Bit-Precise Formal Verification for SystemC Using Satisfiability Modulo Theories Solving

Lydia Jass, Paula Herber

Bit-Precise Formal Verification for SystemC using Satisfiability Modulo Theories Solving

Lydia Jaß and Paula Herber

Technische Universität Berlin, Germany
{lydia.jass,paula.herber}@tu-berlin.de

Abstract. Hardware/software codesigns are often modeled with the system level design language SystemC. Especially for safety critical applications, it is crucial to *guarantee* that such a design meets its requirement. In this paper, we present an approach to formally verify SystemC designs using the UCLID satisfiability modulo theories (SMT) solver. UCLID supports finite precision bitvector arithmetics. Thus, we can handle SystemC designs on a bit-precise level, which enables us to formally verify deeply integrated hardware/software systems that comprise detailed hardware models. At the same time, we exploit UCLID's ability to handle symbolic variables and use k-inductive invariant checking for SystemC designs. With this inductive approach, we can counteract the state space explosion problem, which model checking approaches suffer from. We demonstrate the practical applicability of our approach with a SystemC design that comprises a bit- and cycle-accurate model of a UART and software that reads data from the UART.

1 Introduction

Embedded systems are ubiquitous in today's everyday life, and they are often used in safety-critical applications, e.g. in airplanes or cars. A failure of such a system can lead to high financial losses and even human injuries or deaths. This makes it crucial to verify their correctness under all circumstances.

One of the main challenges for embedded systems verification is that the systems usually consist of deeply integrated hardware and software components. There already exists a large variety of validation and verification techniques for integrated HW/SW systems. However, the validation techniques are mostly non-systematic and incomplete, such as simulation and testing. These techniques cover neither the whole design nor all possible input scenarios. Opposed to that, formal methods have the advantage of covering all possible input scenarios and all possible behaviors of a given system. However, most of the existing formal verification techniques for hardware/software codesigns are either tailored to hardware or to software verification and can not cope well with designs that contain both bit-precise hardware models and high-level software.

In this paper, we present an approach to overcome this problem by using the UCLID verification system. The UCLID system is used to specify and verify systems modeled at the term level, and thus provides adequate abstractions for

the representation of high-level software. At the same time, it supports the theories of bitvector arithmetics and of arrays. As underlying verification technique, UCLID uses a powerful Satisfiability Modulo Theory (SMT) solver supporting both eager and lazy SMT solving and the constructed formulas can be checked with any state-of-the-art SAT solver. Together, this makes UCLID a powerful tool for (bit-precise) system level design verification.

Our main contribution is a fully-automatic transformation of digital HW/SW co-designs that are modeled in SystemC into the UCLID specification language. The transformation enables us to apply the UCLID SMT solver to SystemC designs and thus to prove important properties like reliable safety and timing behavior. UCLID has the potential to cover most of the expressiveness of SystemC, including discrete time, static and dyanamic sensitivity, inter-process communication and bit-vector arithmetics. Our representation of SystemC designs in UCLID and UCLID's symbolic simulation mechanism enable scalable verification using k-inductive invariant checking. By using inductive verification, we avoid the state space explosion problem that model checking approaches typically suffer from. We demonstrate the scalability and practical applicability of our approach with two case studies. The first is a simple producer-consumer example, where we use varying buffer sizes. The second case study is a typical industrial HW/SW codesign, namely a bit- and cycle-accurate model of a UART together with a software component that reads data from the UART.

The paper is structured as follows: In Sec. 2, we briefly introduce SystemC, UCLID, and k-inductive verification. In Sec. 3, we discuss related work. We present our transformation from SystemC to UCLID in Sec. 4. We discuss experimental results in Sec. 5 and conclude in Sec. 6.

2 Preliminaries

2.1 SystemC

SystemC is a system-level design language and a framework for HW/SW cosimulation. The semantics of SystemC is informally defined in an IEEE standard [11]. It is implemented as a C++ class library, which provides language elements for the description of hardware and software, and allows for modeling of both hardware and software components on various levels of abstraction. It also features an event-driven simulation kernel, which enables the simulation of the design. A SystemC design consists of a set of communicating processes, triggered by events and interacting through channels. Modules and channels represent structural information. SystemC also introduces an integer-valued time model with arbitrary time resolution. Listing 1.1 shows an excerpt of a SystemC *producer* module that writes to a FIFO buffer. The *produce* method (which is executed within an *SC_THREAD* process) contains an infinite loop where the producer writes a value between 0 and 32 to the fifo port at every clock cycle.

The execution of SystemC designs is controlled by the SystemC scheduler. Like typical hardware description languages, SystemC supports the notion of

delta-cycles, which impose a partial order on parallel processes. Note that the order in which processes are executed within a delta-cycle is not specified in [11], i. e., it is inherently *non-deterministic*.

```
1  SC_MODULE(producer) {
2      ...
3      sc_port<myfifo_if> fifo;
4      void produce(void) {
5        int c = 0;
6        while(true) {
7          wait();
8          c = (c + 1) % 32;
9          fifo->write(c);
10     } }
11 };
```

Listing 1.1. A SystemC Module

2.2 UCLID

UCLID is a verification system developed in a joint project by Carnegie Mellon University and University of California, Berkeley [13]. It incorporates a decision procedure to verify (possibly infinite) state systems. The specification language supports uninterpreted functions, bit-vector arithmetic and lambda expressions. UCLID can handle symbolic simulation, which allows a design to be verified for an arbitrary start state and thus enables an inductive verification approach.

A UCLID module consists of inputs, variables, constants, macros, and assign expressions. The assign expressions define the state variables and the transition relation of the underlying labeled transition system. UCLID interprets the model together with the property to be verified as one formula and supports eager and lazy Satisfiability Modulo Theories (SMT) solving.

2.3 K-Inductive Invariant Verification

For k-inductive invariant checking [16], two models are needed. One explicit model representing the system from its initial state, and one symbolic model that represents the system in an arbitrary state. Desired properties are expressed as a predicate *P(x)*, which determines whether a requirement P holds in simulation step x of a given model. k-inductive invariant checking is done in two steps:

1. **Base case:** Simulate the explicit model k steps from its initial state and check $P(0) \wedge ... \wedge P(k)$.
2. **Induction:** Symbolically simulate the symbolic model from an arbitrary initial state for *k+1* steps. Then check $P(0) \wedge ... \wedge P(k) \implies P(k+1)$.

If there exists a k so that both base cases can be shown, then the property under verification holds in all reachable system states. If the property under verification does not hold, the system is unsafe and we get a counter example. Note that the counter example may be spurious if k is too small.

3 Related Work

There exist several approaches to the automated formal verification of SystemC designs. For example, in [8], the authors propose program transformations from SystemC into state machines, but they ignore time, the transformation is performed manually, and hardware data types are not explicitly considered. Karlsson et al. [12] verify SystemC designs using a petri-net based representation and the PRES+ model checker. However, the petri-net based approach introduces a huge overhead because interactions can only be modeled by introducing additional subnets. As it is based on model checking, the approach also suffers from the state space explosion problem. Bit-precise hardware data types are not explicitly considered. In [10, 14], an approach for the formal verification of SystemC designs using the model checker UPPAAL is presented. A large subset of SystemC is supported. However, UPPAAL does not support inductive verification techniques and thus suffers from the state space explosion problem. Furthermore, it is not well-suited to support hardware data types.

In [7], bounded model checking is used on untimed SystemC TLM designs. They use k-inductive invariant checking using CBMC [4] and Boolector [15] as underlying SMT solver. This work is very close to our approach. The main idea is to transform a given SystemC TLM design into a sequential C program and perform loop unwinding to achieve a complete model. However, compared to our approach, they only support a small subset of untimed SystemC TLM designs and disregard time and complex process interactions. Furthermore, bit-precise hardware data types are not explicitly considered.

In [6, 9], an encoding from SystemC into the verification toolbox CADP is proposed. This approach is based on a manual definition of callback functions, which are then used to natively execute SystemC/C++ code in the CADP verification system. Still, the transformation has to be done manually. Furthermore, bit-precise hardware data types are not explicitly considered. In [3], Cimatti et al. generate three different verification models from a given SystemC design, each tailored to a specific aspect of the SystemC semantics on different levels of abstraction, and use software model checking techniques. While this approach is capable of handling the most important SystemC constructs including time and communication, it can not handle bit-precise hardware data types. Furthermore, by using model checking techniques, it also suffers from the state space explosion problem. In [5], the authors present an approach for the verification of SystemC designs using the software model checker SPIN. Again, by using a software model checker, they suffer from the state space explosion problem.

Note that there also exist some approaches to automated formal verification of other system level design languages. For example, in [2], the authors present an approach for formal deadlock analysis of SpecC models using SMT. However, they only consider the timing relations in a given design by formulating assertions over time stamps, which are assigned to executable code. This is sufficient for a deadlock analysis but does not allow for checking of other functional or non-functional properties.

4 Transformation from SystemC to UCLID

The key idea of our approach for the bit-precise verification of SystemC designs using satisfiability modulo theories solving is to transform a given SystemC design into a semantically equivalent UCLID specification. Our transformation preserves the (informally defined) bit-precise semantics of a given SystemC designs. The main challenges are to preserve the sequential simulation semantics of SystemC in a synchronous target language, i. e., to model the non-deterministic execution semantics of the SystemC scheduler, and to respect the bit-precise semantics of all data operations. The basic idea of our transformation is to translate all SystemC processes into UCLID state machines, and to control the execution of these processes by modeling the SystemC scheduler and SystemC events as UCLID state machines. To capture the simulation semantics of SystemC, our UCLID model of the scheduler interprets the operational sequences of the SystemC kernel, which manages process scheduling and channel updates.

The transformation result is a UCLID interpretation of the semantics of the given SystemC design. This can then be automatically verified using the UCLID verification system. The main advantage of our approach is that the formal semantics we define for SystemC by translating it into the formal specification language of UCLID is bit-precise, and that the underlying verification engine is based on SMT solving and thus enables inductive proofs.

4.1 Assumptions

The following assumptions on a SystemC design define our supported subset.

1. No dynamic variable or process creation.
2. No recursion.
3. No inheritance nor pointers, no side effects.
4. No external code or library calls.
5. For division and modulo operations the divisor is an integer power of 2.
6. So far, we only support boolean and integer variables with fixed bit width, and arrays and structs thereof.

If all of these assumptions are fulfilled, we can transform a given SystemC design automatically into a semantics-preserving UCLID representation and verify it using the UCLID verification system.

4.2 Representation of SystemC Designs in UCLID

Our representation of a SystemC design in UCLID interprets the simulation semantics of SystemC. This means that not only all the SystemC modules of the design are represented in UCLID, but also the scheduler, processes, events and primitve channels, which altogether define the execution semantics.

SystemC modules contain processes, events and member variables. Module ports are connected to channels, which provide communication methods between

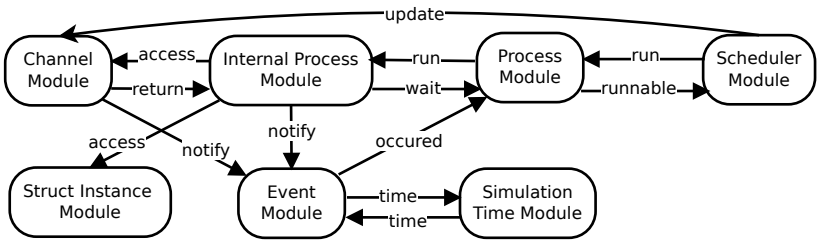

Fig. 1. Structure of a SystemC Design in UCLID

modules. Before simulation, the SystemC kernel creates the module hierarchy in its elaboration phase and performs instantiation and binding. In this phase, all module, channel, and process objects are created and bound together. Because UCLID does not support dynamic module creation, we recreate the SystemC design after elaboration. This means we create UCLID modules for all module and channel instances and their connections. As a preprocessing step for our transformation from SystemC to UCLID, we flatten the design. The hierarchical structure is kept transparent in the UCLID specification using prefixes. To capture the state of a given SystemC design, we use the following state variables: (1) All local and global variables, including all module and all channel variables, (2) the state of each process (including its current program counter), (3) the state of each event, (4) the state of the SystemC scheduler. The structure of our representation of SystemC designs in UCLID is shown in Fig. 1. To capture the static part of a design, we create modules for each channel and module (or, more general, struct) instance. The communication methods provided by a channel are placed within the corresponding UCLID module that represents the channel instance. To capture the simulation semantics, we create modules for the scheduler and the simulation time, and for all processes and events of a given design. We distinguish between an *internal process module*, which keeps track of the program counter and performs event notifications and channel accesses, and a *process module*, which determines the state of the process, and reacts to events to implement static and dynamic sensitivity.

Scheduler and Simulation Time We have defined two separate UCLID modules for the scheduler and simulation time. The scheduler module non-deterministically chooses the next runnable process and defines the phase the system is in (initialize, evaluate, update, or advance time). The simulation time module manages the advancement of time. In the advance time phase, the simulation time module advances time to the earliest pending timed notification.

Processes Processes are SystemC's units of execution. For each process, we introduce two modules. The internal process module *ipm* keeps track of the program counter, notifies events and accesses channels. The process module *pm* determines the state the process is in, and reacts to events to implement static

and dynamic sensitivity. When the scheduler chooses a process to be running, its process module changes its state to running. This triggers the internal process module, which runs until it reaches a wait statement or the process terminates.

An excerpt of the internal process module *ipm* is shown in Listing 1.2. It realizes a state machine that keeps track of the program counter and evaluates control flow conditions. The internal state *istate* is first set to *initialized*. As soon as the state of the process module *pm* is set to running, *istate* is set to the first program counter label *line9_while_loop* where a while loop is entered. Then, *istate* is set to the label *line11_call_wait*. Next, the process is suspended and waits to become runnable again. Function calls are realized by a similar mechanism. For example, the *write* method of the channel module *fifo* is started if the internal state is set to *line13_call_write*. It returns control to the produce process by setting its program counter to *fifo_done*.

```
1  ASSIGN
2  init[istate] := not_initialized;
3  next[istate] := case
4      ((istate = not_initialized) & (scheduler.state = initialize)) : initialized;
5      ((istate = initialized) & (pm.state = running)) : line9_while_loop;
6      ((istate = line9_while_loop) & true) : line11_call_wait;
7      ((istate = line11_call_wait)) : line11_wait_return;
8      ((istate = line11_wait_return) & (pm.state = running)) : line13_call_write;
9      ((istate = line13_call_write)) : line13_wait_for_write;
10     ((istate = line13_wait_for_write) & (fifo.write = fifo_done)) : line9_while_loop;
11     ((istate = line9_while_loop) & ~(true)) : done;
12     default : istate;
13 esac;
```

Listing 1.2. Excerpt of the Internal Process Module *ipm*

Note that we did not include a program counter label for each line of code in the *istate* state machine. Instead, we reduced the set of program counter labels to represent the necessary atomic blocks. An atomic block comprises a sequence of states in the control flow without branches or process suspension. As SystemC uses a cooperative scheduler, atomic blocks can never be interrupted. Thus, it is possible to abstract from some intermediate steps and to combine multiple sequential actions in one (synchronous) simulation step as long as we ensure that no data race may occur. So far, we just exclude cases where a potential data race may occur from this optimization. The benefit of the reduction is a smaller UCLID model while preserving the execution semantics of SystemC.

An excerpt of the process module *pm* is shown in Listing 1.3. It implements the process state, which may be one of *process_initialized, running, runnable, process_done, wait_t, wait_e, or wait_s* (the latter to wait for a given time, an event or for the sensitivity list, respectively). Note that *pm* makes use of the current state of *ipm*, the current process selected by the scheduler, and the current state of the FIFO channel. The latter is necessary because the process might be suspended by a wait call that occurs within the FIFO channel.

```
1  DEFINE
2  sensitivity_list_occurred := case clk_edge_event.occurred : true;
3                                   default : false; esac;
4  ASSIGN
5  init[state] := process_no_state;
6  next[state] := case
7   (ipm.istate = initialized) : process_initialized;
8   (state = process_initialized) : runnable;
9   ((scheduler.next = produce) & (scheduler.current = none)) : running;
10  ((state = wait_s) & sensitivity_list_occurred) : runnable;
11  (ipm.istate = done) : process_done;
12  (ipm.istate = line11_call_wait) : wait_s;
13  (ipm.istate = line13_wait_for_write & fifo.write = line31_call_wait) : wait_e;
14  (ipm.istate = line13_wait_for_write & r_event.occurred & state = wait_e) : runnable;
15   default : state;
16 esac;
```

Listing 1.3. Excerpt of the Process Module *pm*

Events We create one UCLID module for each event. Additionally, we also create timeout event modules for processes that suspend themselves by calling a timed wait. Processes may notify events immediately, delta-delayed or timed. If a process performs an immediate notification of an event, the event immediately occurs and all processes that are sensitive to the event react by changing their states. For a delta-delayed notification, the notification is delayed until the next delta-cycle starts. If a process performs a timed notification, the event adopts the time value and waits until the simulation time is equal to the target time. If a process calls the wait function without any argument, its process module waits for one of the events from the processes sensitivity list to occur. New notifications overwrite pending notifications if they expire at an earlier target time.

An excerpt of a UCLID event module is shown in Listing 1.4. Initially, the event state *estate* is set to *no_notification*. Then, it reacts to all processes that might notify the event and sets the state to *immediate*, *delta* or *timed* accordingly. The event state is reset if *occurred* becomes true. This happens whenever an immediate notification occurs, if we have a pending delta-delayed notification and the scheduler starts a new delta cycle, or if we have a pending timed notification and the simulation time becomes equal to the target time.

```
1  DEFINE
2  occurred := case ((estate = immediate) | ((estate = delta) & scheduler.new_delta)
3                                  | ((estate = timed) & (t = sim_time.time))) : true;
4                        default : false; esac;
5  ASSIGN
6  init[estate] := no_notification;
7  next[estate] := case
8     occurred : no_notification; (* reset the event state *)
```

```
9     process.notify_event_immediate : immediate;
10    ((estate != immediate) & process.notify_event_delta) : delta;
11    ((estate != immediate) & (estate != delta) & process.notify_event_timed) : timed;
12    default : estate;
13 esac;
14 init[t] := ...
```

Listing 1.4. Event Module

Channels For each channel, we create a module that contains all channel variables and all communication methods for all process modules that might call those methods. If a channel method waits for an event to occur, the caller process module changes its state to also wait for the event. Primitive channels implement the request-update semantics. This means that they do not change the state of the channel directly. Instead, they request the channel to be updated during the update phase. For this purpose, primitive channels have a dedicated *update* method, which is executed in the update phase after all evaluations are finished. In UCLID, we represent this mechanism by setting a *request_update* variable in the channel module that activates an *update* state machine.

Variables Beside process and event states, the state of a SystemC design comprises local and global variables. Using the program counter labels described above, we can decide for each variable at what points in the program it is manipulated. Using this information, we can construct one UCLID state machine for each variable of a given SystemC design. An example for a state machine that models the local variable c of our running example is shown in Listing 1.5.

Note that in UCLID, a declaration of a variable does not ensure that the given number of bits is used for its representation. Instead, UCLID always uses as few bits as possible. To ensure that the bit-precise semantics of SystemC is preserved, we use the correct bitwidth for all arithmetic and logic operations and we cast all expressions into the left hand side types within assignments.

So far, we support boolean- and int-typed variables, sc_int and sc_uint, arrays, and structs. We represent boolean types by UCLID's TRUTH type. The types int, sc_int and sc_uint are represented as bitvectors using UCLID's BITVEC-type with the specified length respectively. Arrays are modeled using lambda-expressions, as described in the UCLID documentation [1].

```
1 ASSIGN
2 init[c] := (0 +_32 0);
3 next[c] := case
4    (istate = line9_while_loop) : 0;
5    (istate = line13_call_write) : (c +_32 1);
6    default : c;
7 esac;
```

Listing 1.5. Variable Handling

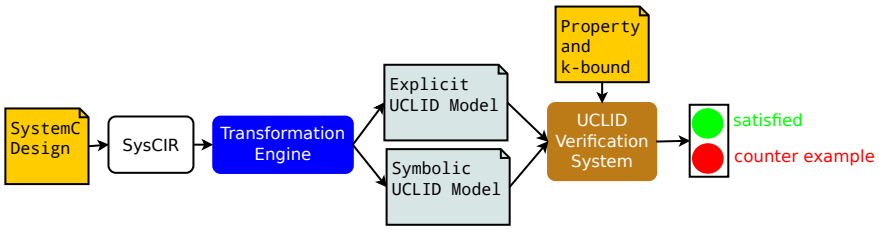

Fig. 2. Framework for Verifying SystemC Designs with UCLID

4.3 Verification of SystemC Designs using UCLID

With our representation of SystemC designs within UCLID as described above, we can automatically transform a given SystemC design into a UCLID specification. Then, it can symbolically be simulated and verified in UCLID using k-inductive invariant checking [16]. To adapt k-inductive invariant checking with UCLID, we create an explicit UCLID model and check the base case and a symbolic model for the induction, as described in Sec. 2.3. The explicit model starts in the same state as the SystemC design. For the symbolic model, we set all state variables to symbolic constants (for truth types, bitvectors and bitvector functions) or non-deterministic choice variables (for enum variables) respectively. If UCLID returns a counter example, it might be spurious, i.e. it might not be reachable by the original system. If UCLID returns that the induction step is valid, we can conclude that the system never violates the requirement and it is safe. There is no straightforward way to determine the value of k needed to prove or disprove a property. Thus, we incrementally increase k until we reach a predefined maximum value of k or the counter example is not spurious.

5 Evaluation

We have implemented our transformation from SystemC to UCLID in Java. The resulting framework is shown in Fig. 2. A given SystemC design is preprocessed and translated into our SystemC intermediate representation SysCIR. There, we flatten the design and resolve all port and channel connections. Our novel transformation engine then generates an explicit and a symbolic UCLID model. Those are then used for k-inductive invariant checking as described above.

To evaluate our approach, we use two case studies and compare the verification times with those achieved with UPPAAL [10, 14]. In the first case study, two producers and one consumer communicate over a FIFO buffer (3 processes, 1 channel). The second case study is a more complex UART design that consists of a bit- and cycle-accurate UART model and software that reads data from the UART (7 processes, 9 channels). We ensure that we cover all possible input scenarios by modeling input data using selections in UPPAAL and symbolic constants in UCLID. Note that the UART case study could not be handled with the approach presented in [7]. All experiments were performed on a 64 bit Linux

Table 1. Verification Times in [hh:]mm:ss

	Producer-Consumer				UART system				
buffer size/bitwidth	10	50	100	1000	4	8	16	24	32
BMC	< 1	< 1	< 1	< 1	10:30	10:25	10:03	10:07	10:03
k-induction	0:06	0:06	0:06	0:06	7:22:00	7:03:00	7:11:00	6:40:00	7:11:32
UCLID Total	0:06	0:06	0:06	0:06	7:32:30	7:13:25	7:21:03	6:50:07	7:21:35
UPPAAL Total	0:02	0:02	0:09	3:00:51	2:42:40	x	x	x	x

system with an Intel Core i7-4770 with 3.2 GHz and 32 GB main memory. All designs are transformed into UCLID respectively UPPAAL in less than a second.

We have shown that the producer-consumer example does not cause a buffer over- or underflow with our k-inductive invariant checking approach. The requirement we check is $P(i) := \mathit{fifo.n} \leq BS \wedge \mathit{fifo.n} \geq 0$. To verify this, k must at least be *10*. This corresponds to the maximum number of steps until the desired property is restored from an arbitrary start state. To evaluate the scalability of our approach, we have varied the buffer size from 10 to 1000. As Table 1 shows, the time it takes UCLID to verify the model with growing buffer capacities stays nearly constant, while the verification time in UPPAAL increases exponentially.

For the UART system, we verify that the software correctly reads all data that is sent over the UART. The requirement we check is $P(i) := \mathit{sw.error\_cnt} \leq 0$, where *error_cnt* is used in the software component *sw* to mark if the values differ. To verify the UART system, k must at least be *200*. We evaluated the scalability by varying the bitwidth of the transmitted data from 4 bit to 32 bit. Table 1 shows the runtimes of both bounded model checking and the induction step. Although the computational effort is considerable, we achieve a complete proof for all possible input scenarios, and the verification time is similar for varying bitwidths. UPPAAL is able to verify the 4-bit system in less time, but exceeds resources for all larger bitwidths[1].

6 Conclusion

In this paper, we have presented an automatic transformation from SystemC to the formal verification system UCLID. Our transformation is able to cover a large subset of SystemC including static and dynamic sensitivity, time, primitive channels, and bit-precise hardware data types. To capture the semantics of a given SystemC design, we translate all variables and processes into UCLID state machines, and we provide predefined state machines that model the execution semantics of SystemC and the event notification mechanism. The structure of the original design is kept transparent by using prefixing. This eases comprehensibility of counter examples. With the result of our transformation, we can use UCLID's powerful verification mechanisms and underlying SMT solver to verify a given SystemC design using k-inductive invariant checking.

[1] UPPAAL did not return results after 48 hours while using 32 GB main memory.

To demonstrate the practical applicability of our approach, we have used a simple producer-consumer example taken from the SystemC reference implementation and an industrial UART system. We have shown that our approach scales well for increasing data ranges and bit widths, which is typically not the case for model checking based approaches. With the UART system, we have also shown the applicability of our approach to (small) industrial applications.

In future work, we plan to optimize our transformation by allowing parallel executions whenever we can safely assume that no data race may occur. We are confident that in doing so, we can significantly reduce the necessary k for k induction, which in turn will significantly reduce the verification effort.

References

1. Brady, B., Seshia, S., Lahiri, S., Bryant, R.: A User's Guide to UCLID (2008)
2. Chang, C.W., Dömer, R.: Formal Deadlock Analysis of SpecC Models Using Satisfiability Modulo Theories. In: Embedded Systems: Design, Analysis and Verification, IFIP AICT, vol. 403, pp. 116–127. Springer (2013)
3. Cimatti, A., Narasamdya, I., Roveri, M.: Software model checking systemc. IEEE Trans. on CAD of Integrated Circuits and Systems 32(5), 774–787 (2013)
4. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS. LNCS, vol. 2988, pp. 168–176. Springer (2004)
5. Elshuber, M., Kandl, S., Puschner, P.P., Choppy, C., Sun, J.: Improving system-level verification of systemc models with spin. In: FSFMA. pp. 74–79 (2013)
6. Garavel, H., Helmstetter, C., Ponsini, O., Serwe, W.: Verification of an industrial SystemC/TLM model using LOTOS and CADP. In: MEMOCODE. pp. 46 –55. IEEE Computer Society (2009)
7. Große, D., Le, H.M., Drechsler, R.: Proving Transaction and System-level Properties of Untimed SystemC TLM Designs. In: MEMOCODE. pp. 113 – 122. IEEE Computer Society (2010)
8. Habibi, A., Moinudeen, H., Tahar, S.: Generating Finite State Machines from SystemC. In: Design, Automation and Test in Europe. pp. 76–81. IEEE (2006)
9. Helmstetter, C.: TLM.open: a SystemC/TLM Frontend for the CADP Verification Toolbox. Leibniz Transactions on Embedded Systems 1(1) (2014)
10. Herber, P., Fellmuth, J., Glesner, S.: Model Checking SystemC Designs Using Timed Automata. In: CODES+ISSS. pp. 131–136. ACM press (2008)
11. IEEE Standards Association: IEEE Std. 1666–2011, Open SystemC Language Reference Manual. IEEE Press (2011)
12. Karlsson, D., Eles, P., Peng, Z.: Formal verification of SystemC Designs using a Petri-Net based Representation. In: DATE. pp. 1228–1233. IEEE Press (2006)
13. Lahiri, S.K., Seshia, S.A.: The UCLID Decision Procedure. In: CAV. pp. 475–478. LNCS 3114 (2004)
14. Pockrandt, M., Herber, P., Kloes, V., Glesner, S.: Model checking memory-related properties of hardware/software co-designs. In: IESS. pp. 92–103. IFIP AICT 403, Springer (2013)
15. R. Brummayer, A.B.: Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In: TACAS. LNCS, vol. 5505. Springer (2009)
16. Sheeran, M., Singh, S., Stalmarck, G.: Checking safety properties using induction and a SAT-solver. In: FMCAD, LNCS, vol. 1954 (2000)