Model Checking SystemC Designs Using Timed Automata

Paula Herber
Software Engineering for Embedded Systems Group
Technical University of Berlin
Germany
pherber@cs.tu-berlin.de

Joachim Fellmuth
Software Engineering for Embedded Systems Group
Technical University of Berlin
Germany
fellmuth@cs.tu-berlin.de

Sabine Glesner
Software Engineering for Embedded Systems Group
Technical University of Berlin
Germany
glesner@cs.tu-berlin.de

ABSTRACT

SystemC is widely used for modeling and simulation in hardware/software co-design. Due to the lack of a complete formal semantics, it is not possible to verify SystemC designs. In this paper, we present an approach to overcome this problem by defining the semantics of SystemC by a mapping from SystemC designs into the well-defined semantics of UPPAAL timed automata. The informally defined behavior and the structure of SystemC designs are completely preserved in the generated UPPAAL models. The resulting UPPAAL models allow us to use the UPPAAL model checker and the UPPAAL tool suite, including simulation and visualization tools. The model checker can be used to verify important properties such as liveness, deadlock freedom or compliance with timing constraints. We have implemented the presented transformation, applied it to two examples and verified liveness, safety and timing properties by model checking, thus showing the applicability of our approach in practice.

Categories and Subject Descriptors

B.6.3 [**Logic Design**]: Design Aids — *verification*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages

General Terms

Verification

Keywords

SystemC, Timed Automata, Model Checking

1. INTRODUCTION

Embedded systems are usually composed of deeply integrated hardware and software components, and they are developed under severe resource limitations and high quality requirements. Thus, a language is required that supports design space exploration and quality assurance efficiently throughout the whole design process even for large and heterogeneous systems. SystemC [11] is such a language, where simulation is used for performance evaluation, design space exploration, and for validation and verification. For quality assurance, however, simulation is necessary but not sufficient. It requires formal verification techniques to guarantee liveness, safety and timing properties. A vital precondition to verify such properties is a formal semantics.

In this paper, we address the problem of defining a formal semantics for SystemC. We require our formal semantics to fulfill the following criteria: First, the behavioral semantics of SystemC informally defined in [11] must be completely preserved. Second, to maintain comprehensibility, the structure of a given SystemC design has to be preserved. Third, we want the formal model of a given SystemC design to be generated automatically. Fourth, the formal semantics must be suitable for model checking, and fifth, there should be tool support to edit, visualize and simulate the formal model of a given SystemC design.

In our approach, we obtain such a formal semantics by a mapping from SystemC designs into the well-defined semantics of UPPAAL timed automata [3]. The UPPAAL tool suite enables model checking, simulation and animation of timed automata models. Furthermore, UPPAAL timed automata have the expressiveness to represent the full semantic scale of SystemC designs, except for dynamic process or object creation and under the restriction that only bounded integer data variables are used. As we will see, these are minor restrictions. Interactions between parallel processes, including dynamic sensitivity and timing behavior, can be naturally modelled. Compared to other state based modeling languages, UPPAAL especially well-suited to model and to verify timing behavior. This is vital, as system designs often contain synchronous hardware and asynchronous software. In both the SystemC design and the UPPAAL model systems are regarded as networks of communicating processes. In our transformation approach, we map SystemC processes to UPPAAL processes. The execution of these processes is controlled by a timed automaton that models the SystemC scheduler. We use parameterized timed automata for events and for primitive channels. The timed automata modeling SystemC processes, events, channels and the scheduler are synchronized by UPPAAL channels. Our method to ensure the correctness of the transformation is twofold. On the one hand, we ensure that the transformation of SystemC processes into timed automata processes preserves their informally defined behavior. On the other hand, we ensure that the semantics of interactions between processes is preserved.

CODES+ISSS'08, October 19–24, 2008, Atlanta, Georgia, USA.

There have been several approaches to give SystemC a formal semantics. A definition of the simulation semantics based on abstract state machines is given in [15, 16]. The purpose of their work is to provide a precise description of the SystemC scheduler. However, the system design itself, as built from modules, processes and channels, is not covered and therefore cannot be verified with this approach. In [17], a denotational semantics for the SystemC scheduler and for SystemC processes is presented, but only for a synchronous subset. Similarly, in [7, 8] some work on formal verification of SystemC designs is done, but only for the synthesizable subset. In contrast to our approach, they are not able to cope with dynamic sensitivity or timing. In [10, 9], program transformations from SystemC into equivalent state machine are proposed. In these approaches, time is ignored, and the transformation is performed manually. Besides, the state machine models do not reflect the structure of the underlying SystemC designs. In [13], the formal language SystemC$^{\mathbb{FL}}$ is proposed, which is based on process algebras and defines the semantics of SystemC processes by means of structural operational semantics style deduction rules. SystemC$^{\mathbb{FL}}$ does not take dynamic sensitivity into account, and considers only simple communications. The concept of channels is neglected. A tool to automatically transform SystemC to SystemC$^{\mathbb{FL}}$ is presented in [14]. However, it does not handle any kind of interaction between processes. In [12], SystemC designs are verified using a petri-net based representation. This introduces a huge overhead because interactions between subnets can only be modeled by introducing additional subnets.

With our approach we can handle all relevant SystemC language elements, including process execution, interactions between processes, dynamic sensitivity and timing behavior. The informally defined behavior and the structure of SystemC designs are completely preserved. The mapping from SystemC designs into Uppaal timed automata is fully automated, introduces a negligible overhead, produces compact and comparably small models and enables the use of the Uppaal model checker and tool suite. This paper is organized as follows: In Section 2 and 3, we briefly review SystemC and Uppaal. In Section 4, we present our transformation from SystemC to Uppaal. In Section 5 we show the applicability of the approach by experimental results, and we conclude in Section 6.

2. SYSTEMC

SystemC [11] is a system level design language and a framework for HW/SW co-simulation. It allows for the modeling and execution of system level designs on various levels of abstraction, including classical register transfer level hardware modeling and transaction-based design. SystemC is implemented as a C++ class library which provides the language elements and an event-driven simulation kernel. A SystemC design is a set of communicating processes, triggered by events and interacting through channels. Modules and ports are used to represent structural information. SystemC also introduces an integer-valued time model with arbitrary time resolution.

The execution of a SystemC design is controlled by the SystemC scheduler. It controls the simulation time, the execution of processes, handles event notifications and updates primitive channels. Like typical hardware description languages, SystemC supports the notion of delta-cycles. Delta-cycles are used to impose a partial order on simultaneous actions and split the concurrent execution of processes into two phases. In the first phase, concurrent processes are evaluated, i. e., their method body is executed. This may include read and write accesses to primitive channels, which store changes in temporary variables. In the second phase, the actual channel state is updated. A delta-cycle lasts an infinitesimal amount of time, and a finite number of delta-cycles may be executed at one point in simulation time.

The simulation semantics of a SystemC design can be summarized as follows: **1.** Initialization: each process is executed once, **2.** Evaluation: all processes ready to run are executed in arbitrary order, **3.** Update: primitive channels are updated, **4.** if there are delta-delay notifications, the corresponding processes are triggered and steps 2 and 3 are repeated, **5.** if there are timed notifications, simulation time is advanced to the earliest pending timed notification and steps 2 – 4 are repeated, **6.** if there are no timed notifications remaining, simulation is finished. For a more comprehensive description of the SystemC simulation semantics, we refer to [6, 15, 16].

3. UPPAAL TIMED AUTOMATA

Timed Automata [1] are a timed extension of the classical finite state automata. A notion of time is introduced by clock variables, which are used in clock constraints to model time-dependent behavior. Systems comprising multiple concurrent processes are modelled by networks of timed automata, which are executed with interleaving semantics and synchronize on channels. Uppaal [3, 4, 2] is a tool set for the modeling, simulation, animation and verification of networks of timed automata. The Uppaal model checker enables the verification of temporal properties, including safety and liveness properties. The simulator can be used to visualize counterexamples produced by the model checker.

The Uppaal modeling language extends timed automata by introducing bounded integer variables, binary and broadcast channels, and urgent and committed location. Timed automata are modeled as a set of locations, connected by edges. The initial location is denoted by ◎. Invariants can be assigned to locations and enforce that the location is left before they would be violated. Edges may be labeled with guards, synchronizations and updates. Updates are used to reset clocks and to manipulate the data space. Processes synchronize by sending and receiving events through channels. Sending and receiving via a channel `c` is denoted by `c!` and `c?`, resp. Binary channels are used to synchronize one sender with a single receiver. A synchronization pair is chosen non-deterministically if more than one is enabled. Broadcast channels are used to synchronize one sender with an arbitrary number of receivers. Any receiver that can synchronize must do so. Urgent and committed locations are used to model locations where no time may pass. Urgent locations are graphically depicted by the symbol Ⓤ, committed locations by the symbol Ⓒ. Leaving a committed location has priority over leaving non-committed locations.

A Uppaal model comprises three parts: global declarations, parameterized timed automata (TA templates) and a system declaration. In the global declarations section, global variables, constants, channels and clocks are declared. In the system declaration, TA templates are instantiated and the system to be composed is given as a list of timed automata.

Figure 1: Method transformation

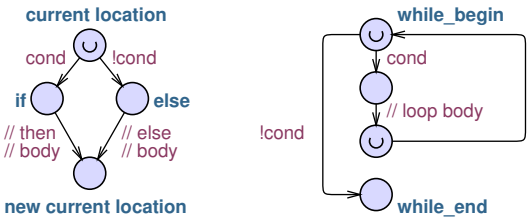

Figure 2: If-else and while transformation

4. DESIGN TRANSFORMATION

In this section we describe how we map SystemC language elements to timed automata representations and how these mappings are embedded in the complete design transformation. The transformation preserves the (informally defined) behavioral semantics and the structure of a given SystemC design. Our approach requires two minor restrictions. First, we do not handle dynamic process or object creation. This should hardly narrow the applicability of the approach, as dynamic object and process creation are rarely used in SystemC designs. Second, UPPAAL supports only bounded integer variables. This is a minor restriction as well, as most data types used in SystemC designs can be converted to bounded integers.

4.1 Method Transformation

Executable SystemC code is solely contained in methods. Each method is translated into a single TA template. To model call-return semantics, we use a synchronization channel `m_ctrl` as shown in Fig. 1. The given two automata behave as in classical call-return semantics. The caller, depicted on the left, hands control over to the callee with `m_ctrl!`, waits until the method body of the callee is executed, and resumes execution when receiving `m_ctrl?`. Fig. 1 also shows how an argument `m_param` and a return value `m_result` are transferred to and from the method, resp. The control transfer channel and the input and output parameters of a method are visible as parameters of the TA templates of the method and the caller. This enables multiple instantiations of a method and the connection with different callers in the system declaration.

The ***method body*** is a list of statements, which can be of type *return*, *arithmetic*, *if-else*, *while*, *continue*, *break*, or *method call*. For each statement, we append transitions and locations. In the following, we use the term *current location* to refer to the lastly appended location in each transformation step. We also keep a reference to the initial location. If we reach a ***return*** statement, we connect the current location with the initial location and label this transition with `m_ctrl!` and possibly with the assignment of the return value as shown in Fig. 1. A return statement aborts the transformation of the current block, and subsequent statements are dead code. We adopt ***arithmetic*** statements as *updates* at transitions in the TA template. Therefore, we make the current location urgent, append a transition with the arithmetic statement as *update* label, and use a new location as target location. To transform ***if-else*** statements, we append two outgoing edges as shown on the left in Fig. 2, one of them labeled with the if-condition, the other with the negated if-condition. We make the current location urgent because the evaluation of the if-condition takes no time. We append the statements of the if-body to the `if`-location and the statements of the else-body to the `else`-location. If there is no return statement in both branches, they are joined in a new current location. If there is a return statement in one of the branches, we use the last node of the other branch as new current location. If there are return statements in both branches, transformation of the current block is finished. We model the condition of ***while*** loops like ***if-else*** conditions, as shown on the right in Fig. 2. The statements in the loop body are appended to the location reached by the fulfilled *while*-condition. For *return* statements, the current location is connected with the initial location, for a *continue* statement with the location `while_begin`, and for a *break* statement with `while_end`.

4.2 The Scheduler

The execution of SystemC designs is controlled by the scheduler. The basic execution units are processes. The scheduler works in delta-cycles, i. e., in evaluation and update phases. In the evaluation phase, processes which are *ready to run* are executed in non-deterministic order. In the update phase, primitive channels are updated by taking over new values.

The TA model we use to model the scheduler is given in Fig. 3. Initialization is implicit in UPPAAL, i. e., processes and methods are executed once before the main simulation loop. We can also handle `dont_initialize`, but this is omitted here due to space limitations. The scheduler starts in the evaluation phase depicted by the location `evaluate`. If there are any processes *ready to run*, the scheduler sends an activation event `activate!`. Processes *ready to run* receive this event and resume their execution. We use a binary channel for the activation to ensure that only one process is executed at a time and that processes are executed in a non-deterministic order. To ensure that the scheduler sends the activation event once for each process *ready to run*, each process increments a counter `ready_procs` when triggered, and decrements the counter when suspending itself. When there are no more processes *ready to run*, i. e., `ready_procs == 0`, the scheduler starts the update phase by going to lo-

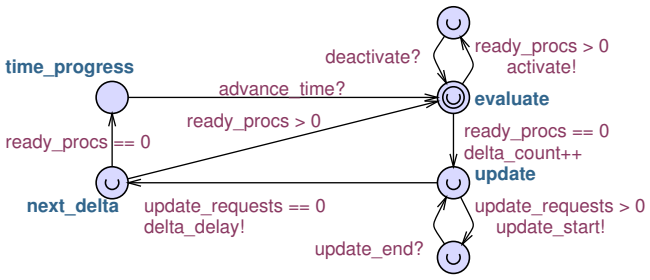

Figure 3: TA model of the scheduler

cation `update`. In the update phase, *update requests* are executed in non-deterministic order using the activation event *update_start*. Immediate notification is not allowed during the update phase. If there are no more update requests, the scheduler starts the next delta-cycle, see location `next_delta`. When leaving the update phase, the scheduler informs events with pending *delta-delay notifications* that a delta-cycle is finished by sending `delta_delay!`. If there are delta-delay notifications, the corresponding processes are immediately triggered and become *ready to run*. They will be executed in the next delta-cycle, which is started by the scheduler without time progress. If there are no processes triggered by *delta-delay notifications*, i. e., `ready_procs == 0`, simulation time must be advanced to the earliest pending timed notification. There are two types of timed notifications in SystemC: events may be notified with a delay by calling `e.notify(t)`, and processes may be delayed for a given time interval by calling `wait(t)`. In SystemC, the timing behavior is completely managed by the scheduler. In the TA model, we have the possibility to wait locally for a given time. Therefore, it is more suitable to model time within processes and event objects. A simple way to wait for the earliest pending timed notification in the scheduler is to let the processes and events with timed behavior send a broadcast synchronization `advance_time!` when their delay expires. The scheduler receives `advance_time?` and starts a new delta-cycle, i. e., executes processes which became ready to run through the *timed notification*.

The TA model of the scheduler behaves exactly like the SystemC scheduler. The binary channels used to control process execution and channel updates guarantee that the model checker considers every possible serialization. The locations used for the execution of delta-cycles are urgent and thus take no simulation time. We ensure that no scheduling phase is started before the preceding phase is completed using counters and committed locations. The counters guarantee that pending executions are completed. The use of committed locations in event notification (as shown in the next section) ensures that event triggering is prioritized over state changes in the scheduler.

4.3 Events

If an event object `e` is notified by its owner, processes that are sensitive to the event resume execution. SystemC supports three types of event notifications. An *immediate notification*, invoked by `e.notify()`, causes processes to be triggered immediately in the current delta cycle. A *delta-delay notification*, invoked by `e.notify(0)`, causes processes to be triggered at the same time instant, but after updating primitive channels, i. e., in the next delta-cycle. A *timed notification*, invoked by `e.notify(t)` with `t > 0`, causes processes to be triggered after the given delay `t`. If an event is notified that already has a pending notification, only the notification with the earliest expiration time takes effect. That means that immediate notifications override all pending notifications, delta-delay notifications override timed notifications, and timed notifications override pending timed notifications if their delay expires earlier.

We model event objects as shown in Fig. 4. The TA template is instantiated for each event object declared in a given SystemC design. Its template parameters are the synchronization channels `notify_imm`, `notify` and `wait`, and the integer variable `t`. Initially, the event just waits to be notified.

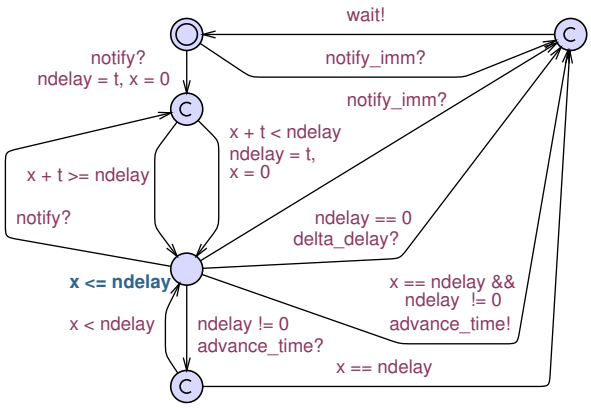

Figure 4: TA model of an event object

If it is immediately notified, it receives `notify_imm?`, and immediately sends `wait!` on a broadcast channel. If the event object is notified by a delta-delay or a timed notification, it receives `notify?` and copies the parameter `t` to a local variable `ndelay`, which yields the notification delay. At the same time, a local clock `x` is reset. The committed location that is now reached is used to reinitialize `ndelay` and to reset `x` if a subsequent delta-delay or timed notification overrides the notification delay. We then have to wait until: **1.** an immediate notification overrides the current pending notification, **2.** we receive `delta_delay?` from the scheduler if `ndelay == 0`, or **3.** the current delay expires, i. e., `x == ndelay && ndelay != 0`. Subsequently, we send `wait!` and go back to the initial location. When a timed notification expires, we have to inform the scheduler to start the next evaluation phase by sending `advance_time!`. Due to the use of a broadcast channel `advance_time!`, only the first `advance_time` is received by the scheduler if the delays of multiple events expire at the same time. As mentioned before, the preservation of the SystemC semantics requires that the scheduler must not start the evaluation phase before event notification is completed. To ensure this, event objects with pending timed notification also synchronize with `advance_time?` as receivers. If they receive `advance_time?` and their delay expires in the same time instant, i. e., if `x == ndelay`, they immediately trigger pending processes. Otherwise, nothing happens. The semantics of broadcast synchronization ensures that events with expiring delays reach the committed location in the same semantic step as the scheduler reaches the evaluation phase. The committed location ensures that events are prioritized in the next semantic step.

4.4 Processes and Sensitivity

Processes are the basic execution unit in SystemC. Each process is associated with a method to be executed. There are two types of processes: *method processes* and *thread processes*. A *method process*, when triggered, always executes its method body from the beginning to the end. It is triggered by a set of events given in a static sensitivity list. The TA model we use to wrap a method process is shown on the left in Fig. 5. A *thread process* may suspend its execution and dynamically wait for events or a given time delay. It is triggered only once at the beginning of the simulation and runs autonomously from the time on. The TA model we use to start a thread process is given on the right in Fig. 5.

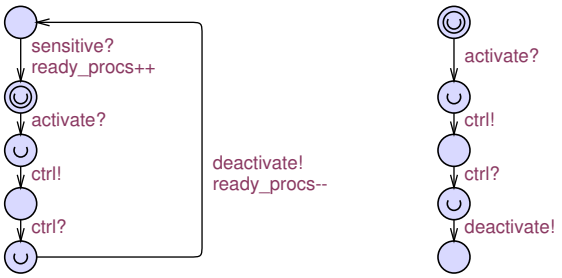

Figure 5: Process templates

A *thread process* may suspend its execution by calling a *wait* function. If `wait()` is called without parameters, it waits for one of the events in the static sensitivity list. If the process calls `wait(e)` with an event `e` as argument, the static sensitivity list is temporarily overridden by `e`. If the process calls `wait(t)`, it is delayed by `t` time units. If the process calls `wait(t,e)`, it waits for event `e` for `t` time units.

We model event sensitivity in UPPAAL using synchronization channels as shown in Fig. 6. A process calling `wait(e)` is shown on the left. It suspends its execution, i. e., synchronizes with `deactivate!`, decrements a counter `ready_procs`, and then waits to be triggered, i. e., synchronizes with the `wait` channel of the event object. When `e_wait?` is received, the process increments the counter `ready_procs` and waits to be activated by the scheduler. We can also handle waiting for composed events such as `e1 & e2` or `e1 | e2`. Static sensitivity is very similar to dynamic sensitivity, but when `wait()` is called the process waits for one of the statically known events from the sensitivity list. We model sensitivity lists by waiting for one of the given events and sending `sensitive!` on a broadcast channel, as shown on the right in Fig. 6. To ensure that immediate event notifications take effect immediately, we use a committed location. How we model static sensitivity within a sensitive process is shown in the middle of Fig. 6. Compared to dynamic sensitivity, `e_wait?` is replaced by `sensitive?`.

We model timed waiting with a special `timeout_event`. Each process has its own `timeout_event`. Calls to `wait(t)` are modeled as shown on the left in Fig. 7. First, a timed notification is released to start the timeout. Second, the process waits for the timeout to expire by synchronizing with `timeout_event_wait?`. Waiting for an event until a timing delay expires (`wait(t,e)`) requires to extend the TA model by a synchronization on `e_wait?`, as shown on the right in Fig. 7. To make sure that a `timeout_event` does not override subsequent timed notifications, we override it with an immediate event notification if event `e` occurs.

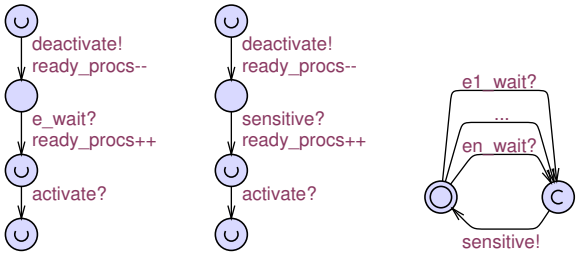

Figure 6: Event sensitivity

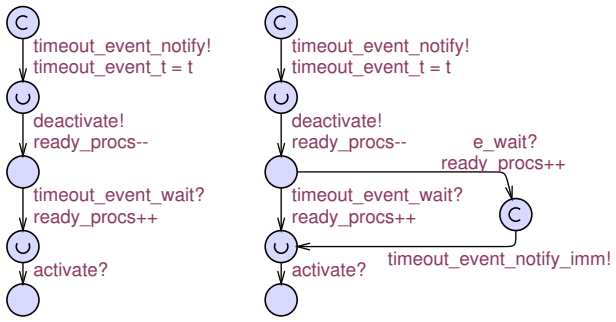

Figure 7: Timing and event sensitivity

4.5 Channels and Modules

Channels define communication methods which may be used by processes. These are translated as described above. Primitive channels have to implement an `update()`-function and call the special function `request_update()` in the evaluation phase if they want the `update()`-function to be executed in the update phase. We use a TA model to manage update requests as shown in Fig. 8. If `request_update?` is received, the `update` method of the corresponding channel is called within the update phase of the scheduler. Calls to `request_update()` in SystemC are modeled by sending `request_update!` in the TA model.

The translation of a module or channel requires that we adopt variables as (global) variables, allocate synchronization channels and parameter declarations, and generate the necessary TA templates. A module or channel may be instantiated multiple times in a SystemC design. To make method templates reusable, we take all declarations that are visible in the module as template parameters. When a module or channel is translated, the corresponding templates are generated. Global and system declarations are not added to the UPPAAL model until a module or channel is instantiated. Then, the TA templates generated from the module or channel are instantiated using these declarations. Event and process templates are generated once for the module or channel. Methods, however, may be used in multiple concurrent processes. Therefore, all methods that are visible to a module must be instantiated once for each process declared within the module. The corresponding global declarations are prefixed with the module name and the process name. Member methods of channels must be instantiated once for each process of each module which is bound to the channel.

Although there is no structural hierarchy in UPPAAL, the module structure of the SystemC design is visible through prefixes. In combination with a one-to-one mapping of SystemC to UPPAAL processes, the design structure is completely transparent to the designer. This is very useful when the model checker produces counter-examples.

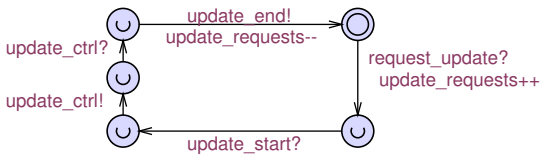

Figure 8: TA model for request-update

Table 1: Results from producer-consumer example

	Verification time in seconds			
Property	BS 10	BS 50	BS 100	BS 1000
(1)	1.78	1.78	1.78	1.81
(2)	1.78	1.78	1.78	1.81
(3)	1.82	1.95	9.22	10851

Table 2: Results from packet switch example

	Verification time(s)			
Property	1m1s	2m1s	1m2s	2m2s
(1)	22.28	56.49	43.73	211.26
(2)	3.02	3.38	3.30	4.89
(3)	129.16	46.63	298.41	544.88

5. EXPERIMENTAL RESULTS

We implemented the transformation and generated UPPAAL models from SystemC designs automatically. We used the Karlsruhe SystemC Parser KaSCPar [5] as a front-end for the SystemC designs. We applied the transformation and verified liveness and safety properties using the UPPAAL model checker. The experiments were run on a machine with an Intel Pentium 3.4 GHz CPU running a Linux operating system. The first example consists of a producer and a consumer communicating through a FIFO. The second example is a slight modification of the packet switch example included in the standard SystemC distribution. In both examples the SystemC channel concept as well as static, dynamic and timing sensitivity are used. For the producer-consumer example, we verified the following properties: (1) deadlock freedom, (2) the absence of buffer overflows, and (3) the consumer reads items sent by the producer within a given time limit. All properties were found satisfied. For the packet switch example, we checked: (1) deadlock freedom, (2) every packet is forwarded to all its receivers, and (3) if a packet is forwarded, this is done within a given time limit. Properties (1) and (3) were found satisfied, property (2) is not satisfied. Due to the semantics of `sc_signal`, the change event of signal ports is only notified if the value changes. If subsequent messages are equal, there is no change event at the input port of the packet switch and thus, only the first message is forwarded. In the producer-consumer experiments, we varied the buffer size (BS 10, BS 50, BS 100, BS 1000), in the packet switch experiments the number of masters and slaves (1m1s, 1m2s, 2m1s, 2m2s). Table 1 and 2 present the verification times averaged over 10 runs.

6. CONCLUSION

We presented an approach to translate SystemC designs into the well-defined semantics of UPPAAL timed automata. The translation enables the usage of the UPPAAL tool suite on SystemC designs, including the UPPAAL model checker to formally verify temporal properties of SystemC designs. Our general idea is to transform SystemC processes into timed automata processes and to synchronize them using channels. The execution semantics is specified using a predetermined model of the scheduler and specific templates for events and processes. The translation is performed automatically, and the transformation time is negligible. Thus, complex and large SystemC designs can be transformed. The informally defined behavior and the structure of a given SystemC design are completely preserved in the generated UPPAAL model. Moreover, the models generated by our method are compact and easily comprehensible and can efficiently be verified by model checking. In further research, we will extend our translation with several optimizations. We also plan to use the generated model for automated selection of simulation inputs and evaluation of simulation results, which is more scalable than model checking.

7. REFERENCES

[1] R. Alur and D. L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126:183–235, 1994.

[2] G. Behrmann, A. David, and K. G. Larsen. A Tutorial on UPPAAL. In *Formal Methods for the Design of Real-Time Systems*, LNCS 3185. Springer, 2004.

[3] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL — a Tool Suite for Automatic Verification of Real–Time Systems. In *Workshop on Verification and Control of Hybrid Systems*, LNCS 1066, pages 232–243. Springer, Oct. 1995.

[4] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In *Lecture Notes on Concurrency and Petri Nets*, LNCS 3098. Springer, 2004.

[5] FZI Research Center for Information Technology. KaSCPar - Karlsruhe SystemC Parser Suite.

[6] T. Groetker. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[7] D. Grosse and R. Drechsler. Checkers for SystemC designs. In *Formal Methods and Models for Codesign*, pages 171–178, 2004.

[8] D. Grosse, U. Kuhne, and R. Drechsler. HW/SW Co-Verification of Embedded Systems using Bounded Model Checking. In *Great Lakes Symposium on VLSI*, pages 43–48. ACM Press, 2006.

[9] A. Habibi, H. Moinudeen, and S. Tahar. Generating Finite State Machines from SystemC. In *Design, Automation and Test in Europe*, pages 76–81, 2006.

[10] A. Habibi and S. Tahar. An Approach for the Verification of SystemC Designs Using AsmL. In *Automated Technology for Verification and Analysis*, pages 69–83, 2005.

[11] IEEE Standards Association. IEEE Std. 1666–2005, Open SystemC Language Reference Manual, 2005.

[12] D. Karlsson, P. Eles, and Z. Peng. Formal verification of SystemC Designs using a Petri-Net based Representation. In *DATE*, pages 1228–1233, 2006.

[13] K. Man. An Overview of SystemCFL. In *Research in Microelectronics and Electronics*, volume 1, 2005.

[14] K. L. Man, A. Fedeli, M. Mercaldi, M. Boubekeur, and M. P. Schellekens. SC2SCFL: Automated SystemC to SystemCFL Translation. In *Embedded Computing Systems: Architectures, Modeling, and Simulation*, LNCS 4599, pages 34–45. Springer, 2007.

[15] W. Müller, J. Ruf, and W. Rosenstiel. An ASM based SystemC Simulation Semantics. *SystemC: Methodologies and Applications*, pages 97–126, 2003.

[16] J. Ruf, D. Hoffmann, J. Gerlach, T. Kropf, W. Rosenstiehl, and W. Müller. The Simulation Semantics of SystemC. In *DATE*, pages 64–70. IEEE Press, 2001.

[17] A. Salem. Formal Semantics of Synchronous SystemC. In *DATE*, 2003.