Electronic Communications of the EASST
Volume 53 (2012)

Proceedings of the 12th International Workshop on Automated Verification of Critical Systems (AVoCS 2012)

Optimized Transformation and Verification of SystemC Methods

Marcel Pockrandt, Paula Herber, Holger Gross, Sabine Glesner

15 pages

Guest Editors: Gerald Lüttgen, Stephan Merz
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Optimized Transformation and Verification of SystemC Methods

Marcel Pockrandt[1], **Paula Herber**[1], **Holger Gross**[1], **Sabine Glesner**[1]

[1]Technische Universität Berlin, Berlin, Germany

Abstract: Concurrent designs can be automatically verified by transforming them into an automata-based representation and by model checking the resulting model. However, when transforming a concurrent design into an automata-based representation, each method has to be translated into a single automaton. This produces a significant overhead for model checking. In this paper, we present an optimization of our previously proposed transformation from SystemC into UPPAAL timed automata. The main idea is that we analyze whether SystemC methods can be executed atomically and then we use the results for generating a reduced automata model. We have implemented the optimized transformation in our **S**ystemC to **T**imed **A**utomata **T**ransformation **E**ngine (STATE) and demonstrate the effect of our optimization with experimental results from micro benchmarks, a simple producer-consumer example, and from an Anti-Slip Regulation and Anti-lock Braking System (ASR/ABS).

Keywords: HW/SW Co-Verification, Model Checking, SystemC, Timed Automata

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. A field where the conflicting priorities are extremely hard to reconcile is the automotive sector. Automotive systems are safety-critical, as their failure may result in death or serious injury, the resource limitations are rigorous due to large quantities of production and the high cost pressure. At the same time, the quantity of digital hardware and software is heavily increasing, and already accounts for up to 30% of the overall cost of a car. To meet the high quality standards and to satisfy the rising quantitative demands, the automatization of quality assurance processes for such systems is gaining more and more importance. A major challenge is to develop *automated* quality assurance techniques that can be used for the integrated verification of complex digital HW/SW systems.

SystemC [IEE05] is a system level description language, which is widely used for the design of concurrent HW/SW systems. It is particularly well-suited for system level design, design space exploration and architecture evaluation. However, existing techniques for formal verification of SystemC are still immature and lack scalability.

In previous work [HFG08, HPG11, PHG11], we have presented an approach for the automated verification of SystemC designs. The main idea is to transform a given SystemC design (with informally defined semantics) into a formally well-defined UPPAAL timed automata model [BDL04]. This enables us to automatically verify safety, liveness, and timing properties of a given SystemC design using the UPPAAL model checker [BY04].

However, the scalability of our previously presented approach is limited. In particular, a significant overhead both in terms of memory consumption and verification time is produced by methods, which are translated into automata and thus are considered as concurrent processes in the UPPAAL timed automata model. The overhead is due to two reasons: first, although SystemC uses a cooperative scheduler, method automata may interleave with event automata, which increases the number of states in the semantic state-space. Second, by adding a location to the semantic state those method automata also increase the size of a single semantic state. This is not only a problem of our SystemC to Uppaal transformation, but also a general problem of approaches which are based on the translation from concurrent programming languages into automata-based languages.

In this paper, we carefully examine the conditions under which a given method in a concurrent design can be executed atomically. Furthermore, we present an analysis, which determines whether a SystemC method can be executed atomically. Finally, we present an optimized transformation from SystemC to UPPAAL which exploits these conditions to reduce the size of the generated UPPAAL timed automata model by using native UPPAAL methods. In the UPPAAL semantics, methods are considered as atomic operations. As a consequence, the transformation of methods into native UPPAAL methods instead of translating them into single timed automata eliminates the overhead of additional locations as well as the overhead from additional interleavings. We will illustrate the effect of this optimization by experimental results from micro benchmarks, a producer-consumer example, and a larger case study, namely an Anti-Slip Regulation and Anti-lock Braking System (ASR/ABS).

The rest of this paper is structured as follows: In Section 2, we summarize related work. Then, in Section 3, we briefly introduce SystemC and UPPAAL timed automata. In Section 4, we review our previously proposed approach for the transformation from SystemC into UPPAAL timed automata [HFG08, HPG11, PHG11], which is necessary to understand the analysis and optimization presented in Section 5. We demonstrate the effect of our optimization in Section 6, and we conclude in Section 7.

2 Related Work

There have been several approaches to provide a formal semantics for SystemC in order to enable automatic and complete verification techniques. However, many of them only cope with a synchronous subset of SystemC [MRR03, RHG+01, Sal03, GKD06]. In contrast to our approach, they are not able to cope with dynamic sensitivity or timing. As a consequence, the problem of concurrently executed methods does not exist in these approaches. The same holds for the work of Bombieri et al. [BFG10], who presented an approach for model checking TLM 2.0 IPs by synthesizing RTL IP models from them and applying RTL model checkers to the resulting model. Similarly, approaches which are based on a transformation from SystemC into process algebras [Man05, GHPS09] or sequential C programs [CMNR10, CGM+11] do not face the problem of concurrently executed methods.

However, there are also some approaches that are based on a mapping from SystemC to some kind of state machine formalism. For example, Habibi [HT05, HMT06] proposed program transformations from SystemC into equivalent state machines, Traulsen et al. [TCMM07] proposed a

mapping from SystemC to PROMELA, Zhang [ZVM07] introduced the formalism of *SystemC waiting-state automata*, and Moy et al. [MMM05] provided a toolbox for the analysis of transactional SystemC designs, which is based on a transformation from SystemC to heterogeneous parallel input/output machines (HPIOM). In [NH06], the authors propose to transform SystemC models into communicating state machines. All of these approaches lack some important features, i. e., they do not support time or they use an approximation of the timing behavior and many of them require a manual transformation from SystemC into the target language. Karlsson et al. [KEP06] overcome this problem by using a petri-net based representation. However, this introduces a huge overhead because interactions between subnets can only be modeled by introducing additional subnets.

To the best of our knowledge, none of the existing approaches considers the idea of analyzing methods for a potential atomic execution and none of them makes use of the atomic execution of multiple operations in order to reduce the semantic state space.

3 Preliminaries

In this section, we briefly introduce the preliminaries that are necessary to understand the remainder of the paper. First, we give an overview over the system level design language SystemC. Then, we briefly introduce UPPAAL timed automata.

3.1 SystemC

SystemC [IEE05] 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 functional modeling, 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. From the perspective of the execution semantics, 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 a cooperative 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

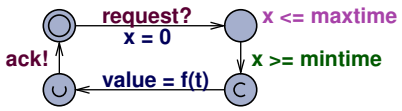


Figure 1: Example Timed Automaton

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 [Gro02, MRR03, RHG+01]. Overall, SystemC allows for the integrated development of digital hardware and software components, and it supports synchronous and asynchronous parts of a design. It is established as a premier choice for the evaluation of design alternatives and high level simulation of integrated HW/SW systems. Furthermore, a subset of SystemC can be automatically synthesized to hardware.

3.2 UPPAAL Timed Automata

Timed automata (TA) [AD94] are finite-state machines extended by clocks. A TA is a set of locations connected by directed edges. Two types of clock constraints are used to model time-dependent behavior: *Invariants* are assigned to locations and enforce progress by restricting the time the automaton can stay in this location. *Guards* are assigned to edges and enable progress only if they evaluate to true. Networks of TA are used to model concurrent processes, which are executed with an interleaving semantics and synchronize on channels. UPPAAL [BDL04] is a tool suite for modeling, simulation, and verification of TA. The UPPAAL modeling language extends TA by templates, bounded integer variables, binary and broadcast channels, and by urgent and committed locations. Templates can be used to model parameterized timed automata, which can then be instantiated with different values (by-value) or variables (by-reference). Binary channels enable a blocking synchronization between two processes, whereas broadcast channels enable non-blocking synchronization between one sender and arbitrarily many receivers. Urgent and committed locations are used to model locations where no time may pass. Furthermore, leaving a committed location has priority over non-committed locations. The UPPAAL modeling language supports a C-like action language, which also enables the use of C-like methods. A small example UPPAAL timed automaton (UTA) is shown in figure 1. The initial location is denoted by ◎, and `request?` and `ack!` denote receiving and sending on channels, respectively. The clock variable `x` is first set to zero and then used in two clock constraints: the invariant `x <= maxtime` denotes that the corresponding location must be left before `x` becomes greater than maxtime, and the guard `x >= mintime` enables the corresponding edge at `mintime`. The symbols Ⓤ and Ⓒ depict urgent and committed locations. The assignment `value = f(t)` corresponds to a call of method `f`, which itself is defined in standard C semantics. Note that the execution of method `f` is performed atomically in one semantic step from the committed to the urgent location.

4 SystemC to Timed Automata Transformation

In this section, we summarize our previously presented approach for model checking SystemC designs, which has been presented in [HFG08, HPG11, PHG11]. The general idea is to map the informally defined semantics of SystemC [IEE05] to the formally well-defined semantics of UPPAAL timed automata [BLL+95]. To this end, we have defined a UPPAAL representation for all relevant executable SystemC language elements, e. g., for assignments, method calls, events, timed notifications, static and dynamic sensitivities and for the wait-notify mechanism. In doing so, we have defined a formal semantics for the informally defined SystemC language elements. Furthermore, we presented a transformation procedure to construct a complete UPPAAL model from a given SystemC design using these representations. To ease debugging, it keeps the structure of the original SystemC design transparent to the designer in the UPPAAL model. Finally, our mapping enables the *automatic* generation of a UPPAAL timed automata model from a given SystemC design. This in turn facilitates the application of the UPPAAL model checker. Note that the UPPAAL tool suite also enables simulation and animation of the generated model and thus allows the visualization and animation of counter-examples (in UPPAAL) if the verification fails.

In the following, we first state a few assumptions that define the subset of SystemC supported by our approach. Then, we present our representation of SystemC in UPPAAL, and we briefly summarize the transformation procedure. Our transformation procedure is implemented in a tool called STATE (**S**ystemC to **T**imed **A**utomata **T**ransformation **E**ngine) and is available online at `http://www.pes.tu-berlin.de/state_project`.

SystemC enables modeling and simulation of digital hardware and software components on different levels of abstraction. To this end, SystemC supports a very diverse set of models of computation. At the same time, as an extension of C++, it inherits the full semantic scale of the C++ language. As the UPPAAL modeling language is less expressive than SystemC, we impose some restrictions on SystemC designs that can be transformed into a UPPAAL timed automata model:

- The SystemC design can be constructed statically, i. e., it does not use dynamic memory management or process creation.
- All statements that are used for instantiation and binding must be evaluable at transformation time.
- We assume that no variables are shadowed (i. e., each variable has a unique identifier in its scope), and that no method overloading is used.
- The SystemC design may only use data types that can be mapped to `int` and `bool`. This also means that we require that no pointers are used (pointer support is under construction).

The first restriction hardly narrows the applicability of the approach for safety-critical embedded systems, as dynamic object and process creation are rarely used in such designs. The second and third restriction can usually be achieved by rewriting without loss of expressivity. The last restriction is the most serious restriction in practical applications, but is acceptable as well because many data types used in SystemC designs can be converted to bounded integers. Furthermore, as

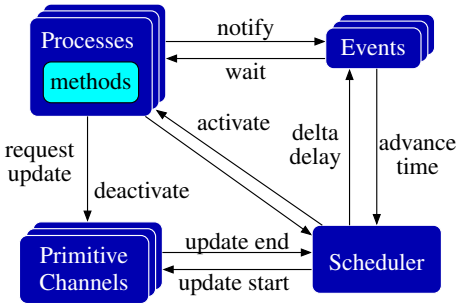


Figure 2: Representation of SystemC Designs in UPPAAL

model checking suffers from the state space explosion problem, our approach is best-suited for abstract models, where only simple data types are used.

If all of the assumptions above are met, we can automatically transform a given SystemC design into a UPPAAL timed automata model.

Figure 2 shows how we represent SystemC designs in UPPAAL. The general idea is that each method is mapped to a single timed automata template. Process automata are used to encapsulate these methods and care for the interactions with events and the scheduler. The scheduler is explicitly modeled, and we use predefined templates for events and other SystemC constructs such as primitive channels. The interactions between the processes and the scheduler are modeled by two synchronization channels, *activate* and *deactivate*, which are used by the scheduler to start (*activate*) processes, and by the processes to yield control (the SystemC Scheduler is cooperative and non-preemptive). The interactions between processes and event objects are modeled by *wait* and *notify*, with the usual meaning. The interactions between the event objects and the scheduler are used to synchronize their timing. The scheduler informs the event objects when a delta-cycle is completed to release delta-delay notifications, and conversely, the event objects inform the scheduler when time is advanced due to a timed notification. The timed automata representations for most standard SystemC language construct, such as methods, processes, and events, are given in [HFG08]. They define the execution semantics of SystemC in terms of the formally defined semantics of UPPAAL timed automata and thus provide a formal semantics for SystemC. Furthermore, they form the basis for our automatic transformation of SystemC designs into UPPAAL timed automata.

Transformation of SystemC Designs

In order to enable compositional transformation, that is to make it possible that each SystemC module can be translated separately, we perform the mapping from SystemC to UPPAAL in three steps:

1. We define a timed automata representation for each SystemC language construct (e. g.,, methods, processes, and events).

2. Using these general representations, we translate each given SystemC module into a set of parameterized timed automata.

3. We perform instantiation and binding. This requires to instantiate the parameterized timed automata, to add variables and channels, and to build the system declaration.

Note that we have to transform each module only once and can use template parameters to instantiate it arbitrarily often. When we compose a design, we instantiate the modules including their methods, events and processes and connect them via their parameters. Using this compositional approach, the transformation scales well even for large SystemC designs. The generated models are structure-preserving and thus easily comprehensible to the designer.

For the transformation of methods, we generate "method automata", which can be called by synchronizing on a control channel. In doing so, we preserve the call-return semantics of SystemC. However, in contrast to SystemC methods, method automata cannot be re-entered. As a consequence, they have to be instantiated once for each SystemC process that can possibly enter them. This produces a significant overhead in the consumption of both memory consumption and CPU time. To overcome this problem, we developed an analysis to find out by which processes a method is possibly entered, and to determine whether a method can be executed atomically.

In the following sections, we first present our analysis and its results. Then, we present an optimized method transformation that makes use of the analysis results in order to reduce memory consumption and CPU time.

5 Optimized Method Transformation

In our previously presented transformation from SystemC to UPPAAL, methods from the SystemC model are translated one-to-one into UPPAAL timed automata templates. This transformation is possible for all SystemC methods. However, every method automaton increases the memory consumption and the verification time of the UPPAAL model. On the one hand, each method automaton contains multiple locations and variables and therefore increases the size of semantic states. On the other hand, as method automata increase the number of interleavings, they also increase the size of the semantic state space. This happens although SystemC uses a cooperative scheduler because methods interleave with event automata. Our idea to overcome this problem is to transform certain methods into UPPAAL native methods instead of translating them into method automata. As those methods are then executed atomically in the sense of UPPAAL states, this transformation helps alleviate the negative impact of the methods on both verification time and memory consumption.

5.1 Analysis

In order to determine which SystemC methods in a given design satisfy the assumptions given above and thus can be executed atomically, we developed an analysis which recursively walks through the call graph of each process in the design and determines which processes can possibly be preempted and thus may *not* be executed atomically. From that we compute the set PREEMPT of methods that can possibly preempted during execution. As native UPPAAL methods are invoked during the transition from one location to another, their execution corresponds to an atomic execution. As a consequence, no time can pass during the execution of native UPPAAL methods and no synchronization with other processes (over channels) is possible.

We conservatively assume that a method can possibly be preempted if one of the following conditions is met:

1. it is bound to a process (i. e., serves as a starting point for a thread),
2. it contains a *wait* call,
3. it contains a *notify* call.

A simplified version of the algorithm for traversing the call graph is sketched in Figure 3. First, the sets PREEMPT and ATOMIC are initialized with the empty set. Then, for each module instance, we analyze the preemptability of each method using the function *isPreempt*. Note that we perform this analysis per module instance. This is necessary because module instances may be bound to different methods over ports and sockets. To avoid unnecessary overhead, the algorithm memorizes for each method whether it has already been analyzed to avoid that the same method is analyzed multiple times.

The function *isPreempt* analyzes for a given method of a given module instance whether it may preempted or not. First, the algorithm checks if the method was already analyzed. Then, it checks the preemptability conditions given above, namely whether the method uses *wait* or *notify* statements, or whether it is bound to a process. If one of these conditions is fulfilled, the method is preemptable and the analysis returns *true*. If not, the algorithms recursively constructs the call graph of the method in order to find out whether the method can be indirectly preempted by any method it calls. For each method call, the algorithm determines whether it is a member method of the given module instance or of one of it members, a method that is bound to a port, or a method that is called over a socket. Depending on the result, the analysis is continued in the corresponding module instance.

Finally, we compute the set of atomically executable methods ATOMIC by subtracting the set of possibly preemptable methods PREEMPT from the set of all methods. By overapproximating the set of possibly preemptable methods conservatively, we ensure that all methods in the ATOMIC set can be executed atomically without losing semantics preservation.

5.2 Transformation and Example

The main idea of our optimized transformation is to map all SystemC methods that allow for atomic execution into UPPAAL native methods. These methods are all elements of the *atomic* set computed by our analysis presented above. In Figure 4 we show a SystemC Module MyMod, containing two methods. The *main* method is bound to a *SC_THREAD* and calls the method *foo* for calculating some value x out of the parameter p and the class variable c. Our analysis recognizes *main* as an element of the PREEMPT set and *foo* as an element of the ATOMIC set.

Figure 5 shows the previous result of a transformation of the two methods *main* and *foo*. Note that each of the methods is modeled as a full UPPAAL template, containing the local variables of their SystemC method equivalents. The method call to *foo* is modeled by copying the parameter z into a special transport variable (*foo$param$p*) and by signaling *foo* via a binary control channel (*foo$ctrl*). While the *foo* automaton calculates the return value, the main automaton is blocked. Afterwards the return value (*foo$return*) is assigned to the value x. In general, we use one

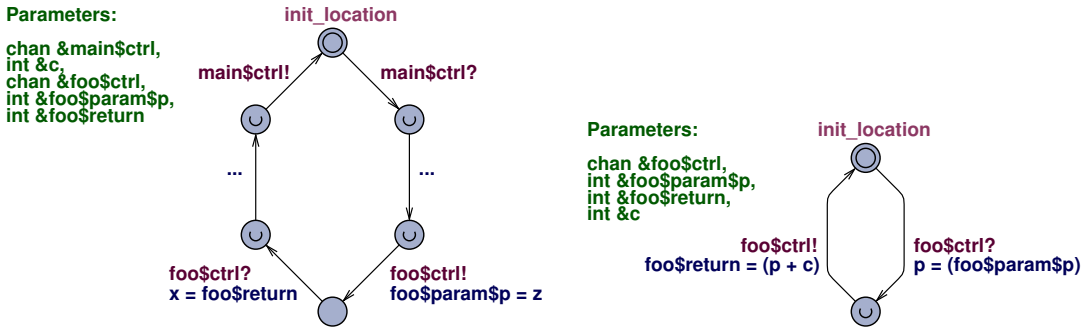
```
PREEMPT := ∅
ATOMIC := ∅
foreach M ∈ ModInsts do
  foreach m ∈ getMethods(M) do
    if isPreempt(m,M) then
      PREEMPT := PREEMPT ∪ m
    else
      ATOMIC := ATOMIC ∪ m
    end if
 end for
end for

boolean isPreempt(Method m, ModInst M)
  if analyzed(m) then
    return getPreemptTag(m)
  end if
  preempt = false
  if usesWait(m) || usesNotify(m) || boundtoProcess(m) then
     return true
  end if
  MC := getMethodCalls(m)
  foreach c ∈ MC do
     if isMember(c,M) then
       preempt = preempt || isPreempt(c,M)
     end if
     if isCalledOverPort(c,M) then
       ch = getChannel(c)
       cM = getChannelMethod(ch)
       preempt = preempt || isPreempt(cM,M)
     end if
     if isCalledOverSocket(c,M) then
       s = getSocket(c)
       sM = getModInst(s)
       preempt = preempt || isPreempt(cm,sM)
     end if
     ... // PEQs omitted here for simplification
  end for
  setPreemptTag(m, preempt)
  return preempt
```

Figure 3: Method Preemption Analysis

```
SC_MODULE(MyMod) {
  int c;
  int foo (int p) {
    return p + c;
  }
  void main() {
    int x = foo(z);
  }
  SC_CTOR(simpleModule) {
    SC_THREAD(main);
  }
}
```

Figure 4: Simple SystemC Example



(a) Template of the *main* Method

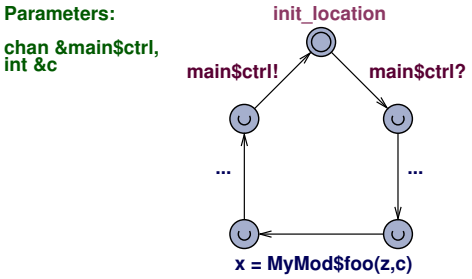
(b) Template of the *foo* Method

Figure 5: Method Transformation without Native Methods

control channel for every method call in order to model the call-return semantics between the caller and the callee. Additionally, transport variables are used for every method parameter and return value. All method templates have parameters to get instantiated with the control channels and transport variables for all methods they call and the control channels and transport variables needed to call the method itself.

Figure 6a shows the optimized template of the *main* Method. Instead of using control channels and transport variables to invoke the method *foo* and then wait until the *foo* automaton returns the calculated value, we call the native method *foo* with the parameters z and c. The native UPPAAL method *foo* is shown in Figure 6b. Note that the method is identical to the SystemC method, except for the additional parameter c. This parameter is necessary as we only generate one method *foo* for all instances of the module MyMod and therefore need to pass the class member variable c as well. In general, all methods transformed into native UPPAAL methods get additional parameters for all class variables used inside of them or inside of other native methods called by them.

The optimized transformation reduces the amount of locations and variables used in the UPPAAL model. As UPPAAL native methods are not modeled as automata, no locations exists for



(a) Optimized Template of the *main* Method

```
int MyMod$foo (int p, int &c) {
  return p + c;
}
```

(b) Native UPPAAL method *foo*

Figure 6: Optimized Method Transformation

the method itself. Additionally, we save one location in every template where such a method is called, compared to our previous transformation. With respect to the variables, all control channels and all control channels for the transformed methods are not needed anymore. The decreased amount of locations and variables reduces the semantic state space of the UPPAAL model and has a positive impact on both verification time and memory consumption.

6 Experimental Results

In this section, we present the results of three case studies to demonstrate the effect of the presented optimizations. Unless otherwise noted all experiments were run on a 3.0 GHz dual core Intel CPU with 4 GB ram running a 64bit Ubuntu. All presented results are averaged over 10 runs. For each case study, we compare verification times, the number of states calculated during model checking and the memory consumption of the model checker.

6.1 Micro Benchmark

The first model we use to evaluate our approach is a micro benchmark consisting of two methods (one of them optimizable) and one process. The process invokes a non-preemptable method to test whether the numbers between 1 and an upper bound are prime numbers. The process runs in an infinite loop and never yields control back to the scheduler. The size of this example is less than 50 lines of code (LOC). We verified the model for deadlock freedom with different upper bounds both with and without our method transformation optimization. The transformation took 0.64 seconds in the unoptimized and 0.69 seconds in the optimized version. The results from model checking are shown in Table 1 and show a drastically reduced amount of states in the optimized model, resulting in much lower memory consumption and an improved verification time. Note that in the unoptimized version an upper bound of 100,000 leads to an out of memory exception (↯). Our optimized model can even be verified with an upper bound of 100,000.

6.2 Producer-Consumer

The producer-consumer example consists of a producer and a consumer that communicate through a *first in first out* (FIFO) buffer. It uses the SystemC channel concept as well as static, dynamic,

Table 1: Results from Model Checking of the Micro Benchmark

	Upper bound for prime number calculation								
	10K			20K			100K		
	base	opt	imprv	base	opt	imprv	base	opt	imprv
CPU time (sec)	32.6	1.5	95.38 %	123.5	5.4	95.63 %	↯	110.1	-
KStates explored	8787	41	99.53 %	32176	82	99.74 %	↯	410	-
Memory (MB)	553	5.5	99.01 %	1972	7.7	99.61 %	↯	26.6	-

Table 2: Results from Model Checking of the Producer-Consumer Example

	baseline	optimized	improvement	satisfied
CPU time (sec)	2.1	0.9	57.14 %	✓
KStates explored	214535	88073	58.95 %	✓
Memory (MB)	20.6	10.5	49.03 %	✓

and timing sensitivity and thus covers many important language constructs of SystemC. Note that the design is non-deterministic, as the execution order of producer and consumer is not predetermined. In this experiment, the producer sends increasing numbers to the consumer. The consumer then checks for every number whether it is a prime number or not. The size of the producer-consumer example is approximately 130 LOC and it consists of two modules, two processes and one channel. Two methods of the model can be optimized. We verified the model for deadlock freedom. The transformation took 0.87 seconds in the unoptimized and 0.92 seconds in the optimized version. The results from model checking are shown in Table 2. The amount of states is reduced by about 59 % in the optimized model, resulting in 49 % less memory consumption and an improvement of the verification time by 57 %.

6.3 Anti-Slip Regulation and Anti-lock Braking System

The ASR/ABS system monitors the speed of each wheel in a car and regulates the brake pressure in order to prevent wheel lockup or loss of traction and to improve the driver's control over the car. For our experiment, we use an abstract design where processes communicate over FIFOs and abstract data types are used. The abstract design consists of approximately 500 LOC and contains 4 modules and 18 processes that communicate over 12 channels. Four methods of the model can be optimized. The main challenge of the ASR/ABS design lies in its heavily time-dependent behavior, which leads to a gigantic state space and makes verification very difficult. As a consequence, it is not possible to fully verify the ASR/ABS case study on a standard PC. One approach to overcome this problem is to use bit state hashing, as discussed in [HG12]. However, for evaluating the effect of our optimized method transformation, we just used a simplified version of the ASR/ABS design where the environment is not open, but constrained to a certain test trace. The transformation took 6.25 seconds in the unoptimized and 6.42 seconds in the optimized version. The results from model checking the simplified ASR/ABS design are shown in Table 3. The amount of states is reduced by about 0.13 % in the optimized model, resulting in 1.58 % less memory consumption and an improvement of the verification time by approximately

Table 3: Results from Model Checking of the ASR/ABS

	baseline	optimized	improvement	satisfied
CPU time (sec)	1574.1	1558.0	1.02 %	✓
KStates explored	19486	19461	0.13 %	✓
Memory (MB)	2901	2855	1.58 %	✓

1 %. As you can see, the results are not as impressive as they are for the micro benchmarks and for the producer-consumer example. This is due to the fact that the verification effort of the ASR/ABS design heavily depends on its timing behavior and hardly on the method execution. However, a positive effect is still visible, which shows that even for large designs where method execution is of low importance the optimization causes a measurable improvement of the verification time and the memory consumption.

7 Conclusion

In this paper, we have presented an approach for the optimized transformation and verification of SystemC methods. The main idea of our optimized transformation is to analyze the preemptiveness of SystemC methods and then to use native UPPAAL methods instead of method automata as far as possible. As native UPPAAL methods are executed atomically in the UPPAAL timed automata semantics, this reduces the semantic state space significantly for non-preempted methods. The optimization improves our previously proposed approach for the transformation of SystemC designs into UPPAAL timed automata and their verification using the UPPAAL model checker [HFG08, HPG11, PHG11].

We have shown the effect of our optimization with experimental results from three case studies: a micro benchmark that repeatedly performs non-preemptive execution of one computationally expensive method, an extended producer-consumer example where the consumer uses a non-preemptive method to compute whether consumed items are prime numbers, and an Anti-Slip regulation and Anti-lock Braking System (ASR/ABS). The micro benchmark has shown that for designs that mainly consist of computationally expensive methods, the overall effect of the optimization is enormous with more than 95 % in terms of CPU time and more than 99 % in terms of memory consumption. For the producer-consumer example, the improvement is about 60 % in terms of CPU time and about 50 % in terms of memory consumption. For the ASR/ABS case study, the improvement is still more than 1 %.

Overall, we have shown that the verification of concurrent designs can be significantly improved when parts of the systems are identified which can be executed atomically, i. e., without preemption. Note that this pays off even in the case of SystemC, which already uses a non-preemptive scheduler. Furthermore, note that we do not sacrifice the semantics and structure preservation of our previous transformation.

For future work, we plan to extend our approach for the atomic execution of multiple operations on a more fine-grained level. To this end, we plan to partition methods into parts that can possibly be preempted and other parts that can be executed atomically. The parts of the method which can be executed atomically can then be extracted into native UPPAAL methods.

This would increase the benefit from the proposed optimization.

Bibliography

[AD94] R. Alur, D. L. Dill. A Theory of Timed Automata. *Theoretical Computer Science* 126:183–235, 1994.

[BDL04] G. Behrmann, A. David, K. G. Larsen. A Tutorial on UPPAAL. In *Formal Methods for the Design of Real-Time Systems*. LNCS 3185, pp. 200–236. Springer, 2004.

[BFG10] N. Bombieri, F. Fummi, V. Guarnieri. Model Checking on TLM-2.0 IPs through automatic TLM-to-RTL Synthesis. In *VLSI System on Chip Conference (VLSI-SoC)*. Pp. 61–66. IEEE Computer Society, 2010.

[BLL+95] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, W. Yi. UPPAAL — a Tool Suite for Automatic Verification of Real–Time Systems. In *Workshop on Verification and Control of Hybrid Systems*. LNCS 1066, pp. 232–243. Springer, Oct. 1995.

[BY04] J. Bengtsson, W. Yi. Timed Automata: Semantics, Algorithms and Tools. In *Lecture Notes on Concurrency and Petri Nets*. LNCS 3098, pp. 87–124. Springer, 2004.

[CGM+11] A. Cimatti, A. Griggio, A. Micheli, I. Narasamdya, M. Roveri. Kratos - A Software Model Checker for SystemC. In Gopalakrishnan and Qadeer (eds.), *Computer Aided Verification*. LNCS 6806, pp. 310–316. Springer Berlin / Heidelberg, 2011.

[CMNR10] A. Cimatti, A. Micheli, I. Narasamdya, M. Roveri. Verifying SystemC: A software model checking approach. In *Formal Methods in Computer-Aided Design (FMCAD), 2010*. Pp. 51 –59. 2010.

[GHPS09] H. Garavel, C. Helmstetter, O. Ponsini, W. Serwe. Verification of an industrial SystemC/TLM model using LOTOS and CADP. In *International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*. Pp. 46–55. 2009.

[GKD06] D. Große, U. Kühne, R. Drechsler. HW/SW Co-Verification of Embedded Systems using Bounded Model Checking. In *Great Lakes Symposium on VLSI*. Pp. 43–48. ACM Press, 2006.

[Gro02] T. Groetker. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[HFG08] P. Herber, J. Fellmuth, S. Glesner. Model Checking SystemC Designs Using Timed Automata. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. Pp. 131–136. ACM press, 2008.

[HG12] P. Herber, S. Glesner. A HW/SW Co-Verification Framework for SystemC. 2012.

[HMT06] A. Habibi, H. Moinudeen, S. Tahar. Generating Finite State Machines from SystemC. In *Design, Automation and Test in Europe*. Pp. 76–81. IEEE, 2006.

[HPG11] P. Herber, M. Pockrandt, S. Glesner. Transforming SystemC Transaction Level Models into UPPAAL Timed Automata. In *Formal Methods and Models for Codesign (MEMOCODE)*. Pp. 161 – 170. IEEE Computer Society, 2011.

[HT05] A. Habibi, S. Tahar. An Approach for the Verification of SystemC Designs Using AsmL. In *Automated Technology for Verification and Analysis*. LNCS 3707, pp. 69–83. Springer, 2005.

[IEE05] IEEE Standards Association. IEEE Std. 1666–2005, Open SystemC Language Reference Manual. 2005.

[KEP06] D. Karlsson, P. Eles, Z. Peng. Formal verification of SystemC Designs using a Petri-Net based Representation. In *Design, Automation and Test in Europe (DATE)*. Pp. 1228–1233. IEEE Press, 2006.

[Man05] K. L. Man. An Overview of SystemCFL. In *Research in Microelectronics and Electronics*. Volume 1, pp. 145– 148. 2005.

[MMM05] M. Moy, F. Maraninchi, L. Maillet-Contoz. LusSy: A Toolbox for the Analysis of Systems-on-a-Chip at the Transactional Level. In *International Conference on Application of Concurrency to System Design (ACSD)*. Pp. 26–35. 2005.

[MRR03] W. Müller, J. Ruf, W. Rosenstiel. *SystemC: Methodologies and Applications*. Chapter An ASM based SystemC Simulation Semantics, pp. 97–126. Kluwer Academic Publishers, 2003.

[NH06] B. Niemann, C. Haubelt. Formalizing TLM with Communicating State Machines. *Forum on specification and Design Languages*, 2006.

[PHG11] M. Pockrandt, P. Herber, S. Glesner. Model Checking a SystemC/TLM Design of the AMBA AHB Protocol. In *IEEE/ACM Symposium on Embedded Systems For Real-time Multimedia (ESTIMedia)*. Pp. 66 – 75. IEEE Computer Society, 2011.

[RHG+01] J. Ruf, D. W. Hoffmann, J. Gerlach, T. Kropf, W. Rosenstiel, W. Müller. The Simulation Semantics of SystemC. In *Design, Automation and Test in Europe*. Pp. 64–70. IEEE Press, 2001.

[Sal03] A. Salem. Formal Semantics of Synchronous SystemC. In *Design, Automation and Test in Europe (DATE)*. Pp. 10376–10381. IEEE Computer Society, 2003.

[TCMM07] C. Traulsen, J. Cornet, M. Moy, F. Maraninchi:. A SystemC/TLM semantics in Promela and its possible applications. In *14th Workshop on Model Checking Software (SPIN '07)*. LNCS 4595, pp. 204–222. Springer, Berlin, 2007.

[ZVM07] Y. Zhang, F. Vedrine, B. Monsuez. SystemC Waiting-State Automata. In *First International Workshop on Verification and Evaluation of Computer and Communication Systems (VECoS 2007)*. 2007.