Towards Proving TLM Properties with Local Variables *

Hoang M. Le Daniel Große Rolf Drechsler
Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
$\{hle, grosse, drechsle\}$@informatik.uni-bremen.de

***Abstract*—With the growing popularity and adoption of Electronic System Level (ESL) design, the verification of SystemC models at Transaction Level (TLM) has become an important research problem. In the context of formal verification, most of the existing approaches for SystemC TLM only consider basic properties such as deadlock-freedom or local assertions. In previous work, a high-level BMC-based property checking approach has been introduced. This approach enables to verify important TLM behavior such as the effect of a transaction and that the transaction is only started after a certain event. This paper proposes an extended approach to handle properties with local variables. The variables are used to store data values which will be referenced later at certain TLM events. By this, data integrity can be formally verified. A technique to improve the efficiency by avoiding the need to consider overlapping scenarios is included. Preliminary results obtained by applying the proposed approach to a design from the OSCI SystemC distribution are presented.**

I. INTRODUCTION

Even if significant improvements in verification techniques have been achieved in the recent years, verification continues to dominate the overall design costs. To manage the steadily increasing complexity, raising the level of abstraction in modeling has been exercised in the past years. As a result, *Electronic System Level* (ESL) design has emerged [1]. In this context, SystemC [2], [3] and its standardized *Transaction Level Modeling* (TLM) abstraction have entered into industry. ESL design using TLM enables early functional verification without considering detailed hardware implementations.

However, for SystemC TLM only a few formal verification approaches have been presented [4], [5], [6], [7], [8]. The majority of them can only verify basic properties such as deadlock-freedom or local assertions. In [7], a property checking approach has been introduced enabling the verification of important TLM behavior such as the effect of a transaction and that the transaction is only started after a certain event. The approach is based on *Bounded Model Checking* (BMC) [9] and uses CBMC, an implementation for C programs [10]. The TLM properties are specified in the *Property Specification Language* (PSL) [11], [12]. However, a major class of behavior cannot be verified with this approach since local variables are not supported. Local variables are necessary if data needs to be captured in a certain situation and this data needs to be referenced later during the execution of the TLM model. For instance, properties describing data integrity cannot be formulated without local variables. A formal semantics extending PSL with local variables can be found in [13]. At RTL several solutions exist, see e.g. [14]. For SystemC however, only the simulation-based approach [15] targets the problem of local variables.

In this paper we extend the approach in [7] to support local variables enabling the verification of data integrity properties. We show how to adopt the monitoring logic that is generated for a TLM property. We also present a technique to improve the efficiency of the underlying BMC proof. Essentially, we use non-determinism to avoid the explicit consideration of different overlapping evaluations of a TLM property.

The remainder of this paper is structured as follows: Section II provides the basics of SystemC and reviews the TLM property checking approach. Then, in Section III the enhancement for local variables is introduced. First results are given in Section IV. Finally, the paper is concluded in Section V.

II. PRELIMINARIES

A. SystemC Basics

In the following only the essential aspects of SystemC are described. For more details we refer to [16], [17]. 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 have been notified. The simulation kernel selects one of the runnable processes 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()*. Basically SystemC offers the following variants of *wait()* and *notify()* for event-based synchronization:

*This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project SANITAS under contract no. 01M3088 and by the German Research Foundation (DFG) within the Reinhart Koselleck project DR 287/23-1.

- *wait(time)* suspends the current process for the given time duration.
- *wait(event)* suspends the current process until the notification of the event.
- *wait(time, event)* suspends the current process for the given time duration or until the notification of the event, whichever comes first.
- *notify(event)* or *event.notify()* notifies the event immediately. Thus, a process waiting for *event* becomes runnable in the current delta cycle. Such an event notification is called *immediate notification*.
- *notify(event, delay)* or *event.notify(delay)* notifies the event after the given time delay. Such an event notification is called *timed notification* and *delta notification* in the special case of zero delay.

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 is carried out. 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*: If there are delta notifications, the waiting processes are made runnable, and then it is continued with Step 2).
5) *Timed notification*: If there are timed notifications, the simulation time is advanced to the earliest one, the waiting processes are made runnable, and it is continued with Step 2). Otherwise the simulation is stopped.

B. High-Level Properties for SystemC

This section briefly describes the PSL-based language used in [7] for property specification. In [12] additional primitives have been introduced to PSL allowing to specify properties at high level of abstraction. Besides the variables in the design, the following primitives are used:

- *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:number* - return value in case $number = 0$ and parameters of a function/transaction otherwise

The *default temporal resolution* samples at all *system events*, which is either the start or the end of any transaction or the notification of any event. It is possible to change the temporal resolution, e.g. to sample only at notification of a certain event. As temporal operators, ***always*** and ***next*** are allowed. Different useful types of properties are summarized in the following.

1) Simple safety properties: This type of properties concern values of variables of the TLM model 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.

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.

3) System-level properties: These properties focus on the order of occurrences of system events, e.g. a given transaction should only begin after a certain event has been notified.

In the remainder of this paper, we also refer to these properties as *TLM properties*. A TLM property can be converted to an equivalent *Finite State Machine* (FSM), which can be used to check if an execution trace satisfies the property. Checking whether the property holds during the execution of the design is done by embedding the FSM into the design together with assertions creating a *monitoring logic*. Examples for properties and FSMs are given in Section III.

C. TLM Property Checking

In this section we briefly review the approach presented in [7] for proving high-level properties of SystemC TLM models. The approach works as follows: First, from the SystemC TLM model, the transformed model $\mathbb{M}$ in C is generated automatically. The transformation consists of three main steps:

1) The static elaborated structure of the design, i.e. the module hierarchy, the processes and the port bindings, is identified. Then the object-oriented features of SystemC/C++ are translated back into plain C. Each object is statically allocated by making its member variables and functions global. The constructors are also transformed to global functions, which are called to initialize the allocated variables. After this step, the SystemC TLM model becomes a set of SystemC processes communicating over global variables under the non-preemptive simulation semantics.
2) The static SystemC scheduler is generated. The scheduler skeleton is illustrated in Fig. 1. As can be seen, it contains all phases from 2 to 5 described in Section II-A. Note that before the depicted scheduler loop is entered, each process gets a global variable indicating its status (RUNNING, RUNNABLE, WAITING, or TERMINATED). Non-deterministic choice, i.e. which runnable process is to be executed next, is embedded into the evaluation loop (Line 2 in Fig. 1). This allows C model checkers to explore *all interleavings* implicitly.
3) To implement the non-preemptive, event-based simulation semantics of SystemC, the handling of events and context switches is mapped to the handling of a set of simple variables. Each event gets a Boolean flag

```
1  while (runnable_count > 0) { // evaluation loop
2    choose_one_runnable_process();
3    runnable_count−−;
4    if (process 1 is chosen) process_1();
5    ...
6    if (process n is chosen) process_n();
7    if (runnable_count == 0) {
8      // delta notification
9      if (event 1 has been delta notified)
10       make_all_waiting_processes_runnable();
11     ...
12     if (event m has been delta notified)
13       make_all_waiting_processes_runnable();
14   }
15   if (runnable_count == 0) {
16     // timed notification
17     t = get_smallest_notification_delay();
18     advance_simulation_time_by(t);
19     reduce_all_delays_by(t);
20     if (notification delay of event 1 == 0)
21       make_all_waiting_processes_runnable();
22     ...
23     if (notification delay of event m == 0)
24       make_all_waiting_processes_runnable();
25   }
26 }
```

Fig. 1. Generated SystemC scheduler

indicating whether it is notified and an integer variable for the notification delay. For each process synchronized by an event, a Boolean flag indicating that the process is waiting for the event is added. After each potential context switch (a call of *wait()*), a label is inserted, to resume the execution of the corresponding process later.

After the model generation, the FSM for the TLM property is embedded into the transformed C model $\mathbb{M}$ in combination with assertions to form the transformed model with monitoring logic $\mathbb{M}_P$. For the verification task, BMC is employed on the C model. The notion of states and how the transition relation is formed with respect to $\mathbb{M}_P$ is also detailed in [7]. The basic idea is to view the current values of the variables as a state s and each iteration of the scheduler loop as the transition relation T. 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 $path(s_{[0..n]}) = \bigwedge_{0 \leq i < n} T(s_i, s_{i+1})$.

The TLM property P holds in the original design, iff no assertions (introduced by the monitoring logic) fail during each iteration of the main loop, or in other words during each transition $T(s_i, s_{i+1})$. Such a transition is called *safe* and written as $\mathit{safe}(s_i, s_{i+1})$. The BMC problem is formulated as proving that there exists an execution path of length k, starting from an initial state, and containing unsafe transitions: $\exists s_0 ... s_k.\ \big(I(s_0) \wedge path(s_{[0..k]}) \wedge \neg \mathit{allSafe}(s_{[0..k]})\big)$ with $\mathit{allSafe}(s_{[0..n]}) = \bigwedge_{0 \leq i < n} \mathit{safe}(s_i, s_{i+1})$ and I is the characteristic predicate for all initial states. The BMC-based verification is done by unrolling the scheduler loop (i.e. the transition relation) to a certain depth and applying CBMC [10] to the unrolled C model. For efficiency and completeness, an induction-based proof technique has also been developed. We refer to [7] for more details.

```
1  class fifo : public sc_channel, ... {
2      ...
3      void write(char c_in) {
4        while (num_elements == max) wait(read_event);
5        data[(first + num_elements) % max] = c_in;
6        ++ num_elements;
7        write_event.notify();
8      }
9
10     void read(char &c_out){
11       while (num_elements == 0) wait(write_event);
12       c_out = data[first];
13       −− num_elements;
14       first = (first + 1) % max;
15       read_event.notify();
16     }
17     ...
18 };
```

Fig. 2. *simple_fifo* example

III. Proving TLM Properties with Local Variables

We first motivate the need for local variables in TLM properties. Then, an informal semantics for TLM properties with local variables (also referred to as TLM^{LV} properties in the remainder of this paper) is given. Afterwards, the monitoring logic for such properties is described based on this semantics. This monitoring logic is embedded in the transformed C model enabling the formal verification of the considered TLM^{LV} property using [7]. Furthermore, we can improve the verification efficiency by an optimization shown in the last part.

A. Motivating Example

We consider the *simple_fifo* example included in the official OSCI SystemC distribution. The design consists of a consumer module and a producer module communicating over a circular FIFO channel. Both modules have their own SC_THREAD. The producer puts an infinite sequence of random characters into the FIFO and will be blocked if the FIFO becomes full. The consumer tries to read characters from the FIFO and will also be blocked if the FIFO becomes empty. The implementation of the FIFO channel is depicted partially in Fig. 2 with the blocking mechanism shown on Line 4 and Line 11. The notification of *read_event* and *write_event* indicates that a character has been read or written, respectively (Line 7 and Line 15).

For this design, the following TLM property can be specified and proven using the approach proposed in [7]: After a notification of *write_event*, among the next *max* (the

*P1: default clock = read_event.**notified** || write_event.**notified**;*
***always** (write_event.**notified** −>**next_e**[1:max] read_event.**notified**)*

*P2: default clock = read_event.**notified** || write_event.**notified**;*
***always** ((write_event.**notified** && c_in == 'A')−>**next_e**[1:max] (read_event.**notified** && c_out == 'A'))*

*P3: default clock = read_event.**notified** || write_event.**notified**;*
***always** ((write_event.**notified**, var x = c_in)−>**next_e**[1:max] (read_event.**notified** && c_out == x))*

Fig. 3. TLM Properties for *simple_fifo*

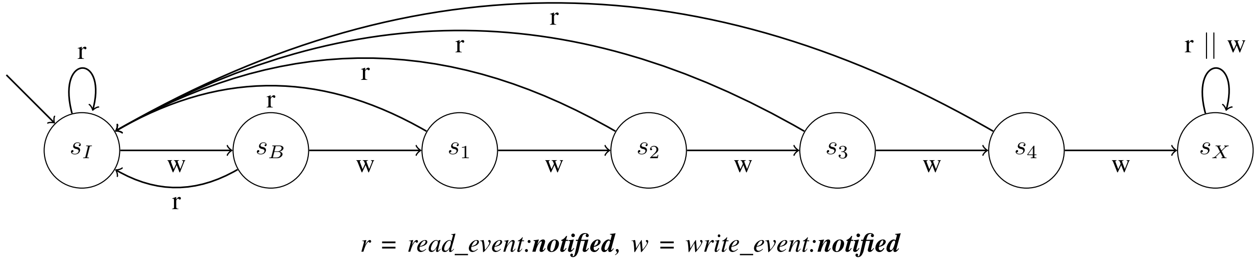

Fig. 4. FSM for P1

FIFO size) notifications there is at least one notification of *read_event*. This is formulated in PSL syntax as the first property P1 in Fig. 3[1]. This property allows to check whether the synchronization of *read* and *write* is correct, but it does not ensure data integrity (i.e. the characters read from and written to the FIFO are the same). For example, if Line 14 is omitted, P1 still holds. The second property P2 in Fig. 3 is a strengthened version of P1 to verify data integrity partially. P2 states that if a character *A* is written to the FIFO, then the same character should be read from the FIFO during the next *max* notifications. Clearly, checking the property for every possible character value is impractical. A better solution is to save the value of *c_in* at the notification of *write_event* in a local variable and compare it to *c_out* at an appropriate notification of *read_event*. This can be achieved by augmenting P1 with a local variable *x* as shown in the third property P3 in Fig. 3. We adopt a very simple syntax: a local variable *x* is introduced in its first assignment *var x = expression*. The next sections discuss the semantics of such properties and how to verify them.

B. Informal Semantics

First, we emphasize that the focus of this paper is not on general PSL properties with local variables. We restrict our discussion to TLM properties as described in Section II-B, augmented with local variables so that the values of some variables of interest can be captured and referenced to during the evaluation of the property. For this class of properties, subtle formal semantics issues can be avoided. Instead, it is sufficient to use the following informal semantics based on the equivalent FSM of a TLM property.

Generally, such a FSM consists of an initial state, several intermediate states, and one state indicating that the property is violated. Transitions between states are conditioned by the occurrence of system events and the values of the SystemC TLM model variables at sampling points as specified by the property. The FSM for the property P1 for the case $max = 5$ is shown in Fig. 4. The state s_B marks the begin of the evaluation after *write_event* has been notified. Each state s_k corresponds to the status of the evaluation after k notifications. A notification of *read_event* enables the transition from each s_k to the initial state s_I indicating a successful evaluation. A notification of *write_event* enables the transition from s_4 to s_X indicating a violation of P1, because *read_event* must have been notified.

Note that it is possible that a new evaluation for the property must be started before previous evaluations have ended, for example consider several consecutive notifications of the event *write_event*. These *overlapping evaluations* can be handled using tokens: Assume that there are an infinite number of tokens in the initial state. Each time an evaluation is triggered, a token is moved from the initial state to the first intermediate state. Every other enabled transition moves all tokens from the source state to the destination. The property fails as soon as a token is moved into the violating state. An evaluation is successful if the corresponding token is returned to the

[1]The *default clock* statement is used to define the temporal resolution. Here, a system event occurs if the read or write event is notified.

initial state.

For TLM[LV] properties, the transitions need to be augmented to include assignments. Consider P3 for example, the transition from s_I to s_B in Fig. 4 becomes *write_event:**notified**, x = c_in*. The tokens also need to be extended to store the captured values. Let $x_1, \ldots, x_n$ be the local variables defined in the considered property. Each token contains a vector of n values $(v_1, \ldots, v_n)$, which are first undefined as the token is moved into the first intermediate state. Then, when x_i is assigned/referenced in an enabled transition, the value of v_i is updated/used accordingly.

C. Monitoring TLM[LV] Properties

Now, we describe how to generate the monitoring logic to be embedded in the transformed C model. For simulation, we can dynamically create a new token for each new evaluation and free it after the evaluation is done. This is not possible in the context of formal verification, instead we need to determine an upper-bound for the number of overlapping evaluations (i.e. the number of tokens needed). Then, the tokens are statically allocated and each token will be reused after the corresponding evaluation is finished. For TLM properties, an upper-bound is the maximum number of sampling points needed for each evaluation. This number can be determined from the syntax of the considered property, for example it is equal to *max* for the property $P3$ because of the operator ***next_e[1:max]***. For each token, we need n C variables for the values of v_i and one additional variable memorizing the current position of the token in the FSM.

The monitoring logic for P3 is shown in Fig. 5 and 6 for a notification of *write_event* and *read_event*, respectively. These are to be embedded in the transformed C model right after the positions where a notification of *write_event* or *read_event* is performed. The variable *pos_1* indicates the position of the first token, and *val_1* is the associated value to capture the value of *c_in*. Line 14 of Fig. 5 and Line 12 of Fig. 6 verify that the token cannot reach the violating state. The FSM transitions are implemented in the if-statements. A token can be used for a new evaluation if its current position is s_I. The value of the variable *tk_found* specifies if a token has been used for the new evaluation already and therefore other available tokens should not be taken. This behavior is exemplarily implemented on Line 9 of Fig. 5. If the condition is satisfied, the first token is moved to state s_B, the associated value *val_1* is assigned to the current value of *c_in*, and the flag *tk_found* is raised. This value *val_1* will be compared to *c_out* whenever *read_event* is notified as can be seen in Fig. 6.

After the monitoring logic has been embedded into the transformed C model, we apply [7] to verify the property. Note that we can optimize the number of tokens needed using the flag *tk_found*. Starting with one token, if a new evaluation should start and all tokens are occupied (*tk_found* == *false* after examining all tokens), we increase the number of tokens needed and start the verification again. This situation can be detected by inserting *assert(tk_found)* at appropriate positions allowing CBMC to report a failure. By applying this incremental procedure, the exact maximum number of overlapping evaluations can be determined. However, a better optimization can be achieved even if this maximum number is equal to the upper-bound as for P3. We present this optimization in the next section.

```
1  tk_found = false;
2
3  // first token
4  if (pos_1 == s4) pos_1 = sX;
5  if (pos_1 == s3) pos_1 = s5;
6  if (pos_1 == s2) pos_1 = s4;
7  if (pos_1 == s1) pos_1 = s3;
8  if (pos_1 == sB) pos_1 = s1;
9  if (pos_1 == sI && !tk_found) {
10     pos_1 = sB;
11     val_1 = c_in;
12     tk_found = true;
13 }
14 assert(pos_1 != sX);
15
16 // second token
17 if (pos_2 == s4) pos_2 = sX;
18 ...
```

Fig. 5. Monitoring logic for P3 – *write_event:**notified***

```
1  // first token
2  if (pos_1 == s4)
3      { if (val_1 == c_out) pos_1 = sI; else pos_1 = sX; }
4  if (pos_1 == s3)
5      { if (val_1 == c_out) pos_1 = sI; else pos_1 = s4; }
6  if (pos_1 == s2)
7      { if (val_1 == c_out) pos_1 = sI; else pos_1 = s3; }
8  if (pos_1 == s1)
9      { if (val_1 == c_out) pos_1 = sI; else pos_1 = s2; }
10 if (pos_1 == sB) {
11     { if (val_1 == c_out) pos_1 = sI; else pos_1 = s1; }
12 assert(pos_1 != sX);
13
14 // second token
15 if (pos_2 == s4)
16     { if (val_2 == c_out) pos_2 = sI; else pos_2 = sX; }
17 ...
```

Fig. 6. Monitoring logic for P3 – *read_event:**notified***

D. Avoiding Overlapping Evaluations

Using the method described in the last section, we need in the worst case m tokens, each requires $n + 1$ additional variables, where n is the number of local variables in the property and m is the bound of the property (i.e. maximum number of sampling points needed for each evaluation). It is well-known that the more additional variables are introduced

TABLE I
RESULTS FOR PROVING DATA INTEGRITY ON FIFO DESIGN

FIFO Size	IP	OPT
max = 5	47.90s	22.25s
max = 10	877.43s	220.69s

the less efficient the verification becomes because of the larger state space. Furthermore, the monitoring logic for each token also makes the verification model larger.

We can reduce the number of tokens needed to just one by introducing non-determinism. The basic idea is that if we start one evaluation of the property from a non-deterministic state of the transformed C model, this evaluation also covers all possible evaluations implicitly and thus we do not need to consider overlapping evaluations. The non-deterministic first state of the C model can be modeled by assigning all variables to a non-deterministic value (the construct *nondet()* is supported directly by CBMC). This non-deterministic first state has been also used to carry out the induction-based proof in [7].

The optimized verification method for TLM^{LV} properties works as follows. After non-determinism has been injected in the first state of the transformed C model, we embed the monitoring logic for the considered TLM^{LV} property using only one token. Then, the scheduler loop (i.e. transition relation) is unrolled and CBMC is applied as in [7]. We need to ensure that the unrolling depth is sufficient to cover one full evaluation because otherwise the property will be satisfied vacuously. This unrolling depth can be determined by integrating an additional counter *sp_cnt* for the number of passed sampling points. At the end of the unrolled C model, *sp_cnt* is asserted to be equal to or greater than the bound m of the considered TLM^{LV} property. The verification is only considered successful if this assertion holds in all possible executions of the unrolled C model. If the assertion has failed, the unrolling depth needs to be increased.

IV. EXPERIMENTAL EVALUATION

In this section, we present the first results for proving TLM^{LV} properties on the introduced SystemC TLM FIFO design. The experiments have been carried out on a 3.4 GHz AMD Phenom II system with 8 GB RAM running Linux. Furthermore, CBMC v4.0 [10] has been used. The results for proving P3 for the FIFO size of 5 and 10 are depicted in Table I (please note that our automated generation of monitoring logic does not support local variables yet, therefore the monitoring logic for P3 must be manually created and hence only small FIFO sizes have been considered). Column IP show the run-times for proving P3 using the monitoring logic described in Section III-C and the induction-based proof method [7]. These results show that data integrity properties for SystemC TLM models can be formally proven. The run-times for proving P3 using the proposed optimization in Section III-D is presented in column OPT. As can be seen, the verification efficiency for TLM^{LV} properties can be considerably improved by using the proposed optimization.

V. CONCLUSIONS

We have presented an extended approach for formal TLM property checking supporting local variables. In particular for properties describing data integrity this feature is very important. The approach extends a high-level BMC-based method. In addition, we have introduced an optimization using non-determinism to improve the efficiency of the underlying BMC proof. First experiments have demonstrated that data integrity properties for SystemC TLM models can be formally proven.

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] OSCI, “SystemC,” 2011, available at http://www.systemc.org.

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

[4] 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.

[5] 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.

[6] 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.

[7] D. Große, H. M. Le, and R. Drechsler, “Proving transaction and system-level properties of untimed SystemC TLM designs,” in *ACM & IEEE International Conference on Formal Methods and Models for Codesign*, 2010, pp. 113–122.

[8] A. Cimatti, A. Micheli, I. Narasamdya, and M. Roveri, “Verifying SystemC : a software model checking approach,” in *Int’l Conf. on Formal Methods in CAD*, 2010, pp. 51–60.

[9] 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.

[10] 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.

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

[12] 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.

[13] C. Eisner and D. Fisman, “Augmenting a regular expression-based temporal logic with local variables,” in *Int’l Conf. on Formal Methods in CAD*, 2008, pp. 1–8.

[14] J. Long and A. Seawright, “Synthesizing SVA local variables for formal verification,” in *Design Automation Conf.*, 2007, pp. 75–80.

[15] L. Pierre and L. Ferro, “Enhancing the assertion-based verification of TLM designs with reentrancy,” in *ACM & IEEE International Conference on Formal Methods and Models for Codesign*, 2010, pp. 103–112.

[16] D. C. Black and J. Donovan, *SystemC: From the Ground Up*. Springer-Verlag New York, Inc., 2005.

[17] D. Große and R. Drechsler, *Quality-Driven SystemC Design*. Springer, 2010.