339.27
2	21.57	328.79	81.52	3328.71
3	10.20	64.73	35.95	219.41
4	35.83	449.45	139.47	4212.40

2. All messages sent by a master will be received by a slave.
3. Slaves may receive messages.
4. The switch will forward every message it receives.

Property 2, initially surprisingly, turned out to be false. The reason lies in the semantics of the signals connecting the switch with masters and slaves. An event notification only occurs in the case when, during an update, the new value is not equal to the current one. If several subsequent messages are identical, only the first message will actually pass the signal and reach the switch. Consequently, the subsequent (identical) messages will not reach their destinations. The model, as given in [1], misses such consecutive messages that are identical.

The experiments were conducted using several different combinations on the number of masters and slaves. Table 2 shows the verification times. Verifying 2 masters and 2 slaves for properties 1 and 4 took between 4 to 5 hours. Verification of the other properties only took a few seconds.

5.3. AMBA bus

An AMBA bus consists of three entities: arbiter, address bus and data bus. Masters sending on the bus must first request access to it through the arbiter, and the arbiter will then eventually grant access. Slaves have the possibility of delaying or temporarily blocking (splitting) incoming messages. However, they must eventually accept all messages. Messages are transmitted pipelined on the bus. First, the address is sent on the address bus. The associated data is sent on the data bus only in the next clock cycle. As data is sent, the address of the next message is simultaneously transmitted on the address bus, hence the pipeline.

The AMBA bus example has been modelled in two versions, at different levels of detail, transaction-level and signal-level. At the transaction level, communication is implemented in a channel and transmissions are method calls with the outside behaviour of a real AMBA bus. The signal implementation, on the other hand, explicitly implements all signal exchanges between the bus, arbiter, and masters and slaves. The signal implementation is consequently more detailed than the transaction-level model.

The properties verified on both models are:
1. No deadlock.
2. If a master requests the bus, the request will eventually be granted.
3. A master may request access to the bus.
4. Messages sent by a master will always eventually be read and acknowledged by a slave.

Tables 3 and 4 present the results of the transaction-level and signal-level AMBA bus examples respectively.

6. Conclusions

The popularity of SystemC is growing, for several good reasons. In order to more efficiently trap corner case design errors, formal verification has to be employed. We have presented an approach to formally verify (model check) SystemC designs, in particular at high levels of abstraction. The approach uses a Petri-net based internal representation. We have focused on the translation of some SystemC mechanisms into this representation.

7. References

[1] M. Baird, "SystemC 2.0.1 Language Reference Manual", *Open SystemC Initiative*, 2003
[2] J. Ruf, D.W. Hoffmann, T. Kropf et al., "Simulation-guided Property Checking based on Multi-valued ar-automata", *Proc. DATE*, 2001, pp. 742-748
[3] F. Ferrandi, M. Rendine, D. Scuito, "Functional Verification for SystemC Descriptions using Constraint Solving", *Proc. DATE*, 2002, pp. 744-751
[4] D. Große, R. Drechsler, "Formal Verification of LTL Formulas for SystemC Designs", *Proc. ISCAS*, 2003, pp. 245-248
[5] D. Große, R. Drechsler, "CheckSyC: An Efficient Property Checker for RTL SystemC Designs", Proc. ISCAS, 2005, pp. 4167-4170
[6] D. Kroening, N. Sharygina, "Formal Verification of SystemC by Automatic Hardware/Software Partitioning", *Proc. MEMOCODE*, 2005, pp. 101-110
[7] L.A. Cortés, P. Eles, Z. Peng, "Verification of Embedded Systems using a Petri Net based Representation", *Proc. ISSS*, 2000, pp. 149-155
[8] E.M. Clarke, O. Grumberg, D.A. Peled, *Model Checking*, MIT Press, 1999
[9] R. Alur, C. Courcoubetis, et al., "Model-checking for Real-time Systems", *Proc. Logic in Computer Science*, 1990, pp. 414-425
[10] UPPAAL homepage, http://www.uppaal.com/
[11] A. Rose, S. Swan, J. Pierce et al., "Transaction Level Modeling in SystemC", *Open SystemC Initiative*, 2005