Proving Transaction and System-level Properties of Untimed SystemC TLM Designs*

Daniel Große[1,2] Hoang M. Le[1] Rolf Drechsler[1]
[1]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
[2]Institute of Computer Science, Albert-Ludwigs-University, 79110 Freiburg im Breisgau, Germany
{grosse,hle,drechsle}@informatik.uni-bremen.de

***Abstract*—Electronic System Level (ESL) design manages the enormous complexity of todays systems by using abstract models. In this context Transaction Level Modeling (TLM) is state-of-the-art for describing complex communication without all the details. As ESL language, SystemC has become the de facto standard. Since the SystemC TLM models are used for early software development and as reference for hardware implementation their correct functional behavior is crucial. Admittedly, the best possible verification quality can be achieved with formal approaches. However, formal verification of TLM models is a hard task. Existing methods basically consider local properties or have extremely high run-time. In contrast, the approach proposed in this paper can verify "true" TLM properties, i.e. major TLM behavior like for instance the effect of a transaction and that the transaction is only started after a certain event can be proven.**

Our approach works as follows: After a fully automatic SystemC-to-C transformation, the TLM property is mapped to monitoring logic using C assertions and finite state machines. To detect a violation of the property the approach uses a BMC-based formulation over the outermost loop of the SystemC scheduler. In addition, we improve this verification method significantly by employing induction on the C model forming a complete and efficient approach. As shown by experiments state-of-the-art proof techniques allow proving important non-trivial behavior of SystemC TLM designs.

I. Introduction

System-on-Chips (SoCs) combine hardware and embedded software on a single chip. Developing such complex systems within todays time-to-market constraints requires to build abstract models for architectural exploration and early software development. This procedure has been systematized resulting in the so-called *Electronic System Level* (ESL) design [1]. For ESL design *SystemC* [2], [3] has become the de facto standard. In particular, the concept of *Transaction Level Modeling* (TLM), which enables the description of communication in terms of abstract operations (transactions), improved the success of SystemC. The simulation of SystemC TLM models is orders of magnitudes faster in comparison to RTL. Furthermore, TLM allows interoperability between models from different IP vendors.

SystemC is a C++ class library and provides modules, ports, interfaces and channels as the fundamental modeling components whereas the functionality is described by processes. In addition, the SystemC library also includes an event-driven simulation kernel. Essentially, the simulation kernel executes the processes non-preemptive and manages their parallel execution by using delta-cycles.

Clearly, an abstract SystemC TLM model provides the first formalization of the design specification. This first TLM model is usually untimed and will be successively refined by adding timing information to a timed TLM model, which in turn is refined down to RTL. Hence, it is very important revealing potential bugs already at TLM. However, this functional verification task is difficult [4]. Methods commonly applied at TLM rely on simulation (see e.g. [5], [6], [7]) and therefore cannot guarantee the functional correctness. The existing formal verification approaches for SystemC TLM designs mainly check properties local to processes or have extremely high run-time (more details are discussed in the related work section). Hence, they cannot be used to verify major TLM behavior such as the start of a transaction after a certain event. In contrast, the approach proposed in this paper makes the following contributions:

- Verification of "true" TLM properties
  In addition to simple safety properties the user can check the effect of transactions and the causal dependency between events and transactions. The *Property Specification Language* (PSL) is used to formulate a property.
- Adjustment of temporal resolution
  The approach allows to specify the sampling rate of the temporal operators, e.g. the user can focus on certain events or start/end of specific transactions.
- Automated verification method
  The approach performs a fully automatic SystemC-to-C transformation. Then, monitoring logic for the property is automatically embedded into the C model. This monitoring logic uses C assertions and *Finite State Machines* (FSMs). To verify the property, the verification method of *Bounded Model Checking* (BMC) is employed on the C model.
- Efficiency and completeness
  An induction-based verification method working on the level of C is proposed for the generated models, making the approach complete and much more efficient.

For different SystemC TLM designs we report the verification of properties describing important behavior at TLM

*This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project SANITAS under contract no. 01M3088.

which has not been possible before. Moreover, the experiments demonstrate that complete proofs can be carried out efficiently using induction.

This paper is structured as follows: In Section II related work is discussed. The preliminaries are provided in Section III. Section IV introduces the TLM property checking approach. First, the model generation from a SystemC TLM design is explained. Then, the property language and the creation of respective monitors are introduced. In the last part of this section the BMC-based verification technique is presented. Section V describes the induction-based verification method for the transformed models. The experimental evaluation is given in Section VI. Finally, the paper is summarized and ideas for future work are outlined.

II. Related Work

One of the first formal approaches for SystemC TLM verification has been introduced in [8]. However, the design entry of this method is UML and only during the construction of the derived FSM some properties can be checked. Another approach has been proposed in [9]. A formal model can be extracted in terms of communicating state machines and can be translated into an input language for several verification tools. Simple properties on very small designs have been verified with this approach. The authors of [10] translate a SystemC design into Petri nets and then apply CTL model checking. However, the resulting Petri nets become very large even for small SystemC descriptions as the experiments have shown. In [11] a technique has been presented that allows CTL model checking for SystemC TLM designs, but one has to follow a very restricted modeling style while specifying the system. The approach of [12] translates a SystemC TLM design into Promela. The Promela model is then checked by the model checker SPIN. The translation is entirely manual and properties related to events and transactions are not considered. A translation of untimed SystemC TLM to the process algebra LOTOS is proposed in [13]. However, the focus of the work is to compare two possible LOTOS encodings, property checking is not discussed. In [14] an approach mapping SystemC designs into UPPAAL timed automata has been proposed. It differs from our approach in particular regarding the expressiveness of the properties. The work in [15] presented an approach combining static and dynamic *Partial Order Reduction* (POR) techniques to detect deadlocks and simple safety property violations. [16] proposed a static POR technique for state space exploration of SystemC designs using model checking, but property checking was not considered. The limitations of both approaches are that representative inputs need to be provided and the absence of corner-case errors cannot be proven.

Recently, a fundamental work has been published [17] which defines a trace semantic for SystemC covering also abstract models. Furthermore, a PSL [18] oriented language has been introduced which additionally includes new primitives to allow expressing software aspects like for example pre- or post-conditions. We use the introduced PSL primitives in this work. The respective details are discussed in Section IV.

III. Preliminaries

A. Bounded Model Checking and Induction

BMC has been introduced by Biere et al. in [19] and gained popularity very fast. For a LTL formula φ the basic idea of BMC is to search for counter-examples to φ in executions of the system whose length is bounded by k time steps. More formally, this can be expressed as:

$$BMC^k = I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \neg\varphi^k,$$

where $I(s_0)$ denotes the predicate for the initial states, T denotes the transition relation and $\neg\varphi^k$ constraints that the property φ is violated by an execution of length k. In case of simple safety properties of the form AGp where p is a propositional formula, the violation of the property reduces to $\bigvee_{i=0}^{k} \neg p_i$, where p_i is the propositional formula p at time step i. The overall problem formulation is then transformed into an instance of SAT. If this instance is satisfiable a counter-example of length k has been found. Usually, BMC is applied by iteratively increasing k until a counter-example for the property has been found or the resources are exceeded. One of the possibilities to make BMC complete, i.e. to prove a property, is to apply induction-based methods as proposed in [20], [21]. For verifying safety properties the basic idea is to show that, if p holds after k time steps, then it must also hold after the $(k+1)$-th step. For completeness, a constraint requiring the states of an execution path to be unique has to be added.

CBMC [22] is an implementation of BMC for C programs applying loop unwinding. In particular, CBMC adds *unwinding assertions* for the unwound loops. If such an assertion is violated, not all possible execution paths of the program have been checked. Hence, the corresponding loop is unwound once more. User-input can be modeled by means of built-in non-deterministic choice functions. CBMC supports assertions and assumptions embedded in the program code. Assertions are checked for all execution paths of the program that satisfy the assumptions.

B. SystemC Basics

In the following only the essential aspects of SystemC are described. SystemC provides a single language to model and execute hardware and software systems on various levels of abstraction. SystemC has been implemented as a C++ class library, which includes an event-driven simulation kernel. The structure of the system is described with ports and modules, whereas the behavior is described in processes which are triggered by events and communicate through channels. A process gains the *runnable* status when one or more events of its sensitivity list has been notified. If more than one process is runnable, the simulation kernel selects an arbitrary process and gives this process the control. The execution of a process is non-preemptive, i.e. the kernel receives the control back if the process has finished its execution or suspends itself by calling *wait()*.

The simulation semantics of SystemC can be summarized as follows [3]: First, the system is elaborated, i.e. instantiation of modules and binding of channels and ports. Then, there are the following steps to process:

1) Initialization: processes are made runnable.
2) Evaluation: A runnable process is executed or resumes its execution. In case of immediate notification, a waiting process becomes runnable immediately. This step is repeated until no more processes are runnable.
3) Update: Updates of signals and channels are performed.
4) Delta notification phase: If there are delta notifications, the waiting processes are made runnable, and then it is continued with Step 2.
5) If there are timed notifications, the simulation time is advanced to earliest one, the waiting processes are made runnable, and it is continued with Step 2, otherwise the simulation is stopped.

In this paper we focus on untimed SystemC TLM designs, thus timed notifications and simulation time are irrelevant. Furthermore, dynamic process creation and recursion are not allowed. Moreover, we assume that a SystemC design repeatedly receives input from the environment/user. The simpler, special case, where a design receives some inputs, processes them and then terminates, is not explicitly discussed for the sake of simplicity.

IV. TLM Property Checking

This section presents the property checking approach to verify transaction and system-level properties of untimed SystemC TLM designs. Before we give the details, first a simple but conceptually representative SystemC TLM model is discussed. Moreover, this model also serves as running example throughout the rest of this paper.

Example 1: The SystemC TLM program shown in Figure 1 models a simple communication between an initiator and a target using an internal event (declared in Line 19). The example has two processes: *initiate* (Line 13) from the initiator and *increase* (Line 26) from the target. The target is connected to the initiator through a port (Line 35). The process *increase* waits for the notification of the internal event *e* before it increases the variable *number*. The event *e* will be notified when the function *activate* (Line 25) of the target is called from the process *initiate* (Line 13) through the port.

Obviously, already this simple example shows that deriving a formal model as basis for property checking is non-trivial. Moreover, the sample points for the temporal operators have to be defined, a convenient property specification language has to be identified as well as an appropriate verification method has to be found. The solutions to these questions are introduced in the following subsections. Before they are presented the overall flow of our approach is illustrated in Figure 2. At first, the model generation is performed which basically transforms the SystemC TLM model to C and integrates an abstracted static SystemC scheduler (see Section IV-A). Then, the monitoring logic for a concrete TLM property is built and embedded into the C model. This task including the property language

```
1  class activate_if : virtual public sc_interface {
2  public:
3      virtual void activate() = 0;
4  };
5
6  class initiator : public sc_module {
7  public:
8      sc_port<activate_if> port;
9      SC_HAS_PROCESS(initiator);
10     initiator(sc_module_name name) : sc_module(name) {
11         SC_THREAD(initiate);
12     }
13     void initiate() { port−>activate(); }
14 };
15
16 class target : public activate_if, public sc_module {
17 public:
18     int number;
19     sc_event e;
20     SC_HAS_PROCESS(target);
21     target(sc_module_name name) : sc_module(name) {
22         number = 0;
23         SC_THREAD(increase);
24     }
25     void activate() { e.notify(SC_ZERO_TIME); }
26     void increase() {
27         wait(e);
28         number++;
29     }
30 };
31
32 int sc_main (int argc , char ∗argv[]) {
33     initiator initiator_inst(”Initiator”);
34     target target_inst(”Target”);
35     initiator_inst.port(target_inst);
36     sc_start();
37     return 0;
38 }
```

Fig. 1. Simple SystemC TLM program

and mappings for the different variants of TLM properties are discussed in Section IV-B. Finally, the BMC-based verification method and the necessary formalization to search for property violations is detailed in Section IV-C.

A. Model Generation

As suitable formal model we have chosen C. On the one hand the transformation process is manageable and can be automated (as done in this work). On the other hand we can leverage available model checkers.

The transformation into a C model consists of three steps, which will be demonstrated for Example 1. At the beginning of the first step, we identify the static elaborated structure of the design (that means the module hierarchy, the processes and the port bindings). Afterwards the object-oriented features of SystemC/C++ are translated back into plain C. Member variables, member functions and constructors of each object are transformed to global variables and global functions. The port bindings are resolved already, thus all function calls through a port can now be replaced by calls of actual functions. The

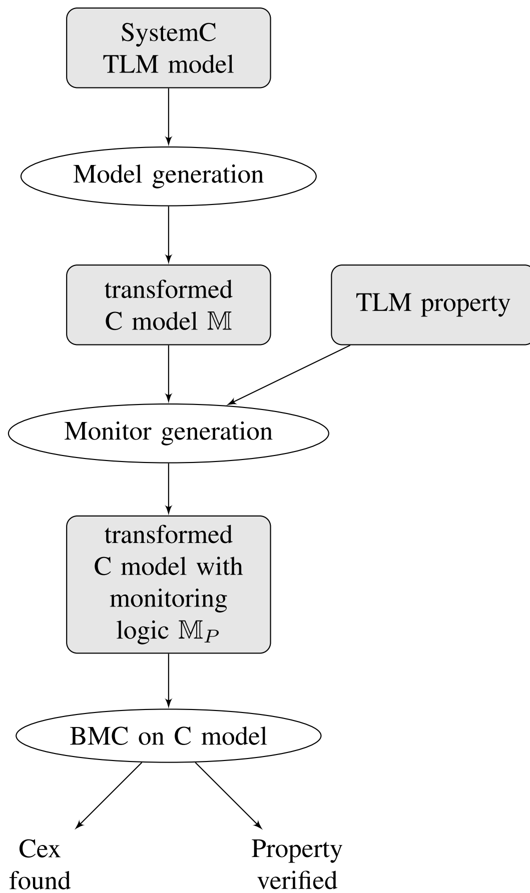

Fig. 2. Overall Flow

```
1   void initiator_inst_init() { }
2   void initiator_inst_initiate() {
3       target_inst_activate();
4   }
5
6   int target_inst_number;
7   sc_event target_inst_e;
8
9   void target_inst_init() {
10      target_inst_number = 0;
11  }
12  void target_inst_activate() {
13      target_inst_e.notify(SC_ZERO_TIME);
14  }
15  void target_inst_increase() {
16      wait(target_inst_e);
17      target_inst_number++;
18  }
19
20  int main(int argc , char *argv[]) {
21      initiator_inst_init();
22      target_inst_init();
23      sc_start();
24      return 0;
25  }
```

Fig. 3. Result of the first step

result of the first step for Example 1 is shown in Figure 3. For example, the transformed code for the target module is shown between Line 6 and Line 18. The first two lines define two global variables, which were member variables of the module before (Line 18 and Line 19, respectively). The remaining lines show the transformed constructor *target_inst_init* and two former member functions *target_inst_activate* and *target_inst_increase*. At the beginning the of *main* function, the two transformed constructors are called (Line 21 and Line 22). That corresponds to the instantiation of the two modules in *sc_main*.

The second step generates the static scheduler implementing the non-preemptive simulation semantics of SystemC. A counter for the number of runnable processes *runnable_count* is added. For each process a global variable indicating the status of the process is generated (RUNNING, RUNNABLE, WAITING, or TERMINATED). The delta cycle loop and the evaluation loop are also inserted into the *main()* function. In the body of the evaluation loop, non-deterministic choice, i.e. which runnable process is to be executed next, is implemented. This non-deterministic choice allows a C model checker to explore *all interleavings* implicitly. In case the design contains only immediate notifications, the delta cycle loop is unnecessary and can thus be removed. The handling of events completes the scheduler and is considered in the next and last step. In Example 1 the call of *sc_start* (Line 36) is replaced with the code in Figure 4. Note that implementation details of non-deterministic choice of runnable processes are unnecessary to understand the transformation and thus are removed.

```
1   while (runnable_count > 0) { // delta cycle loop
2       while (runnable_count > 0) { // evaluation loop
3           choose_one_runnable_process();
4           runnable_count−−;
5           if (process initiate is chosen) {
6               initiator_inst_initiate_status = RUNNING;
7               initiator_inst_initiate();
8           }
9           if (process increase is chosen) {
10              target_inst_increase_status = RUNNING;
11              target_inst_increase();
12          }
13      }
14  }
```

Fig. 4. Scheduler implementation

At the beginning of the last step of the transformation, all function calls are inlined. After that the remaining function calls are *notify()* and *wait()*. The handling of events is mapped to the handling of Boolean flags. For each event E a Boolean flag *E_notified* is generated and for each process P, that will be waiting for the notification of E at some point, a Boolean flag *P_waiting_E* is added. After each potential context switch (a call of *wait()*), a label (resume point) is inserted. The execution of the corresponding process can be resumed later by jumping

```
1  if (target_inst_e_notified) {
2      target_inst_e_notified = false;
3      if (target_inst_increase_waiting_target_inst_e) {
4          target_inst_increase_waiting_target_inst_e = false;
5          target_inst_increase_status = RUNNABLE;
6          runnable_count++;
7          }
8  }
```

Fig. 5. Delta notification

```
1  if (target_inst_increase_current_resume_point==1)
2      goto target_inst_increase_resume_point_1;
3  target_inst_increase_waiting_target_inst_e = true;
4  target_inst_increase_status = WAITING;
5  target_inst_increase_current_resume_point = 1;
6  goto target_inst_increase_end;
7  target_inst_increase_resume_point_1: ;
8  target_inst_number++;
9  target_inst_increase_status = TERMINATED;
10 target_inst_increase_end: ;
```

Fig. 6. The inlined call of *target_inst_increase*

to this label. For each process we also need a variable to keep track of its current resume point. The notifications of an event *E* and the waits on *E* are modeled as follows. Each notification raises the flag *E_notified* and the status of any process *P* currently waiting for *E* (flag *P_waiting_E*) becomes *runnable* immediately for immediate notification, and at the delta notification phase for delta notification. The code in Figure 5 models the delta notification phase and is to be inserted after the evaluation loop. Each call of *wait(E)* in the process *P* is replaced with the following: the process status is set to *waiting*, *P_waiting_E* is raised, the current resume point of *P* is set accordingly and *P* is interrupted by jumping to the end of the process (see Line 6 of Figure 6). Figure 6 shows the inlined code for the call *target_inst_increase()*. There is only one resume point in this process defined on Line 7. The first two lines implement the resuming of the process. Line 3-6 show the implementation of *wait(target_inst_e)*.

In the remainder of this paper we denote the transformed C model as $\mathbb{M}$.

B. Property Language and Monitor Generation

For property specification we use PSL [18] which initially was not designed for property specification at high level of abstraction. In [17] additional primitives have been introduced – coming from the software world – which are well suited for TLM property specification. Besides basic atomic primitives we use the following:

- *func_name:**entry*** - start of a function/transaction
- *func_name:**exit*** - end of a function/transaction
- *event_name.**notified*** - notification of an event
- *func_name:non−negative integer* - return value and parameters of a function/transaction

It is left to define the temporal sampling rate as well as the supported temporal operators. As **default temporal resolution** we sample at start/end of any transaction and at notification of any event. PSL clock expressions[1] can be used to change the temporal resolution, e.g. to sample only at notification of a certain event. As temporal operators we allow ***always*** and ***next*** . Both have similar semantics as G and X in LTL, however the "time" advances with respect to the specified (default) temporal resolution.

In the following we discuss different useful types of properties and the generation of *monitoring logic* by means of C assertions. The task of the monitoring logic is to detect the violation of the property. The properties – simple safety up to system-level properties – can be translated to appropriate assertions as described below. After its generation the monitoring logic becomes a part of the model. For the model including the monitor for a property P we use the notation $\mathbb{M}_P$ (see also Figure 2). Then, we have to define when a property holds.

Definition 1: The property P holds in the original design or in the transformed model $\mathbb{M}$, respectively, iff no assertions fail in any execution of $\mathbb{M}_P$.

Now the different types of properties and the respective monitoring logic are explained.

1) Simple Safety Properties: This type of properties concern values of variables of the TLM model at any time during the execution, e.g. the values of some certain variables should always satisfy a given constraint. Generally, this property type can be expressed by a C logical expression. To verify those properties we only need to insert assertions right after the lines of code that change the value of at least one variable involved. As an example see the property depicted at the top of Figure 7 specified for a FIFO.

2) Transaction Properties: This type of properties can be used to reason about a transaction effect, e.g. checking whether a request or a response (both are parameters or return value of some functions) is invalid or whether a transaction is successful. Monitoring logic for these properties is created by inserting assertions before/after the body of corresponding inlined function calls. For example, the property "the memory read transaction always operates on a valid address" for a TLM bus can be formulated in a transaction property as shown in the middle of Figure 7.

3) System-level Properties: These properties focus on the order of occurrences of event notifications and transactions, e.g. a given transaction should only begin after a certain event has been notified. We implement the monitoring logic using FSMs. Each state of the FSM corresponds with one position in the order specified by the property. Code for transitions of the FSM is inserted right after event notifications, begin or end of transactions (depending on the property). The FSM also has one state indicating the violation of the property. Our assertion is that this state is never reached. As example see the lower part of Figure 7. The first system-level property has been specified for Example 1 and states that after the transaction *activate* has finished the event is notified which causes the

[1] In the considered TLM models there are no clocks. We only use the clock expression syntax to define sampling points.

Simple safety property:
// the number of processed blocks never exceeds the number of blocks which have been read
always *(num_block_processed <= num_block_read)*

Transaction property:
// the memory read transaction always operates on a valid address
always *(0 <= mem_read:1 && mem_read:1 <= MAX_ADDR)*

System-level property:
// Two properties for running Example 1
always *(target_inst.activate:**exit** −>**next** (target_inst.e.**notified** && **next** target_inst.increase:**exit**))*
always *(target_inst.e.**notified** −>**next** (target_inst.increase:**exit** && target_inst.number == 1))*

Fig. 7. Several example properties

execution of the function *increase* to end. The second system-level property additionally defines the expected value of the integer *number* of the target.

The next section gives a detailed presentation on verifying the C model using BMC. Recall that the C model $\mathbb{M}_P$ at hand has been automatically generated and the monitoring logic for P has also been embedded automatically.

C. BMC-based Verification

First of all we explain the notion of states and how the transition relation is formed with respect to the transformed SystemC TLM model including the assertions, i.e. $\mathbb{M}_P$. The basic idea is to view the current values of the variables as a state s and each iteration of the outermost loop of the scheduler – which is either the evaluation loop or the delta cycle loop – as the transition relation T. In the following we will also refer to this outermost loop as the *main loop* (see also Figure 4). Each execution of the model can be formalized as a path, which is a sequence of states $s_{[0..n]} = s_0 s_1 ... s_n$ satisfying the condition

$$path(s_{[0..n]}) = \bigwedge_{0 \leq i < n} T(s_i, s_{i+1}).$$

Note that the path can be infinite, in that case $n = \infty$.

The property P holds in the original design, iff the general property "no assertions fail" holds in $\mathbb{M}_P$, which also means no assertion failure during each iteration of the main loop, or in other words during each transition $T(s_i, s_{i+1})$.

Definition 2: A transition without assertion failure is called *safe* and written as $safe(s_i, s_{i+1})$.
Thus, for the property to hold, every sequence of states of an execution must satisfy as well the condition

$$allSafe(s_{[0..n]}) = \bigwedge_{0 \leq i < n} safe(s_i, s_{i+1}).$$

The need for safe transitions instead of the conventional safe states is explained as follows. The transition relation in our context is defined by the main loop. Therefore, a state and its successor state are defined at the start and at the end of each iteration of the main loop, respectively. When an assertion fails, the execution is immediately stopped somewhere in the middle of an iteration. We already left the last state but have not reached the next one yet. It follows that the need for safe transitions is directly implied by the way the monitoring logic is generated. On the other hand, if we want to use the notion of safe states, the monitoring logic must be modified as follows. We would need to add one more Boolean flag indicating whether the property is already found to be violated. Then, each assertion would be replaced with a piece of code, that raises the flag and jumps to the end of the evaluation loop, where the flag is asserted to be false. However, this extension makes the model and its state space bigger, thus using safe transitions is actually better.

Let I be the characteristic predicate for all initial states, which are reachable states before entering the main loop – note that there can be more than one initial state because some variables are uninitialized or modeled as inputs, and thus have a non-deterministic value. Then, the BMC problem can be formulated as proving that there exists an execution path of length k, starting from an initial state, and containing unsafe transitions. This is encoded in the following formula:

$$\exists s_0 ... s_k . \; \big(I(s_0) \wedge path(s_{[0..k]}) \wedge \neg allSafe(s_{[0..k]}) \big)$$

Now BMC checks the formula for increasing k starting from zero. In the experiments (see Section VI-A) we show that this already gives good results. At our level of abstraction, for a fixed value of k checking the above formula is equivalent to verify the program $\mathbb{M}_P^k$, which is $\mathbb{M}_P$ with the main loop unwound k times. If a trace is returned, the formula is proven to hold for that fixed value of k, and the property P is proven to be false. Otherwise, the property holds up to k. Recall that as mentioned above a SystemC design repeatedly receives input from the environment/user. Hence, for a complete proof the property has to hold for all values of k. In principle, we do not need to check a transition more than one time, thus we can stop increasing k if it reaches the number of states. However, this becomes infeasible very fast. Hence, we devise a method using induction where we derive much better terminating conditions. The main advantages of the induction-based method are that much better run-times are achieved and the method is complete, i.e. properties are proven not only up to a certain bound k but under all circumstances.

```
1  k = some constant which can be greater than zero
2  while true do
3      if not CPROVER(M^k_P) then
4          return Trace c_[0..k]
5      if CPROVER( Σ_{i=1}^{k} ( s_{i-1} = s; M[i] assume(newState(i)); ) assert(!(runnable_count > 0)); ) then
6          return true
7      if CPROVER( s = non_det; Σ_{i=1}^{k} ( s_{i-1} = s; M_P̂[i] assume(newState(i)); ) M_P[k + 1] ) then
8          return true
9      k = k + 1
10 end
```

Algorithm 1. High-level strengthened induction with depth for transformed SystemC TLM model including monitoring logic

V. Induction-based TLM Property Checking

This section introduces the induction-based method which forms a complete and efficient approach to prove transaction and system-level properties. Traditional induction-based techniques (like e.g. [20]) addressed safety state properties in the context of circuits, whereas our general property *safe* for $\mathbb{M}_P$ involves a transition (a pair of states) and our level of abstraction is higher. Nevertheless, the underlying ideas give a good starting point.

First, we only need to check each transition once, thus only paths, where all states except the last one are different, need to be considered. Second, after the base case is proved (i.e. no counter-example of length up to k exists), we check two terminating conditions: the "forward" condition and the inductive step. The forward condition checks whether a path of length $(k + 1)$ starting from an initial state exists. The inductive step checks if a path with k first safe transitions and a last unsafe one exists. If no such paths exist, we can stop and conclude that the property P holds. In summary, the forward condition checks the satisfiability of

$$I(s_0) \wedge loopFree(s_{[0..k]})$$

and the inductive step checks the satisfiability of

$$loopFree(s_{[0..k]}) \wedge allSafe(s_{[0..k]}) \wedge \neg safe(s_k, s_{k+1})$$

where

$$loopFree(s_{[0..k]}) = path(s_{[0..k]}) \wedge \bigwedge_{0 \le i < j \le k} s_i \neq s_j.$$

Now, to make induction possible at our level of abstraction, the main challenge is the embedding of the constraints into the transformed C model. Conceptually, new variables s_0, s_1, ... are needed to capture the state s after each iteration of the main loop and the constraints can then be imposed by means of assumptions[2]. For example, the constraint $loopFree(s_{[0..k]})$ can be emulated by inserting the statement $\mathrm{assume(newState(i));}$ after the i-th unwound iteration of the main loop with

$$newState(i) = (s! = s_{i-1})\ \&\&\ ...\ \&\&\ (s! = s_0).$$

For a precise description of the algorithm, we use the interpretation of C programs as strings. As defined earlier, $\mathbb{M}^k_P$ is $\mathbb{M}_P$ with the main loop unwound k times. Let $\mathbb{M}_P[i]$ be the code fragment of the i-th unwound iteration of the main loop of $\mathbb{M}_P$ and let $+$ be the string concatenation operator. Then, we have

$$\mathbb{M}^k_P = \mathbb{M}_P[1]\mathbb{M}_P[2] \ldots \mathbb{M}_P[k] = \sum_{i=1}^{k} \mathbb{M}_P[i].$$

Additionally, we define $\mathbb{M}_{\widehat{P}}$ as the resulting program after each assertion related to P in $\mathbb{M}_P$ is substituted by an assumption. The introduced notation applies for $\mathbb{M}$ and $\mathbb{M}_{\widehat{P}}$ as well. We end up with a *high-level strengthened induction with depth* shown in Algorithm 1, which has a similar structure as the Algorithm 3 and 4 of [20], but the level of the induction differs significantly. The base case is checked in the first if-statement (Line 3). The second if-statement (Line 5) checks the forward condition. The assertion at the end of Line 5 is the unwinding assertion for the main loop. The last if-statement (Line 7) is the inductive step. The first arbitrary state is emulated by the statement $s = non_det$, which assigns non-deterministic values to the variables and thus allows the model checker to examine all possible values implicitly. The underlying C model checker is invoked to verify the assertions in the passed parameter by the call $CPROVER$, which returns *true* if no assertions are violated and *false* otherwise, in this case a violating trace can be extracted.

As a final note, our approach can deal with nested loops, which are commonly present in our transformed models. The outermost loop is handled explicitly as described above. The other loops must be unwound up to at least their run-time bounds before applying our approach. Those run-time bounds can be determined with the aid of unwinding assertions [22]. Also note that unbounded loops can still have a finite run-time bound due to the simulation semantics of SystemC and our transformation method. As an example, consider the infinite

[2] C model checkers typically support an assumption concept, i.e. assertions are checked for all execution paths of the program that satisfy the assumptions.

```
1  if (t_current_resume_point==1)
2     goto resume_point_1;
3  // begin of the first unwound iteration
4  body1;
5  t_waiting_e = true;
6  t_status = WAITING;
7  t_current_resume_point = 1;
8  goto t_end;
9  t_resume_point_1:
10 body2;
11 // end of the first unwound iteration
12 // begin of the second unwound iteration
13 body1;
14 t_waiting_e = true;
15 t_status = WAITING;
16 t_current_resume_point = 1;
17 goto t_end;
18 body2;
19 // end of the second unwound iteration
20 t_end: ;
```

Fig. 8. The infinite SC_THREAD loop unwound

loop *while (true) { body1; wait(e); body2; }* commonly used in a SC_THREAD. It only needs to be unwound twice. In its first execution, the SC_THREAD performs *body1;* then suspends itself because of *wait(e)*. Any further execution resumes exactly after the wait statement, performs *body2;* then *body1;* and is suspended again by *wait(e)*. Figure 8 shows the transformation with two unwound iterations for a SC_THREAD named *t*.

VI. Experiments

The proposed approach has been implemented and evaluated on different TLM designs. The model checker CBMC v3.3 [22] with Boolector v1.2 [23] as the underlying SMT solver has been used to verify the generated C models. The proposed approach has been built on top of CBMC, i.e. unwinding and the transformations for induction are performed before giving the problem to CBMC. The internal slicer of CBMC is activated to remove subformulas, which are irrelevant for the properties. All experiments have been carried out on a 3GHz Intel Xeon system with 4 GB RAM running Linux.

In the first part of the experiments we present the results for the BMC-based verification approach. Then, in the second part we give the results for the induction-based method as introcuded in Section V.

A. BMC-based Verification

The first design is the *simple_fifo* TLM example included in the official SystemC distribution. The original design consists of a consumer module and a producer module communicating over a FIFO channel. Both modules have their own SC_THREAD. We modified the design so that the producer writes an infinite sequence of arbitrary characters into the FIFO. The SystemC model (the generated C model) has approximately 80 (150) lines of code. We considered the following properties of the FIFO:

TABLE I
Results for corrected FIFO

	1 consumer + 1 producer		
	48 chars	64 chars	80 chars
P1	26.37s	51.55s	67.69s
P2	25.97s	49.39s	64.96s
P3	38.23s	67.92s	105.02s
P4	28.04s	54.60s	72.98s
	2 consumers + 1 producer		
P1	178.18s	302.42s	492.24s
P2	170.28s	351.68s	468.60s
P3	250.40s	515.42s	677.92s
P4	194.75s	400.92s	530.50s
	1 consumer + 2 producers		
P1	438.25s	919.20s	1193.58s
P2	429.45s	920.38s	1191.10s
P3	617.08s	1280.24s	1690.12s
P4	479.66s	969.02s	1275.94s

TABLE II
Results for disproving P1 on original FIFO

2 c. + 1 p.	77.65s
1 c. + 2 p.	190.05s

- The number of elements in the FIFO never exceeds the limit - P1: ***always*** *(0 <= num_elements && num_elements <= max)*
- After a *write* transaction, the FIFO is not empty - P2: ***always*** *(write:**exit** −>num_elements > 0)*
- If the FIFO is full, the next event notified is *read_event* - P3: *default clock = read_event.**notified** || write_event.**notified**; **always** (num_elements == max −>**next** read_event.**notified**)*
- After a notification of *read_event*, the next 10 (the FIFO size) notifications includes at least one notification of *write_event* - P4: *default clock = read_event.**notified** || write_event.**notified**; **always** (read_event.**notified** −>**next_e**[1:10] write_event.**notified**)*

The design malfunctioned as soon as we tried to connect more consumers or producers to the FIFO channel. We fixed the implementation and proved the properties on the corrected design. Since the BMC-based method from Section IV-C is incomplete, the properties can only be verified for a fixed number of inputs. The results are shown in Table I. Each subcolumn gives the run-times required to verify each property for the bounded input of 48, 64 and 80 arbitrary characters, respectively. As can be seen the run-times increase with the number of characters. Moreover, adding another consumer or producer also results in higher run-times.

Table II shows the run-times needed by CBMC to disprove P1 on the original FIFO with 2 consumers and 1 producer, and with 1 consumer and 2 producers, respectively. In both cases, 48 characters are written into the FIFO.

As another SystemC model we considered the *chain* benchmark presented in [13]. This benchmark consists of a chain of m modules communicating through transactions. Each module has a SC_THREAD, which waits for an internal event before

TABLE III
COMPARISON ON *chain* BENCHMARK

m	CBMC	[13] w/o sched.	[13] w. sched.
5	1.01s	1.40s	1.80s
9	5.14s	2.08s	7.28s
13	25.86s	11.71s	118.78s
17	92.07s	166.73s	2443.77s
19	180.01s	686.93s	not completed
21	383.92s	3201.33s	not completed

TABLE IV
RESULTS FOR CORRECTED FIFO USING INDUCTION

	1 c. + 1 p.		2 c. + 1 p.		1 c. + 2 p.	
	BS	IS	BS	IS	BS	IS
P1	0.01s	1.09s	0.01s	2.04s	0.01s	2.50s
P2	0.01s	0.92s	0.01s	1.93s	0.01s	2.35s
P3	0.77s	11.90s	4.49s	48.18s	7.66s	59.07s
P4	0.55s	8.22s	1.41s	18.24s	3.20s	39.39s

initiating a transaction with the next module. This transaction notifies the internal event of the next module, so that this module can start a transaction with the after next, and so on, until the last module completes its transaction. No "real" property is checked in the benchmark, instead the whole model state space is explored. The results are shown in Table III. The first column gives the number of modules/processes in the chain. The column "CBMC" provides our results, while the results from [13] using the encoding without and with a non-preemptive scheduler respectively[3] can be seen in the last two columns. The results show that our approach can handle a large number of processes and scales much better than [13].

B. Induction-based Verification

We applied our induction-based method to the FIFO design discussed in the previous section. The results shown in Table IV are obtained without the restriction to loop-free paths. The sub-columns "BS" and "IS" give the run-time of the base step and the inductive step, respectively. Significant improvements over the BMC-based method with respect to run-time can be observed. Furthermore, using induction, the proofs are *complete*, i.e. the properties are verified for unbounded input where arbitrary characters are repeatedly written into the FIFO.

The last design considered in this paper is our TLM implementation of a part of a JPEG encoder [24] consisting of eight modules: a simple bus, two memory slaves, a reader, a zig-zag scanner, a run-length encoder, an output module, and a controller coordinating the encoding process. The input of the design are quantized DCT 8x8 pixel blocks and the output are run-length encoded sequences. The SystemC model (the generated C model) has approximately 200 (450) lines of code. We successfully verified the following properties using induction:

- All memory accesses are successful, i.e. *mem_read* and *mem_write* of the bus always return *true* - P5: ***always*** *(mem_write:0 && mem_read:0)*

[3]Their experiments were done on a 2GHz AMD Opteron system with 4GB RAM running Linux.

TABLE V
RESULTS FOR RLE ENCODER DESIGN

	BS	IS
P5	0.01s	5.17s
P6	0.01s	5.41s
P7	85.82s	218.90s

- As specified in [24], while encoding the 8x8 block, if 16 consecutive zeros are encountered, a pair (15, 0) should be written to the output, instead of waiting for a non-zero value. Therefore, the first parameter of the *write* method of the output module should always be less than or equal to 15 - P6: ***always*** *(write:1 <= 15)*
- The synchronization between the threads implies, that after the completion of each scan transaction, the next scan should always follow after two other transactions - P7: *default clock = read_block:**exit** || zigzag_scan:**exit** || rle_encode:**exit**;* ***always*** *(zigzag_scan:**exit** −>**next[3]** zigzag_scan:**exit**)*

The results are given in Table V. Again, the efficiency of the induction-based method can be observed and the proofs are *complete*, i.e. the properties are proven for any number of input blocks and arbitrary blocks' contents.

VII. CONCLUSIONS

We have presented an efficient property checking approach for untimed SystemC TLM designs. The approach consists of three steps: the fully automated transformation of SystemC to C, the generation and embedding of monitoring logic for a TLM property, and the verification of the transformed C models. For the verification task, a BMC formulation over the outermost loop of the scheduler has been developed. Furthermore, we improved the BMC-based technique with respect to efficiency and completeness by performing induction at the level of C programs. The experiments show that complete proofs of important TLM properties can be carried out efficiently. The large state spaces of SystemC designs, which also consist of all possible inputs and interleavings, are fully explored by our approach.

For future work we see possible enhancements by incorporating some light-weight static partial order reduction techniques into the generated scheduler to prune unnecessary interleavings. Furthermore, we would like to investigate the use of abstractions. For the verification of a property with fine temporal granularity, providing sound abstractions for the model behavior at the start/end of irrelevant transactions or at the notification of unrelated events would reduce the model size and its state space. Abstraction is also needed to deal with library code. In addition, we also want to evaluate the use of other C model checkers and to extend our approach to handle timed SystemC TLM constructs. This extension is a necessary next step towards the formal verification for designs using the TLM-2 library.

REFERENCES

[1] B. Bailey, G. Martin, and A. Piziali, *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. Morgan Kaufmann/Elsevier, 2007.

[2] *Functional Specification for SystemC 2.0*, Synopsys Inc., CoWare Inc., and Frontier Design Inc., http://www.systemc.org.

[3] *IEEE Standard SystemC Language Reference Manual*, IEEE Std. 1666, 2005.

[4] M. Y. Vardi, "Formal techniques for SystemC verification," in *Design Automation Conf.*, 2007, pp. 188–192.

[5] A. Habibi and S. Tahar, "On the extension of SystemC by SystemVerilog assertions," in *Canadian Conference on Electrical and Computer Engineering*, 2004, pp. 1869–1872.

[6] W. Ecker, V. Esen, M. Hull, T. Steininger, and M. Velten, "Requirements and concepts for transaction level assertions," in *Int'l Conf. on Comp. Design*, 2006, pp. 286–293.

[7] N. Bombieri, F. Fummi, and G. Pravadelli, "Incremental ABV for functional validation of TL-to-RTL design refinement," in *Design, Automation and Test in Europe*, 2007, pp. 882–887.

[8] A. Habibi and S. Tahar, "Design for verification of SystemC transaction level models," in *Design, Automation and Test in Europe*, 2005, pp. 560–565.

[9] M. Moy, F. Maraninchi, and L. Maillet-Contoz, "LusSy: an open tool for the analysis of systems-on-a-chip at the transaction level," *Design Automation for Embedded Systems*, pp. 73–104, 2006.

[10] D. Karlsson, P. Eles, and Z. Peng, "Formal verification of SystemC designs using a petri-net based representation," in *Design, Automation and Test in Europe*, 2006, pp. 1228–1233.

[11] B. Niemann and C. Haubelt, "Formalizing TLM with communicating state machines," in *Forum on specification and Design Languages*, 2006, pp. 285–293.

[12] C. Traulsen, J. Cornet, M. Moy, and F. Maraninchi, "A SystemC/TLM semantics in promela and its possible applications," in *SPIN*, 2007, pp. 204–222.

[13] C. Helmstetter and O. Ponsini, "A comparison of two SystemC/TLM semantics for formal verification," in *ACM & IEEE International Conference on Formal Methods and Models for Codesign*, 2008, pp. 59–68.

[14] P. Herber, J. Fellmuth, and S. Glesner, "Model checking SystemC designs using timed automata," in *IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, 2008, pp. 131–136.

[15] S. Kundu, M. Ganai, and R. Gupta, "Partial order reduction for scalable testing of SystemC TLM designs," in *Design Automation Conf.*, 2008, pp. 936–941.

[16] N. Blanc and D. Kroening, "Race analysis for SystemC using model checking," in *Int'l Conf. on CAD*, 2008, pp. 356–363.

[17] D. Tabakov, M. Vardi, G. Kamhi, and E. Singerman, "A temporal language for SystemC," in *Int'l Conf. on Formal Methods in CAD*, 2008, pp. 1–9.

[18] *Accellera Property Specification Language Reference Manual, version 1.1*, http://www.pslsugar.org, 2005.

[19] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Tools and Algorithms for the Construction and Analysis of Systems*, 1999, pp. 193–207.

[20] M. Sheeran, S. Singh, and G. Stålmarck, "Checking safety properties using induction and a SAT-solver," in *Int'l Conf. on Formal Methods in CAD*, 2000, pp. 108–125.

[21] P. Bjesse and K. Claessen, "SAT-based verification without state space traversal," in *Int'l Conf. on Formal Methods in CAD*, 2000, pp. 372–389.

[22] E. M. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2004, pp. 168–176.

[23] R. Brummayer and A. Biere, "Boolector: An efficient SMT solver for bit-vectors and arrays," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2009, pp. 174–177.

[24] ISO, *ISO/IEC 10918-1:1994: Information technology — Digital compression and coding of continuous-tone still images: Requirements and guidelines*. International Organization for Standardization, 1994.