Towards Analyzing Functional Coverage in SystemC TLM Property Checking*

Hoang M. Le[1] Daniel Große[1,2] 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
{hle,grosse,drechsle}@informatik.uni-bremen.de

***Abstract*—For Electronic System Level (ESL) design SystemC has become the standard language due to its excellent support of Transaction Level Modeling (TLM). But even if the complexity of the systems can be handled using the abstraction levels offered by TLM – the most abstract one is untimed and focuses on functionality – still verification is the major bottleneck. In particular, as untimed TLM models are the reference for the following refinement steps their correctness has to be ensured. Thus, formal verification approaches have been developed to prove properties for these models. However, even if several properties have been checked this does not guarantee that the complete functionality of the TLM model has been verified. Thus, in this paper we consider the problem of functional coverage analysis in formal TLM property checking. We present a coverage approach which can analyze whether the property set unambiguously describes all transactions in a SystemC TLM model. The developed coverage analysis method identifies uncovered scenarios and hence allows to close all coverage gaps. As an example we consider an automated teller machine and we show the benefits of the proposed approach.**

I. Introduction

Transaction Level Modeling (TLM) [1] has become the key aspect for *Electronic System Level* (ESL) design. The C++-based language SystemC [2], [3] perfectly supports TLM [4] and hence is well accepted for ESL design in industry. In TLM-based design flows, a system is first modeled at a high level of abstraction using TLM with the particular focus on functionality. Hence, the initial TLM model is untimed, the communication between the system components is described by transactions and the synchronization is carried out by means of events.

However, since the TLM model serves as reference for the RTL implementation its correctness has to be ensured. Thus, formal verification approaches have been proposed to prove properties of a TLM model in a mathematical sense (see e.g. [5], [6], [7], [8]) which is a much stronger result than simulation-based methods can provide. In this context, the work in [8] introduced a *Bounded Model Checking* (BMC) [9] based verification approach to check transaction- and system-level properties of untimed TLM SystemC designs. But even if major TLM behavior has been formally proven there is no guarantee that the complete functionality has been verified. In classical RTL-based formal verification this question is well known as “have I written enough properties?” [10]. Several practical solutions showing the completeness of a property set for RTL designs have been proposed in the last years (see e.g. [11], [12], [13], [14]). However, they cannot be applied at TLM due to the different characteristics of the models regarding in particular the notion of time.

In this paper we propose the first approach for coverage analysis in TLM property checking. As mentioned, we consider untimed TLM models communicating via transactions synchronized by events. Hence, the goal of our approach is to ensure that the initiation of any transaction in a TLM model follows a certain event notification (or another transaction) which has been unambiguously described by a TLM property. Since the user is interested in coverage gaps we propose a method to identify scenarios where no property describes an implemented but unchecked transaction initiation. Applying this approach iteratively allows to close all gaps and as a result the transaction behavior of the considered SystemC TLM design is completely specified by the final property set.

Clearly, as the proposed coverage notion as well as the coverage analysis algorithm address the transaction start as the result from some computation and certain events only, the focus of this work is on the communication functionality specified in the TLM design and not on the complete functionality. However, this is a first step towards analyzing functional coverage in property checking of SystemC TLM designs.

Overall, we summarize the contributions of this paper as follows:

- Transaction-based coverage for TLM property checking
  For the first time a coverage notion is proposed to analyze whether the defined TLM properties specify all possible initiations of transactions.
- Computation of uncovered scenarios
  An algorithm to identify transactions where no TLM property checks their initiation is presented. The results can be used to reduce the number of iterations to achieve full coverage.
- Integration in verification flow
  The employed TLM property checking approach is based on BMC and the proposed coverage analysis approach is also formulated as a BMC problem. Hence, it can be easily integrated in a verification tool with only minor changes.

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

We present first experimental results demonstrating the advantages of our approach.

The rest of this paper is structured as follows: Section II introduces the preliminaries, i.e. the basics of SystemC including a running example are described and the TLM property checking approach of [8] is briefly reviewed. The proposed transaction-based coverage approach is presented in Section III. After the description of the basic idea the problem is formalized and a BMC-based method for coverage computation is described. Section IV gives the experimental evaluation. Finally, the paper is concluded in the last section.

II. Preliminaries

A. SystemC

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 have 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.

As a running example consider the SystemC TLM design in Figure 1. This is a simple but conceptually representative SystemC TLM model. It consists of two modules, each module has a SC_THREAD processes *main*. Module *m1* implements the interface *add_sub_if* (Line 8 - Line 12) and has two transactions *add* (Line 24 - Line 27) and *sub* (Line 28 - Line 31), while module *m2* implements the interface *receiver_if* (Line 3 - Line 6) and has only one transaction *receive* (Line 48 - Line 51). Note that, while in both modules, *main* is also

```
1  #include <systemc.h>
2
3  class receiver_if : virtual public sc_interface {
4  public:
5    virtual void receive(int) = 0;
6  };
7
8  class add_sub_if : virtual public sc_interface {
9  public:
10   virtual void add(int) = 0;
11   virtual void sub(int) = 0;
12 };
13
14 class m1 : public sc_module, public add_sub_if {
15 public:
16   sc_event e1;
17   int sum;
18   sc_port<receiver_if> port;
19   SC_HAS_PROCESS(m1);
20   m1(sc_module_name name) : sc_module(name) {
21     sum = 0;
22     SC_THREAD(main);
23   }
24   void add(int x) {
25     sum += x;
26     e1.notify(SC_ZERO_TIME);
27   }
28   void sub(int x) {
29     sum −= x;
30     e1.notify(SC_ZERO_TIME);
31   }
32   void main() {
33     while (true) {
34       wait(e1);
35       port−>receive(sum);
36     }
37   }
38 };
39
40 class m2 : public sc_module, public receiver_if {
41 public:
42   sc_event e2;
43   sc_port<add_sub_if> port;
44   SC_HAS_PROCESS(m2);
45   m2(sc_module_name name) : sc_module(name) {
46     SC_THREAD(main);
47   }
48   void receive(int x) {
49     cout << x << endl;
50     e2.notify(SC_ZERO_TIME);
51   }
52   void main() {
53     while (true) {
54       if (rand() % 2) port−>add(rand());
55       else port−>sub(rand());
56       wait(e2);
57     }
58   }
59 };
60
61 int sc_main (int argc , char ∗argv[]) {
62   m1 mod1(”m1”);
63   m2 mod2(”m2”);
64   mod1.port(mod2);
65   mod2.port(mod1);
66   sc_start();
67   return 0;
68 }
```

Fig. 1. Simple SystemC example

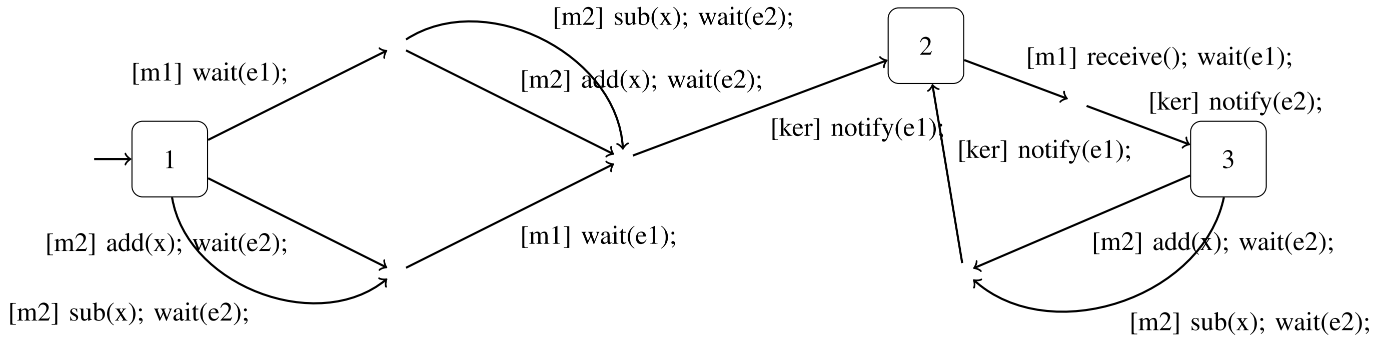

Fig. 2. The execution of the example

declared as a function, it is however not considered as a transaction because it will never be called explicitly during the execution of the design. Module *m2* can either initiate an *add* or a *sub* transaction of module *m1* (via calling the respective transaction method via the port, as example see Line 54). Both transactions deliver an integer to module *m1*, which is then added to or subtracted from a cumulative variable *sum* respectively. The value of this variable is sent back to module *m2* afterwards by the transaction *receive*. It follows that the transaction order should be either *add* then *receive* or *sub* then *receive*.

The execution of the program with respect to the simulation semantics of SystemC is illustrated in Figure 2. The states correspond to the beginning of each delta cycle and the paths between them describe the behavior during each delta cycle. In the figure [m1], [m2] and [ker] represent that process *main* of module *m1*, process *main* of module *m2* and the kernel process/scheduler currently have the control, respectively. Also note that even though the function *notify* is called inside the transactions *add*, *sub* and *receive*, the actual notification of the corresponding event is delta-delayed. There are overall four outgoing paths from state 1 to state 2. They result from the two possible schedulings for "starting" the model (either the main process of *m1* or the main process of *m2* can be scheduled first) and the non-deterministic choice between transaction *add* and *sub* in Line 54 of *m2*.

B. TLM Property Checking

In this section we briefly review the approach presented in [8] for proving properties of untimed SystemC TLM models. The *Property Specification Language* (PSL) [15] with extension of TLM primitives (begin/end of transaction, notification of event) [16] is used as the property language. In addition to simple safety properties the effect of transactions and the causal dependency between events and transactions can be checked. Sampling at different temporal resolution is also supported using PSL clock expressions, for instance at certain events only or at the begin/end of certain transactions.

The overall flow of the approach is depicted in Figure 3. First, from the SystemC TLM model, the transformed model $\mathbb{M}$ in C is generated automatically. The transformation consists of three main steps:

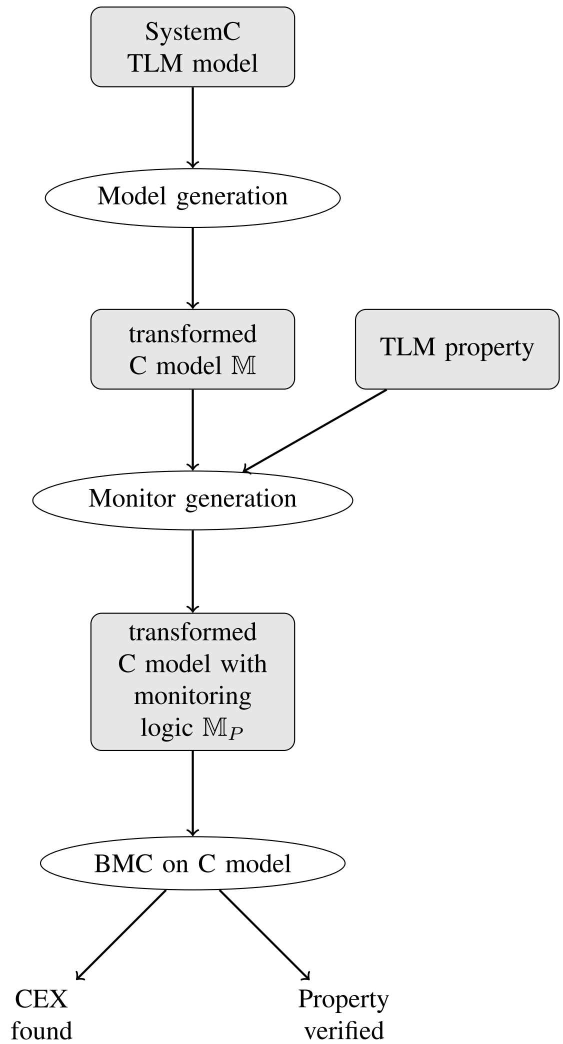

Fig. 3. Overall Flow

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.

2) The static scheduler implementing the non-preemptive simulation semantics of SystemC is generated including the delta cycle loop and the evaluation loop. 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. This allows a C model checker to explore *all interleavings* implicitly.
3) Each event gets a Boolean flag indicating whether it is notified. 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 (resume point) is inserted, to resume the execution of the corresponding process later. The handling of events is then mapped to the handling of the Boolean flags.

Dynamic process creation and dynamic memory allocation cannot be handled by the transformation yet.

After the model generation, the monitor for the TLM property is generated as a *Finite State Machine* (FSM) and this FSM is embedded into $\mathbb{M}$ in combination with assertions to form the transformed model with monitoring logic $\mathbb{M}_{\mathbb{P}}$.

For the verification task BMC can be employed by applying CBMC [17] on the C model, but an additionally proposed induction-based method gives completeness guarantee and is more efficient. The notion of states and how the transition relation is formed with respect to $\mathbb{M}_P$ is also detailed in [8]. 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 (also called the *main loop*) – which is either the evaluation loop or the delta cycle 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 $path(s_{[0..n]}) = \bigwedge_{0 \le i < n} T(s_i, s_{i+1})$.

The property P holds in the original design, iff no assertion fails 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 $safe(s_i, s_{i+1})$. 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: $\exists s_0 ... s_k. \ (I(s_0) \wedge path(s_{[0..k]}) \wedge \neg allSafe(s_{[0..k]}))$ with $allSafe(s_{[0..n]}) = \bigwedge_{0 \le i < n} safe(s_i, s_{i+1})$ and I is the characteristic predicate for all initial states.

For induction, two terminating conditions are added. 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 $loopFree(s_{[0..k]}) = path(s_{[0..k]}) \wedge \bigwedge_{0 \le i < j \le k} s_i \neq s_j$. The constraints are embedded into the transformed model by means of assumptions[1] to make induction possible directly at level of C.

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

III. Transaction-based Coverage in TLM Property Checking

After presenting the basic idea, the exact problem formulation of transaction-based coverage in TLM property checking is presented. The second part of this section introduces a BMC-based approach to analyze transaction-based coverage by detecting coverage gaps.

A. Basic Idea and Problem Formulation

Transactions are arguably the most important elements of a SystemC TLM design. Synchronized by events, transactions carry out the communication between modules of a design. With the property language and the verification method from [8], one can specify and prove properties about when a transaction should be initiated, for example after a certain event notification or after the design reaches a given internal state. However, especially for safety-critical transactions (e.g. banking operations, see Section IV), it is also very important that they can only be initiated in the intended ways. Our idea is that any transaction should not be initiated "unexpectedly", i.e. all circumstances leading to the initiation of a transaction (which are called *coverage target* or simply *target* in the following) should be fully captured by the properties. More concretely, whenever a coverage target is observed during the execution of the design, some suffix of the current execution path must satisfy at least one property from the property set.

For the exact problem formulation, we use the notion of states and transitions as defined in [8]. In the transformed C model, a state is defined as a tuple of values of the state variables at the beginning of an iteration of the main loop, which can be either the evaluation loop or the delta cycle loop of the generated SystemC scheduler. In the first case, a transition corresponds to an iteration of the evaluation loop, i.e. an execution of a process. In the second case, a transition consists of several process executions in a delta cycle. In an execution of a process, transactions can be initiated or completed and events can be notified. Thus during a transition, begin/end of transactions and notification of events can occur. We call such occurrences *attachments* to the transition.

Back to the running example, assume that our coverage target is the begin of the transaction *receive*. According to the transformation from [8], we have the following state variables: *m1_main_status, m2_main_status, e1_notified, m1_main_waiting_e1, e2_notified, m2_main_waiting_e2, m1_main_resume_point, m2_main_resume_point, sum*. Each state of the transformed model can be fully described by a tuple of values of those variables. The FSM for the transformed model is shown in Figure 4. Note that the variable *sum* has been omitted because it is not important for the communication. *Runn* and *Wait* correspond to the status *Runnable* and *Waiting* of a process. The transitions are marked with the corresponding attachments in the order of their occurrence. Note that both *e1_notified* and *e2_notified* have the value 0 in all states. The reason for that is the following: those flags are raised when the function *notify()* is

*P1: default clock = add:**exit** || sub:**exit** || receive:**entry**;*
***always** (add:**exit** −>**next** (receive:**entry**))*

*P2: default clock = add:**exit** || sub:**exit** || receive:**entry**;*
***always** (sub:**exit** −>**next** (receive:**entry**))*

Fig. 5. Properties for the example at the default temporal resolution

called and they are set to 0 again after the actual notification occurs (the black circles in Figure 4).

Now consider the property P1 from Figure 5, which means that after the transaction *add* is finished, the transaction *receive* will be initiated. Recall that our coverage target is the initiation of the transaction *receive*, thus only paths leading to *receive:entry* (the black triangle with subscript *receive*) need to be checked. Two of all four paths leading to *receive:entry* follow an end of transaction *add* and are therefore covered by the property. The two other paths are not covered. We need at least one other property to achieve the desired coverage, for example the property P2 in Figure 5.

We introduce the following notion of an *extended path*: a path $s_0 s_1 \dots s_n$ can be extended by its attachments to $s_0 att_1^{T(s_0,s_1)} \dots att_{m_1}^{T(s_0,s_1)} s_1 att_1^{T(s_1,s_2)} \dots att_{m_n}^{T(s_{n-1},s_n)} s_n$ with m_i = the number of attachments of $T(s_{i-1}, s_i)$ and $att_k^{T(s_{i-1},s_i)}$ = the k-th attachment of $T(s_{i-1}, s_i)$. Recall that a TLM property refers to a specific order of begin/end of transactions and notification of events during any execution (i.e. regardless the scheduling of processes), for example a transaction must end after a notification of a certain event. In the current context that specific order during an execution path corresponds to the order of the attachments along this path.

A path fragment $f = att_p^{T(s_{i-1},s_i)} \dots s_i \dots s_j \dots att_q^{T(s_j,s_{j+1})}$ satisfies a property P *unvacuously* (written as $P \Vdash f$) if the attachments conform strictly to the order specified by the property with $att_p^{T(s_{i-1},s_i)}$ and $att_q^{T(s_j,s_{j+1})}$ being the first and the last attachment in the order respectively. Then, the coverage problem can be formalized as follows: a single coverage target X is covered by a set of properties $\mathbb{P}$ iff $\forall n\ \forall s_0, \dots, s_n$:

$$\Big(path(s_{[0..n]}) \wedge I(s_0) \wedge \big(\exists k:\ X = att_k^T(s_{n-1}, s_n)\big)\Big)$$

$$\longrightarrow \Big(\exists i, j\ \exists P \in \mathbb{P}:\ P \Vdash att_i^{T(s_{j-1},s_j)} \dots att_k^{T(s_{n-1},s_n)}\Big)$$

Essentially, we have formalized the following: For any extended path that ends with the coverage target, there must exist at least one property from the property set which is unvacuously satisfied by a suffix of the path. Otherwise, the formula is false and the target is not covered.

On a side note, in the properties as formulated here transaction initiations are to be explicitly specified and hence non-vacuous but irrelevant properties such as *true −>**next**(true)* – which could provide a false coverage picture – are not expected.

```
...
targetCovered = false;
make possible transition on monitor 1;
if (monitor 1 reports success) targetCovered = true;
...
make possible transition on monitor n;
if (monitor n reports success) targetCovered = true;
assert(targetCovered);
occurrence of X; // inlined transaction
...
```

Fig. 6. Coverage assertion for a target X

B. Coverage Analysis

In the following we show how to analyze the coverage of a single target. The main idea is to translate the coverage formulation into assertions that are embedded in the transformed C model. After that the induction-based verification method from [8] can be applied. A target is either proven to be covered or we get a counter-example, which represents an uncovered scenario. Recall that in [8] the monitoring logic of a property is generated as a FSM and is embedded in the transformed model in combination with assertions. The code for transitions and the assertions are inserted at the beginning/end of transactions or at notification of events, which are the sampling points of the property. The property is satisfied by the current execution path if there has been no assertion failure so far. However, in the coverage formulation, at least one property is required to be *unvacuously* satisfied at any occurrence of the target. Recall that each state of the FSM of a property indicates that a certain prefix of the specified order has been observed so far. Thus, it follows that a property is unvacuously satisfied iff the FSM is just brought to the initial state from the state corresponding to the observation of the specified order. The translation for the coverage analysis works as follows:

1) For each property in the property set, the monitoring logic (FSM) is generated and embedded into the transformed model.
2) A new Boolean variable *targetCovered* is introduced into the model.
3) This variable is set to *false* before the monitoring logic code at each occurrence of the coverage target X (i.e. the inlined call of transaction X, since in this paper only the initiation of a transaction is considered as a target).
4) The code for the monitors is subsequently modified, so that if any of the monitors reports success (i.e. the corresponding property is unvacuously satisfied), *targetCovered* is set to *true* indicating that the target is covered.
5) Finally, before each occurrence of the coverage target, we need to insert a *coverage assertion* stating that *targetCovered* is true.

The pseudo-code in Figure 6 shows the result of the translation for the general case.

Total coverage in terms of our approach is achieved by considering all single targets of a design. If all those trans-

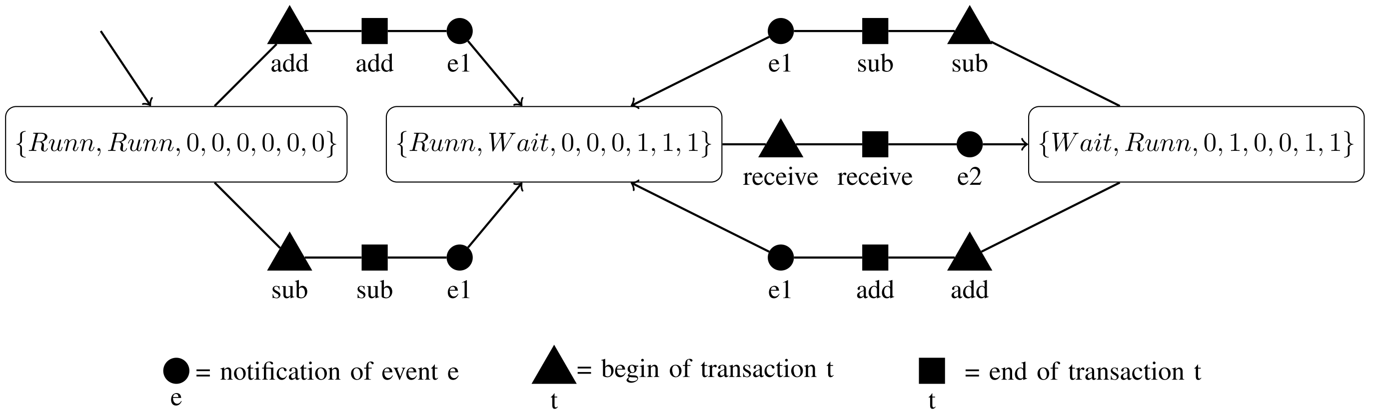

Fig. 4. States and transitions of the transformed example

actions and events are successfully proven to be covered by the properties then it can be concluded that the transaction behavior of the design is fully specified.

IV. Experimental Evaluation

In this section, first results are presented and discussed. We have integrated the proposed approach in the verification framework of [8]. Our benchmark is a medium-sized TLM skeletal implementation of an *Automated Teller Machine* (ATM). The experiments have been carried out on a 3GHz Intel Xeon system with 4 GB RAM running Linux and Boolector v1.2 [18] is used as the underlying SMT solver for CBMC v3.3 [17].

First, we give a short description of the ATM. Then the verification of the design is discussed and finally the results of the coverage analysis are given.

A. ATM

The ATM has approximately 150 lines of code and two modules: a front-end receiving the inputs from user/environment and a back-end processing those requests. Each module has a main SC_THREAD process. The main process of the front-end waits repeatedly for an event *input_request* and then transports the input to the back-end through a transaction. This transaction also notifies the event *input_received* and as a consequence, the main process of the back-end is woken up to process the input and subsequently informs the front-end that new input is needed by initiating a transaction that issues a notification of *input_request*.

The main functionality of the ATM is described by the FSM in Figure 7. The design accepts five possible inputs: RESET, INC, TAKE, OP, VAL and has five internal states: INIT, CARD_IN, CARD_OUT, CODE_CHECK, CODE_OK. Whenever the input RESET is received, the card is ejected and the state of the design will be set to CARD_OUT. The input INC (TAKE) is activated if the card is inserted into the ATM (withdrawn from the ATM). Then, the state is also changed from INIT to CARD_IN (from CARD_OUT to INIT), respectively. The input OP represents a banking operation and the first time OP is issued, the customer is requested to give the PIN through the input VAL. If the customer gives the correct code before a number of attempts MAX_TRY, the requested banking operation is performed and the state CODE_OK is reached and henceforward banking operations are allowed without asking for PIN. For a banking operation, a transaction *do_op* will be initiated. Otherwise the card is kept and the initial state INIT is reached.

B. Verification

In the following we focus on the verification and coverage of the representative banking operation OP. For the verification purpose, the three properties depicted in Figure 8 have been formulated. The first two properties P4 and P5 specify that the transaction *do_op* will be initiated if the customer has entered the correct code (ival == ok) in the first or in the second attempt, respectively. The property P6 specifies the case that the correct code has been given already (i.e. the design is in state CODE_OK) and thus the transition *do_op* can be immediately initiated upon request. All properties were successfully proven to hold using the induction-based method from [8] in less than 60 seconds.

C. Coverage Analysis

However, the coverage of the target *do_op:**entry*** cannot be proven with P4, P5 and P6. The resulting error trace/uncovered scenario consists of a sequence of three validation attempts whereas the last one is successful and the transaction *do_op* is consequently initiated. The scenario reveals a gap in the property set: It has not been specified what happens when more than two validation attempts have failed. There are now two possible interpretations of the coverage result. In our case, by analyzing the uncovered scenario we find a "off-by-one bug" in the implementation ($<=$ instead of $<$ has been written in the SystemC TLM model). After fixing this bug, the coverage for *do_op:**entry*** can be proven successfully in about 180 seconds. The other case would be that the specification actually allows up to three validation attempts. We would need to include a new property describing this unchecked behavior and hence if no other gap remains with the new set of properties the target would be covered.

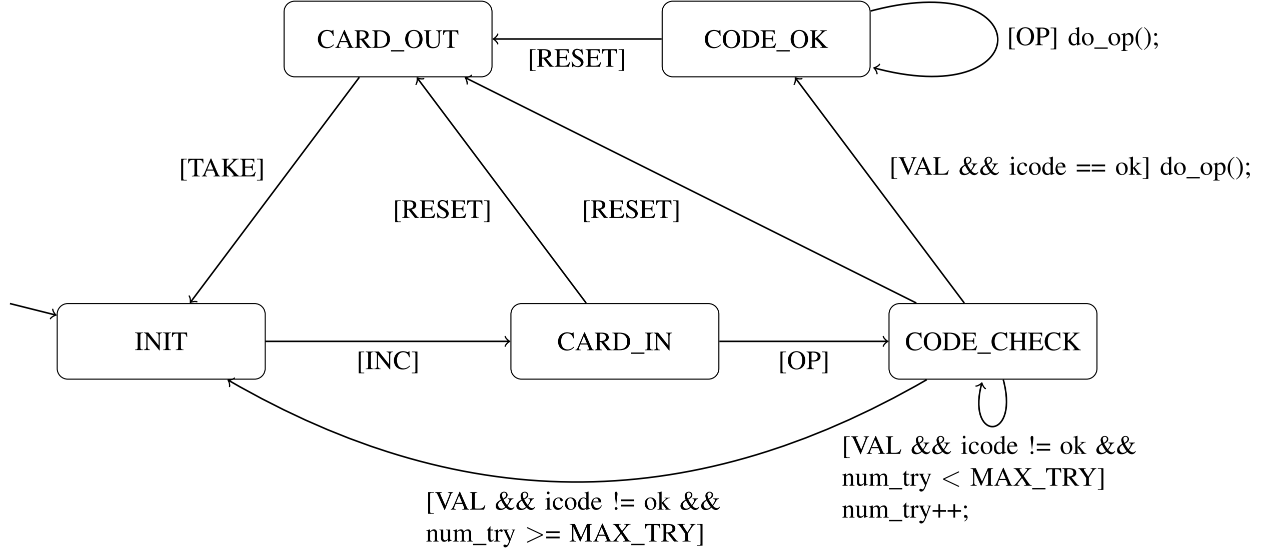

Fig. 7. FSM of the ATM

*P4: default clock = input_received.***notified** *|| do_op:***entry***;*
always *((input_received.***notified** *&& ((state == CARD_IN) && (icode == OP)))*
*−> (***next** *((input_received.***notified** *&& ((icode == VAL) && (ival == ok)))*
*−> (***next** *(do_op:***entry***)))))*

*P5: default clock = input_received.***notified** *|| do_op:***entry***;*
always *((input_received.***notified** *&& ((state == CARD_IN) && (icode == OP)))*
*−> (***next** *((input_received.***notified** *&& ((icode == VAL) && (ival != ok)))*
*−> (***next** *((input_received.***notified** *&& ((icode == VAL) && (ival == ok)))*
*−> (***next** *(do_op:***entry***)))))))*

*P6: default clock = input_received.***notified** *|| do_op:***entry***;*
always *((input_received.***notified** *&& ((state == CODE_OK) && (icode == OP)))*
*−> (***next** *(do_op:***entry***)))*

Fig. 8. Properties for the ATM

V. Conclusions

We have presented a transaction-based coverage notion for property checking of untimed SystemC TLM designs. This is to the best of our knowledge the first formal coverage notion proposed for SystemC TLM. The notion enables to analyze whether all possible ways to initiate a transaction in the implementation are also described by the TLM property set. The coverage analysis is then formulated as a BMC problem allowing an easy integration into the existing BMC-based TLM property checking approach introduced in [8]. The C model with monitoring logic generated for property checking is modified and augmented by a coverage assertion. The coverage analysis is reduced to verify a C program with assertions in the same manner as the property checking problem. The BMC formulation also enables a very important feature: the efficient computation of uncovered scenarios (i.e. counter-examples to the coverage analysis) providing valuable feedback. By analyzing the traces, the designer is pinpointed to behavior which has not been checked by a property. This behavior is either correct and hence he/she has to add an appropriate property to achieve the desired coverage or a bug has been discovered.

This work with its focus on communication functionality is the first step towards the formal analysis of complete functional coverage for SystemC TLM designs. For future work we want to extend our coverage notion such that the data transferred by a transaction and the effect of a transaction on the design is also captured. For example, after the end of a transaction all possible following states of the design should be fully described by the property set.

REFERENCES

[1] L. Cai and D. Gajski, “Transaction level modeling: an overview,” in *IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, 2003, pp. 19–24.

[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] F. Ghenassia, *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems*. Springer, 2006.

[5] 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.

[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] H. Garavel, C. Helmstetter, O. Ponsini, and W. Serwe, “Verification of an industrial SystemC/TLM model using LOTOS and CADP,” in *ACM & IEEE International Conference on Formal Methods and Models for Codesign*, 2009, pp. 46–55.

[8] 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.

[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] S. Katz and O. Grumberg, “Have I written enough properties - a method of comparison between specification and implementation,” in *Correct Hardware Design and Verification Methods*, 1999, pp. 280–297.

[11] J. Bormann, S. Beyer, A. Maggiore, M. Siegel, S. Skalberg, T. Blackmore, and F. Bruno, “Complete formal verification of Tricore2 and other processors,” in *Design and Verification Conference (DVCon)*, 2007.

[12] A. Fedeli, F. Fummi, and G. Pravadelli, “Properties incompleteness evaluation by functional verification,” *IEEE Trans. on Comp.*, vol. 56, no. 4, pp. 528–544, 2007.

[13] K. Claessen, “A coverage analysis for safety property lists,” in *Int’l Conf. on Formal Methods in CAD*, 2007, pp. 139–145.

[14] D. Große, U. Kühne, and R. Drechsler, “Analyzing functional coverage in bounded model checking,” *IEEE Trans. on CAD*, vol. 27, no. 7, pp. 1305–1314, 2008.

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

[16] 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.

[17] 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.

[18] 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.