Formal Verification of SystemC Designs using the BLAST Software Model Checker

Paula Herber and Bettina Hünnemeyer

Technische Universität Berlin
Ernst-Reuter-Platz 7, 10587 Berlin, Germany
paula.herber@tu-berlin.de,
WWW home page: http://www.pes.tu-berlin.de

Abstract. SystemC is widely used in hardware/software codesign. Although it is also used for the design of safety-critical applications, existing formal verification techniques for SystemC are still hardly used in industrial practice. The main reasons for this are scalability issues, the lacking support for many practically relevant SystemC language constructs, and that counter-examples are not always easy to use for debugging. In this paper, we present an approach for the formal verification of SystemC designs using the BLAST model checker. The main advantages of our approach are: First, we enable a fully automatic verification of SystemC designs that makes use of counter-example guided abstraction refinement. Second, we support a large subset of SystemC, including pointers, arrays, and structs. Third, we ease debugging by keeping the structure of the design transparent to the designer. We demonstrate the applicability of our approach with experimental results from an Anti-Slip Regulation and Anti-Lock Braking system.

Keywords: SystemC, Formal Verification, Model Checking

1 Introduction

Embedded systems are often used in domains where a failure results in high financial losses or even in serious injury or death, e. g., in cars, airplanes and transportation systems. This makes it indispensable to ensure their correctness with systematic and comprehensive verification techniques. At the same time, embedded systems typically consist of deeply integrated hardware and software components, which makes comprehensive verification a difficult challenge. A language that is widely used for modeling such systems is the system level design language SystemC [10]. SystemC enables modeling and simulation of both hardware and software on various abstraction levels. SystemC designs often serve as reference model for the remainder of the development process. This in turn makes the correctness of SystemC designs a crucial issue. With simulation alone, it is not possible to cover all input scenarios and corner-cases may be overlooked. Only formal verification techniques can provide the degree of assurance needed for such key models in the design process.

In this paper, we present an approach for the formal verification of SystemC designs using the software model checker BLAST [1]. With our approach, we tackle the scalability issue by applying the BLAST model checker's capability for counter-example guided abstraction refinement. At the same time, the BLAST model checker supports a large subset of C, including pointers, arrays, and structs. With that, it is also very well-suited for supporting a large subset of SystemC designs. Our main contribution is an interpretation of the SystemC semantics that is executable as a sequential, non-deterministic C program, and keeps the structure of the design transparent to the designer, which is very important for debugging. We have implemented an automatic transformation from SystemC into the input language of BLAST, and thus we can verify SystemC designs using the BLAST model checker fully automatically. We demonstrate the practical applicability of our approach with a case study from the automotive domain, namely an Anti-Slip Regulation and Anti-Lock Braking System.

The rest of this paper is structured as follows: In the next section, we briefly introduce the preliminaries that are necessary for understanding the remainder of the paper. In Section 3, we summarize related work. In Section 4, we present the main contribution of this paper, namely our approach for the formal verification of SystemC designs using the BLAST model checker. We present experimental results in Section 5 and conclude in Section 6.

2 Preliminaries

In this section, we briefly introduce the preliminaries that are necessary to understand the remainder of the paper.

2.1 SystemC

SystemC is a system-level design language and a framework for HW/SW co-simulation. The semantics of SystemC is informally defined in an IEEE standard [10]. SystemC is implemented as a C++ class library, which provides the language elements for the description of hardware and software, and allows for the modeling of both hardware and software on different levels of abstraction. It also features an event-driven simulation kernel, which enables the simulation of the design during the whole development process.

Like typical hardware description languages, SystemC supports the notion of delta-cycles, which impose a partial order on parallel processes. This means that the execution is split into an *evaluate* and an *update* phase. In the first phase, concurrent processes are evaluated. This may include read and write accesses to so-called *primitive channels*, which store changes in temporary variables and do not update their channel state until the update phase. This ensures that although the processes are serialized, they all work on the same channel states (i. e., input data). A delta-cycle lasts an infinitesimal amount of time, and an arbitrary, finite number of delta-cycles may be executed at one point in simulation time. Note that the order in which processes are executed within a delta-cycle is not specified in [10], i. e., it is inherently *non-deterministic*.

2.2 BLAST

The software model checker BLAST (Berkely Lazy AbStraction verification Tool) [1] is an open source tool for the automatic verification of temporal logic properties of C programs. It was developed at the University of California, Berkeley. BLAST can be used both for program verification and for test case generation. The decision procedure used in BLAST is based on the paradigm of counter-example guided abstraction refinement (CEGAR) and lazy abstraction [7]. The CEGAR loop starts with a coarse abstraction of the program, and iteratively checks the abstract program against the specification. If there is a spurious counter-example in the abstract program, which is due to the imprecision of the abstraction, it is step-wise refined until the program is proved safe, or a non-spurious counter-example is found. Lazy abstraction is a technique that improves this iterative process by searching the abstract state space on the fly, and by only refining the coarse abstraction along the path of the spurious counter-example, leaving the abstraction in other parts unchanged. The use of CEGAR together with lazy abstraction makes BLAST strong when applied to data-intensive systems, in particular if strong abstractions can be found.

3 Related Work

There exist several approaches to provide a formal semantics for SystemC. Many of them rely on the transformation of SystemC designs into some sort of state machine, e.g. [6, 14, 15]. Habibi and Tahar [6] transform untimed SystemC models into equivalent state machines but do not maintain the structure of the underlying SystemC. Traulsen et al. [14] map SystemC to PROMELA, but only handle SystemC designs on an abstract level, do not model the SystemC scheduler and do not support primitive channels. Zhang et al. [15] introduce a formalism called SystemC waiting-state automata, which model SystemC designs at the delta-cycle level. They also do not model the scheduler and they do not consider complex interactions between processes. Other approaches use process algebras [13, 5], petri-nets [11] or a C representation [4, 3] for the verification of SystemC designs. The formal language SystemC$^{\mathbb{FL}}$ [13] is based on process algebras and defines the semantics of SystemC processes using structural operational semantics style deduction rules. It considers only simple communications, and no dynamic sensitivity or channels. Garavel et al. [5] translate SystemC/TLM into the process algebra LOTOS and import C Code into the LOTOS model using the verification toolbox CADP. They are able to support many SystemC and C++ constructs, but the transformation has to be done manually and they only support untimed SystemC designs. Karlsson et al. [11] use a petri-net based representation to verify SystemC designs. As interactions between subnets introduces additional subnets this approach produces a huge overhead. Cimatti et al. [4, 3] propose a transformation from SystemC into both sequential and threaded C programs. Their work is similar to our approach, and we adopt some of their ideas.However, their approach uses method inlining and thus does not keep the structure of a given design transparent to the designer, which makes

debugging hard. Furthermore, it is limited to a restricted set of data types and can not handle structs or arrays.

Besides Kroening et al. [12, 2], none of the related approaches can cope with pointers, arrays, and structs. In [12, 2], the authors propose a semantics for SystemC that is based on a labeled Kripke structure and automatically partition the design into a hardware and a software part to increase efficiency of verification. However, they abstract from hardware and do not consider timing or inter-process communication via sockets and channels, which makes it difficult to cope with deeply integrated hardware and software components.

In our own previous work, we have presented an approach for the transformation of SystemC designs into UPPAAL timed automata and their verification using the UPPAAL model checker [8, 9]. However, the UPPAAL model checker cannot handle data very well. In contrast, BLAST with its counter-example driven abstraction refinement can cope with data comparatively efficiently.

4 Transforming SystemC into Non-deterministic C

The main contribution of this paper is a transformation from SystemC into non-deterministic sequential C that keeps the structure of the design transparent to the designer. The resulting C program can be verified using the BLAST model checker. The main idea of the transformation is that we interpret the SystemC semantics in sequential C code, using the non-deterministic choice of the BLAST model checker to model the SystemC scheduler. To this end, we transform each SystemC process of a given design into a set of C methods, whose execution is controlled by C implementations of the (non-deterministic) scheduler, events, and sensitivity. Our transformation supports a large subset of SystemC programs, including static and dynamic sensitivity, time, pointers, arrays, and structs. In the following subsections, we first define this subset by a set of assumptions we have to impose on a given input design. Then, we explain how a SystemC design is represented as a C program using our interpretation of its semantics. Finally, we describe our implementations of methods, events, processes, the scheduler and the *request-update* mechanism of primitive channels.

4.1 Assumptions

We impose the following assumptions on a given input design:

1. No recursion is used.
2. No function pointers are used.
3. We assume that pointer arithmetic is used safely and that type safety of memory accesses is given.
4. We assume that no integer overflows are present in the system.
5. So far, we do not support any hardware data types.
6. So far, we do not support class inheritance.

The first four assumptions are directly derived from the subset of C programs that is supported by the BLAST model checker. Assumptions 5 and 6 could potentially be lifted in future work.

4.2 Representation of SystemC Designs in C

The main challenge for the transformation from SystemC into non-deterministic sequential C programs is to interpret the SystemC execution semantics and find an adequate representation of each SystemC language construct in the resulting C program. The SystemC execution semantics is defined by the interplay between the scheduler, processes, events, and primitive channels. In particular, the execution of processes is controlled by events and the scheduler. A process may be interrupted by a wait call, which results in the suspension of the process until it is triggered by an event or the expiration of a given timeout. To model this behavior in a C program, it is necessary to implement interrupt routines, which are able to save the current state of a process, suspend its execution and resume it at a later point of execution.

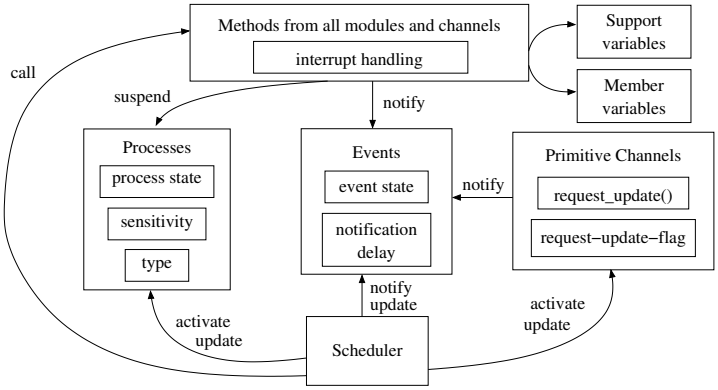

Fig. 1. SystemC Designs as C Programs

Figure 1 shows how we represent the elements of a SystemC design and their interactions in a BLAST-compatible C program. Methods from all modules and channels are translated into C methods that make use of member variables. They are extended with interrupt routines that handle process suspension and resumption. Processes are represented by a process state. Additionally, sensitivity lists are added for each process to handle static and dynamic sensitivity of processes, and we add a type tag to distinguish between method and thread processes. For each event, variables to store the event state and the point of time for delayed notifications are generated. To model the *request-update* semantics of primitive channels, we introduce a flag that indicates whether an update is requested for the current delta cycle and a function to set this flag.

The scheduler controls the event-triggered execution of a SystemC design. In our C program, it consists of multiple methods that read and write sensitivity lists, event and process states to determine processes that are executable and activates them by calling the corresponding methods. It also controls simulation time and calls update methods during the update phase.

```
int prod_produce(int c, int pid) {                                    1
...                                                                   2
int wait_time = 100;                                                  3
// timed notification of an event ev                                  4
if(event_state[ev] != DELTA && (event_state[ev] != TIME ||            5
                        event_time[ev] > time + wait_time)) {         6
    event_state[ev] = TIME;                                           7
    event_time[ev] = time + wait_time;                                8
}                                                                     9
// set dynamic sensitivity                                            10
dyn_sensitivity[pid] = ev;                                            11
...                                                                   12
```

Listing 1.1. Transformation of Event Notification

4.3 Design Transformation

The structure of a SystemC design is defined by modules that are connected via ports and channels. Each of the modules and channels contains local variables, events, methods and processes, and possibly other modules or channels. In contrast to SystemC, C does not support object-orientation. To cope with this, we flatten the hierarchy of a given SystemC design and translate all methods and variables into global methods and variables. We keep the structure of the original design transparent to the designer using prefixing. Port connections are implicitly modeled by replacing the call of a method through a port with a call to the channel method that is bound to this port.

As stated above, the SystemC execution semantics is defined by the interplay of the scheduler, events, and processes. The main challenge when modeling the SystemC scheduler in a sequential C program is to resolve concurrent processes into a sequential execution order. Following the approach of [4], we use a non-deterministic variable to execute processes in an arbitrary order. This ensures that the later verification of a given design considers all possible execution orders. As this approach is already described in [4], we focus on the transformation of events and processes in this section.

For illustration, we use a simple producer-consumer example, where a producer and a consumer communicate through a *first-in-first-out* (FIFO) buffer. Both the producer and the consumer have a *main_method*, which is bound to a process. Additionally, the producer has a method *produce*, which is called from the *main_method* and produces an item.

Event Transformation As in [4], we model events by two variables representing their state and the point of time where the event will occur next (if notified with a timed notification). The event state is one of DELTA, FIRED, TIMED, and NONE. To notify an event, the corresponding process updates the event state and possibly the event time accordingly. Immediate notifications can be modeled as shown in [4] by simply setting the event state to FIRED and setting processes that are sensitive to the event to runnable.

```
...                                                                    1
// put process to sleep                                                2
state[pid] = SLEEP;                                                    3
// save local variables                                                4
prod_produce_c[pid] = c;                                               5
prod_produce_wait_time[pid] = wait_time;                               6
// set return label and suspend                                        7
prod_produce_pc[pid] = 3;                                              8
return -1;                                                             9
// restore local variables at return label                             10
Label 3 :                                                              11
c = prod_produce_c[pid];                                               12
wait_time = prod_produce_wait_time[pid];                               13
...                                                                    14
}                                                                      15
```

Listing 1.2. Transformation of Method Suspension

An example of a timed notification of an event *ev* is shown in Lines 5 to 8 in Listing 1.2. The semantics of SystemC event notifications dictates that a timed notification overwrites pending timed notifications if the given timing delay expires earlier than the pending notification. A timed notification is discarded if a delta-delay notification is pending. We check these two conditions in Lines 5-6. The event sensitivity is set to dynamic in Line 11.

Process Suspension and Resumption To model event-triggered execution, we have to cope with suspension and resumption of processes. For example, in the producer-consumer design, the method *produce* is suspended after timed notification. In order to enable the suspension of processes in the middle of a C method, we do the following (similar to [4]): **1.** We save the values of local variables (incl. parameters) in global variables. **2.** We introduce a label and save it as return label of the current process. **3.** We suspend the method by using the return statement. **4.** At the return label, we restore local variables and continue execution. Unlike Cimatti et al. [4], who use method inlining to achieve exactly one method per process, we parameterize all C methods with a unique process ID to enable that multiple processes may enter the same method. The process ID (`pid`) is used to update the process state to SLEEP, to save and restore local variables, and to save the return label of the current process. The corresponding global variables are implemented as arrays with an entry for each process of the system. In future work, the size of the arrays may be reduced by checking how many processes may enter a given method.

An example for method suspension is shown in Listing 1.2. In Line 3, the process is put to sleep. To actually suspend method execution, local variables and the return label are saved in Line 5, 6 and 8. The method is left in Line 9. The process can resume execution at the point where it was suspended by jumping to the return label defined in Line 11. Then, local variables are restored and execution is continued.

```
void prod_main_method(int pid) {                                    1
  ...                                                               2
  // save local variables                                           3
  prod_main_method_c[pid] = c;                                      4
  // add label and restore variables                                5
  Label4 : c = prod_main_method_c[pid];                             6
  // function call                                                  7
  fifo_inst_write (c , process id );                                8
  // method could have been interrupted                             9
  if (state[pid] == SLEEP) {                                        10
    // save label and give up control                              11
    prod_main_method_pc[pid] = 4;                                   12
    return; }                                                       13
  ...                                                               14
}                                                                   15
```

Listing 1.3. Transformation of a Method Call

Method Call Transformation A further challenge when modeling process suspension that a process may be not necessarily in the method that is directly bound to the process, it may also be in any method of its call stack. In [4], this problem is solved using method inlining. This, however, has the drawback that it destroys the original structure of the SystemC design and thus makes debugging much harder. Furthermore, method inlining seriously increases the code size of the resulting C program. In our approach, we add return labels to method calls and save local variables and the return label before a method is called. If the method returns, we check whether the process was suspended during the execution of the method call. If it was suspended, we save the return label and suspend the outer method as well. For systems with many methods, this leads to a large number of return labels. However, the number of return labels for each method corresponds to the number of method calls plus one, and thus it is linear in the code size.

An example for a method call with process suspension is shown in Listing 1.3. Note that local variables have to be saved before the method call (Line 4), and also adding a return label and restoring of local variables have to be done before the method is called (Line 6) to cope with the case where the called method already was suspended and has to be called again. After the method call (Line 8), the check whether it was suspended is performed and, if necessary, the return label to execute the method call again is saved and the outer method is suspended as well (Line 10 to 13).

5 Experimental Results

We have implemented the transformation from SystemC to the input language of BLAST as described above in Java. To show the practical applicability of our approach and to compare it with our previous approach where SystemC designs are translated into UPPAAL timed automata [9], we have used an Anti-Slip

```
if(wheel_slip && !ASR_triggered)  { ERROR : goto ERROR; }
```

Listing 1.4. Example property

Regulation and Anti-Lock Braking System (ASR/ABS) The ABS/ASR system monitors the speed at each wheel and regulates the brake pressure in order to prevent wheel slip or lockup and improve the driver's control over the car. It consists of approximately 500 LOC (18 processes, 12 channels). The C program generated by our transformation engine comprises approximately 2400 LOC. All experiments were run on a machine with an Intel Pentium 3.4 GHz CPU and 4 GB main memory and averaged over 10 runs.

In our experiments, we checked that a wheel slip always triggers the anti-slip regulation (ASR) and that a wheel lock always triggers the anti-lock braking system (ABS). Both properties can be specified as reachability properties as shown in Listing 1.4.

The BLAST model checker automatically checks for reachability of the ERROR label, which in this case is unreachable. This proves that the ASR is always triggered if a wheel slip is detected. The verification times are shown in Table 1. The verification of the ASR took approximately 7 minutes, the verification of the ABS approximately 8 minutes. As shown in [9], model checking of the ASR/ABS system with our previous approach was only possible with bit-state hashing enabled, which uses a potentially unsafe abstraction of the state space. With our novel approach for the automatic verification of SystemC designs using the BLAST model checker we make counter-example guided abstraction refinement (CEGAR) available for SystemC designs and thus can handle such designs with reasonable effort.

Table 1. Verification times for ASR/ABS

	BLAST	**UPPAAL**	**UPPAAL** with bit-state hashing
ASR	420.72	*out of memory*	844.15 (maybe)
ABS	491.61	*out of memory*	555.56 (maybe)

6 Conclusion

In this paper, we have presented an approach for the automatic verification of SystemC designs using the BLAST model checker. Our main contribution is an interpretation of the SystemC semantics in a non-deterministic, sequential C program, which keeps the structure of the design transparent to the designer. Our approach supports many important SystemC constructs, including static and dynamic sensitivity, time, pointers, arrays, and structs. By providing an explicit definition of the SystemC scheduler, events, and processes, we soundly capture the SystemC semantics. At the same time, our approach enables us to

use the BLAST model checker for SystemC designs. The main advantage is that the BLAST model checker enables us to use counter-example guided abstraction refinement, which scales well for an important class of SystemC designs, namely asynchronous and mainly sequential SystemC designs with intensive data handling. We have demonstrated this advantage with an Anti-Slip Regulation and Anti-Lock Braking system, which could not be handled with previous approaches for the verification of SystemC designs. With our novel approach, we can verify properties of the ASR/ABS system in less than 10 minutes.

In future work, we plan to combine our previous work, where the UPPAAL model checker is used for verification, with our novel approach for the verification of SystemC designs using the BLAST model checker. We think that the former is better suited for systems or subsystems where time, concurrency and communication play the most important role while the latter is better suited for systems or subsystems where data handling predominates.

References

1. D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker BLAST: Applications to software engineering. *International Journal on Software Tools and Technology Transfer*, 2007.
2. N. Blanc, D. Kroening, and N. Sharygina. Scoot: A Tool for the Analysis of SystemC Models. In *TACAS*, LNCS 4963, pages 467–470. Springer, 2008.
3. A. Cimatti, A. Griggio, A. Micheli, I. Narasamdya, and M. Roveri. Kratos - A Software Model Checker for SystemC. In *CAV*, LNCS 6806. Springer, 2011.
4. A. Cimatti, A. Micheli, I. Narasamdya, and M. Roveri. Verifying SystemC: A software model checking approach. In *FMCAD*, pages 51 –59, 2010.
5. H. Garavel, C. Helmstetter, O. Ponsini, and W. Serwe. Verification of an industrial SystemC/TLM model using LOTOS and CADP. In *MEMOCODE*, 2009.
6. A. Habibi and S. Tahar. An Approach for the Verification of SystemC Designs Using AsmL. In *ATVA*, LNCS 3707, pages 69–83. Springer, 2005.
7. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages*, pages 58–70. ACM, 2002.
8. P. Herber, J. Fellmuth, and S. Glesner. Model Checking SystemC Designs Using Timed Automata. In *CODES+ISSS*, pages 131–136. ACM press, 2008.
9. P. Herber and S. Glesner. A HW/SW Co-Verification Framework for SystemC. *ACM Transactions on Embedded Computing Systems*, 2013.
10. IEEE Standards Association. IEEE Std. 1666–2011, Open SystemC Language Reference Manual. IEEE Press, 2011.
11. D. Karlsson, P. Eles, and Z. Peng. Formal verification of SystemC Designs using a Petri-Net based Representation. In *DATE*, pages 1228–1233. IEEE Press, 2006.
12. D. Kroening and N. Sharygina. Formal Verification of SystemC by Automatic Hardware/Software Partitioning. In *MEMOCODE*, pages 101–110. IEEE, 2005.
13. K. L. Man. An Overview of SystemCFL. In *Research in Microelectronics and Electronics*, volume 1, pages 145– 148, 2005.
14. C. Traulsen, J. Cornet, M. Moy, and F. Maraninchi:. A SystemC/TLM semantics in Promela and its possible applications. In *SPIN*, LNCS 4595. Springer, 2007.
15. Y. Zhang, F. Vedrine, and B. Monsuez. SystemC Waiting-State Automata. In *International Workshop on Verification and Evaluation of Computer and Communication Systems*, 2007.