Automatic Fault Localization for SystemC TLM Designs*

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*—To meet today's time-to-market demands catching bugs as early as possible during the design of a system is absolutely essential. In Electronic System Level (ESL) design where SystemC has become the de-facto standard due to Transaction Level Modeling (TLM), many approaches for verification have been developed. They determine an error trace which demonstrates the difference between the required and the actual behavior of the system. However, the subsequent debugging process is very time-consuming, in particular due to TLM-related faults caused by complex process synchronization and concurrency.**

In this paper, we present an automatic fault localization approach for SystemC TLM designs. The approach determines components that can be changed such that the intended behavior of the design is obtained removing the contradiction given by the error trace. Techniques based on Bounded Model Checking (BMC) are used to find the components. We demonstrate the quality of our approach by experimental results.

I. Introduction

The next level of abstraction beyond RTL is clearly *Transaction Level Modeling* (TLM) [1]. TLM allows to describe the communication in a system in terms of abstract operations (transactions) rather than assignments to low level signals or wires. For TLM, SystemC [2], [3] has become the most widely accepted language; meanwhile all big EDA companies offer *Electronic System Level* (ESL) design solutions based on SystemC. This is mainly due to the SystemC TLM standard enabling the development of reusable and interoperable *Intellectual Property* (IP). As a C++ class library, SystemC allows to model hardware and software in one model. Furthermore, SystemC TLM enables a simulation performance which is orders of magnitudes faster in comparison to RTL.

In the last years a substantial body of academic and industrial progress in methods for SystemC TLM has been made. For instance, these include design methodologies [4], algorithms for design space exploration [5], [6] and verification approaches [7], [8], [9], [10], [11], [12], [13]. However, the existing debugging solutions for SystemC TLM have serious limitations (for a detailed discussion we refer to the related work section). This is a problematic issue since the event-based communication and process synchronization at TLM as well as the concurrency makes SystemC debugging extremely challenging.

In this paper, we propose a new approach for debugging of SystemC TLM designs. In general, for debugging three steps are necessary: (1) fault detection (an error trace, or a counter-example demonstrating the difference between the actual and expected behavior), (2) fault localization (finding the incorrect component(s)) and (3) fault correction.

This work targets the problem of fault localization for SystemC TLM designs extending the concepts of [14]. More precisely, we devise an automatic approach to determine components that can be changed such that the intended behavior of the design is obtained removing the contradiction given by the error trace. In contrast to [14], components in our case are not only "simple" program expressions, but TLM specific parts of a SystemC model which are typically prone to design errors. For instance, erroneously a blocking transaction is used instead of a non-blocking transaction or the wrong event is notified or waited for, etc.

The overall flow of our approach is as follows: We assume that an error trace demonstrating the incorrect behavior with respect to the specification is given. Then, the considered SystemC TLM design is instrumented by adding *abnormal predicates* [14] which control the change of a component. That is, a new variable is introduced and tested in an if-statement such that the component behavior is either unaltered or changed based on the above mentioned typical TLM faults. Then, a formal model in C is generated from the instrumented design using [12]. This model is fixed to the input values and the process schedule according to the error trace. Also, the model execution is constrained to be compliant to the specification. Now, CBMC (*Bounded Model Checking* for C programs [15]) is employed on the final C model to search for an execution trace. If such a trace exists, we extract a diagnosis based on the abnormal predicates. This diagnosis identifies faulty components and possible changes removing the contradiction given by the error trace.

The remainder of this paper is structured as follows: In Section II related work is discussed. Section III introduces the preliminaries, i.e. the basics of SystemC including a running example are described. Furthermore, the TLM property checking approach [12] which is used for formal model generation is reviewed. The proposed fault localization approach for SystemC TLM designs is presented in Section IV. Section V gives experimental results. Finally, the paper is concluded in the last section.

II. Related Work

The simplest form of debugging SystemC TLM models is to use transaction recording and then printing the logged information. Since this procedure gives poor results only, alternatives have been developed. In [16] a SystemC-aware extension for the GDB debugger has been presented. This approach enhances the observability in comparison to a standard software debugger but does not compute a reason for a failure.

In [17] an approach for SystemC debugging is presented which improves the understanding of a failure. It is based on delta debugging [18] which aims to isolate the failure cause by narrowing down the difference between a passing and failing test case. This technique is used in [17] for debugging of process schedules. However, it does not identify possible fault locations. Moreover, only a small fraction of the 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.

```
1  class receiver_if : virtual public sc_interface {
2  public:
3    virtual void receive(unsigned int) = 0;
4  };
5
6  class slave_if : virtual public sc_interface {
7  public:
8    virtual void add(unsigned int) = 0;
9    virtual void sub(unsigned int) = 0;
10 };
11
12 class sender : public sc_module {
13 public:
14   sc_port<receiver_if> port;
15   unsigned int v;
16   SC_HAS_PROCESS(sender);
17   sender(sc_module_name name) :
18     sc_module(name) {
19       v = rand();
20       SC_THREAD(main); }
21
22   void main() {
23     while (true) {
24       port->receive(v);
25       v++;
26     }
27   }
28 };
```

```
29 class receiver : public receiver_if, public sc_module {
30 public:
31   sc_event done_receiving;
32   sc_event done_processing;
33   sc_port<slave_if> port;
34   unsigned int data;
35   SC_HAS_PROCESS(receiver);
36   receiver(sc_module_name name) :
37     sc_module(name) {
38       SC_THREAD(main); }
39
40   void receive(unsigned int x) {
41     data = x;
42     done_receiving.notify();
43     wait(done_processing);
44   }
45
46   void main() {
47     while (true) {
48       wait(done_receiving);
49       if (data != 0) port->add(data);
50       done_processing.notify();
51     }
52   }
53 };
```

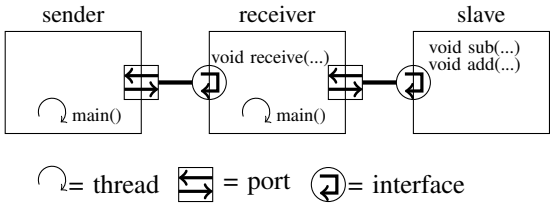

Fig. 1. SystemC TLM example consisting of the three modules: sender, receiver, and slave

specific faults as considered in this work can be handled by the approach.

For debugging deadlocks specialized methods have been developed. In [19] the SystemC simulation is monitored and based on a synchronization dependency graph deadlocks are reported. The work in [20] introduces a simulation-based method which modifies the SystemC scheduler such that wait/notify dependencies and possible cycles resulting in deadlocks can be found. Improvements using partial order reduction have been published in [21]. For race analysis a combination of static analysis and model checking can be found in [22]. Please note that all these approaches help in debugging but do not report a diagnosis.

For sequential circuits, a diagnosis approach based on *Boolean Satisfiability* (SAT) has been proposed in [23]. Multiplexers are inserted into the circuit enabling non-deterministic replacements of gates. The approach determines fault candidates which fix the counter-examples. In [24] fault localization for property checking has been presented.

As already mentioned above, we extend the fault localization approach for C programs proposed in [14]. In this work, we aim fault localization of SystemC TLM designs. Furthermore, we consider dedicated fault models for these designs and target concurrency of the TLM models on top of the SystemC scheduler.

III. Preliminaries

A. SystemC

In the following only the essential aspects of SystemC are described. 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 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 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 the earliest one, the waiting processes are made runnable, and it is continued with Step 2. Otherwise the simulation is stopped.

As a running example we use the SystemC design shown in Fig. 1. It consists of three modules: one sender, one receiver and one slave. The sender can initiate a transaction that sends an unsigned integer to the receiver by calling its *receive* method through a port (Line 24, see also the graphical representation on the right-hand side of Fig. 1). The sender has one SC_THREAD which repeatedly initiates this transaction. The transaction is synchronized in the receiver by two events *done_receiving* and *done_processing* as follows. When the integer is received, the event *done_receiving* will be notified (immediate notification at Line 42) and *receive* will be blocked until the event *done_processing* is notified (respective *wait* see Line 43). This notification is issued by the SC_THREAD *main* of the receiver (immediate notification in Line 50) after it is waken up by the notification of *done_receiving* and the processing is done, i.e. the transaction *add* of the slave module has finished.

B. Bounded Model Checking for SystemC TLM

In this section we briefly review the approach presented in [12] for proving properties of SystemC TLM models. The *Property Specification Language* (PSL) [25] with extension of TLM primitives (begin/end of transaction, notification of event) [26] is used as the property language. In addition to

```
1  while (runnable_count > 0) { // time loop
2    while (runnable_count > 0) { // delta cycle loop
3      while (runnable_count > 0) { // evaluation loop
4          choose_one_runnable_process();
5          runnable_count−−;
6          if (process 1 is chosen) {
7              process_1_status = RUNNING;
8              process_1();
9          }
10         ...
11         if (process n is chosen) {
12             process_n_status = RUNNING;
13             process_n();
14         }
15     }
16     // delta notification
17     ...
18   }
19   // timed notification
20   ...
21 }
```

Fig. 2. Generated SystemC scheduler

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.

Summarized, the bounded model checking approach for SystemC TLM designs 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.
2) The static scheduler implementing the non-preemptive simulation semantics of SystemC is generated. The scheduler skeleton is illustrated in Fig. 2. 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 4 in Fig. 2). This allows a C model checker to explore *all interleavings* implicitly.
3) Each event gets a Boolean flag 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 (resume point) is inserted, to resume the execution of the corresponding process later. The handling of events is then mapped to the handling of those variables.

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

For the verification task CBMC [15] 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 [12]. 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*) 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 $path(s_{[0..n]}) = \bigwedge_{0 \le i < n} T(s_i, s_{i+1})$.

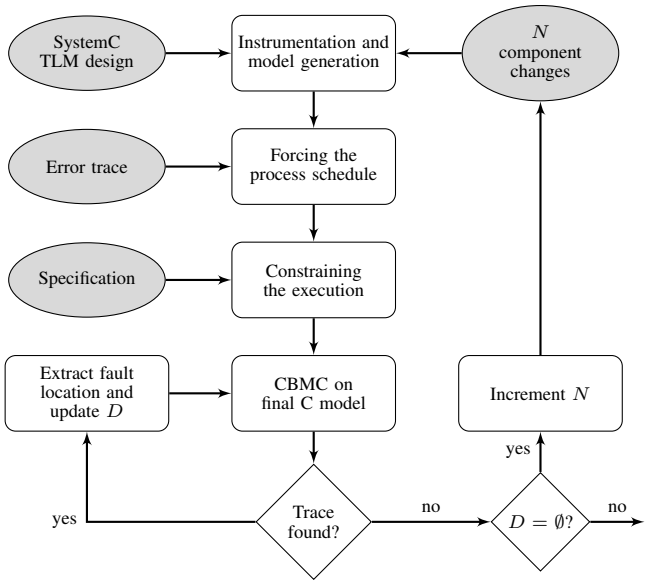

Fig. 3. Overall flow for fault localization

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

IV. Fault Localization of SystemC TLM designs

We assume that an error trace showing the difference between the specification and the SystemC TLM design is given. Such a trace can be obtained by simulation or formal verification.

The general flow of the proposed fault localization approach for SystemC TLM designs is depicted in Fig. 3 and consists of the following four main steps:

1) The SystemC TLM design is instrumented by adding abnormal predicates which control the change of the components. There is also a parameter *N* which limits the number of components that can be changed. We start with *N = 1* to perform single fault diagnosis at first.
2) The inputs, the non-deterministic variable choices and the scheduling sequence are fixed to the values given by the error trace.
3) The execution of the instrumented design is constrained to traces that are compliant with the specification. Traces that can reach the end of the execution, do not violate the specification. Those traces contain *N* changed components which eliminate the faulty behavior demonstrated by the error trace.
4) The C Bounded Model Checker CBMC [15] is employed to search for such a trace. If it exists, we extract a diagnosis based on the abnormal predicates. This diagnosis identifies faulty components and possible changes removing the contradiction given by the error trace. We

iterate this process to compute the set D of all diagnoses. If it is not possible to find such a trace (i.e. $D = \emptyset$), we increase the value of *N* and go back to Step 1 to search for multiple faults.

We use the running example introduced in Section III-A to explain our approach in more detail. For this example, the following scenario is considered: According to the specification the TLM property

always *(done_receiving.**notified*** $->$ ***next*** *((data != 0)$->$sub:**entry**))*

has been formulated, stating that "after the event *done_receiving* has been notified, the transaction *sub* should start if the value of *data* is not equal to 0". Obviously, the incorrect transaction *add* is called instead by the design.

With the example and the scenario at hand, we describe the ingredients of our approach and the flow as follows:

a) Error traces: In our context, an error trace is defined as an execution of the SystemC TLM model that leads to an assertion violation. Thus, an error trace of a SystemC TLM design consists not only of the inputs to the design, but also the non-deterministic choices made by the SystemC scheduler need to be taken into account. In particular, the error trace defines which runnable process is started in each evaluation loop iteration (see also Fig. 2, Line 4).

In the running example, the error trace leading to the property violation contains the input value *v* = 0 and the scheduling sequence *receiver_main*, *sender_main*, *receiver_main*, *sender_main*, *receiver_main* (*receiver_main*/*sender_main* denotes the SC_THREAD *main* of the receiver/sender).

b) Components: Since a SystemC TLM designs heavily relies on complex communication mechanisms, they often cause errors. These mechanisms are based on transactions and the synchronization is carried out using events. We propose to use both as components for fault localization. In the following, we describe our set of components and how to change them to avoid the error. We also show the relation of the components to the common faults found in SystemC TLM designs (similar observations have been reported in [27]).

- **Transaction**: one of the most common faults is to use a non-blocking instead of a blocking transaction and vice versa, or generally calling a similar but incorrect function, which expect the same set of parameters. Such a fault can be detected by replacing a function call with another function within the scope and with the same function signature.
- **Transaction data**: transporting incorrect transaction data can also cause the design to malfunction. Function parameter can be changed to a non-deterministic value to detect those errors in the same way as expression debugging of C programs.
- **Concurrent function**: the wrong use of concurrent functions, e.g. using untimed instead of timed constructs or using the wrong parameter for wait/notify may result in a deadlock. Therefore, we allow the change of time parameter for wait/notify and the exchange of untimed and timed wait/notify.
- **Event**: waiting for or notifying an incorrect event respectively, can also cause faulty behavior. Such an error can be modeled by replacing the event used in wait/notify by another event within the scope.

```
1   int diag = nondet_uint();
2   ...
3   class receiver : public receiver_if, public sc_module {
4   public:
5     ...
6     void receive(unsigned int x) {
7       data = x;
8       if (diag == 1) done_processing.notify();
9       else if (diag = 2) done_receiving.notify(nondet_uint());
10            else done_receiving.notify();
11      if (diag == 3) wait(done_receiving);
12      else if (diag == 4) wait(done_processing, nondet_uint());
13           else wait(done_processing);
14    }
15
16    void main() {
17      while (true) {
18        ...
19        if (data != 0) {
20          if (diag == 7) port->sub(data);
21          else if (diag == 8) port->add(nondet_uint());
22               else port->add(data);
23        }
24        ...
25      }
26    }
27  };
```

Fig. 4. Instrumented SystemC example

In addition, we can also add the expressions (right-hand sides of assignments, the conditions of if, while, and case statements, etc.) in the design to the set of components.

For our example, the components for fault localization are the parameter *x* and *data* of the transaction *receive* and *add* respectively, both events *done_receiving* and *done_processing*, the calls of *wait* and *notify* and the transaction *add*, since the slave also offers the function *sub* with the same signature.

c) Instrumentation: The general procedure for instrumentation of a SystemC TLM design is based on the replacement of a C++ statement by a new statement. This new statement consists of several of if/else blocks which model the possible changes of the component as described above. For instance, the event notification statement "e1.notify()" is replaced by

```
if (diag == i ) e2.notify();
else if (diag = i + 1) e1.notify(nondet_uint());
     else e1.notify();
```

where *diag* is a new variable inserted at the beginning of the model and i is a unique integer. Note that the value of *diag* will be chosen non-deterministically by the model checker. As can be seen the immediate notification *e1.notify()* can be changed to an immediate notification of another event, here *e2*, in the case when *diag* equals i, or a delta notification (timed notification) of *e1* using the argument 0 (greater than 0) in the case *diag* equals $i + 1$.

Parts of the instrumented design of the running example are depicted in Fig. 4. In Line 8 the instrumentation for the notification of *done_receiving* is shown. This follows exactly the example described in the previous paragraph. Then, Line 11 gives the replacement of the wait for the event *done_processing*. Finally, in Line 20 the replacement for the initiated transaction *add* is listed. As can be seen, the *sub* transaction may be called instead, a non-deterministic value for the transaction argument may be used, or the behavior is unaltered.

d) Forcing the process schedule: The inputs of the design can be set according to the trace without any problem. However, forcing the process schedule requires modifications of the generated scheduler. Those modifications are challenging to implement efficiently with the scheduler shown in Fig. 2.

```
1  while (runnable_count > 0) { // evaluation loop
2    choose_one_runnable_process();
3    runnable_count−−;
4    if (process 1 is chosen) {
5      process_1_status = RUNNING;
6      process_1();
7    }
8    ...
9    if (process n is chosen) {
10     process_n_status = RUNNING;
11     process_n();
12   }
13   if (runnable_count == 0) {
14     // delta notification
15     ...
16   }
17   if (runnable_count == 0) {
18     // timed notification
19     ...
20   }
21 }
```

Fig. 5. New implementation of the scheduler

The main reason is that the scheduler consists of three nested loops which have to be unwound adequately for CBMC to check all possible executions.

Due to page limitation, we cannot discuss this problem in detail. We propose instead an equivalent scheduler with only one loop. The new scheduler shown in Fig. 5 has been derived by exploiting the fact that the nested scheduler loops in Fig. 2 share the same loop condition runnable_count > 0. The single loop of the new scheduler corresponds to the evaluation loop. The preservation of the simulation semantics can be explained as follows using Fig. 5: If Line 14 is reached, it means there is no more runnable process, thus the current evaluation loop iteration is finished and the delta notification phase is entered. If we have at least one runnable process (runnable_count > 0) after this phase, the timed notification phase (Line 18) is skipped and the execution continues with a new evaluation loop iteration. Otherwise (runnable_count $== 0$), the current delta cycle is over and therefore the timed notification phase can start (Line 18). If this phase makes at least one process runnable, a new evaluation loop iteration starts. Otherwise, the loop condition on Line 1 fails and the simulation stops.

With the new scheduler the process schedule can be forced easily according to the error trace: we unwind the evaluation loop and keep only the corresponding scheduled process in each unwound iteration. For the running example, the evaluation loop needs to be unwound five times, because the process schedule has five elements. In the first iteration, we execute *receiver_main*, in the second iteration, *sender_main* and so on. The first two unwound iterations are shown in Fig. 6.

e) Constrained execution and CBMC: We constrain the execution of the C model of the instrumented design to traces that are compliant with the specification. If the specification is given as a TLM property we simply convert all assertions (which are part of the monitoring logic of the corresponding FSM) to assumptions. After that, the assertion *assert(false)* is added to the execution point where the finite error trace ends. Then, we can run CBMC on the final C model thereby following [12]. From a compliant trace, we extract the value of *diag* providing us the to be changed component as well as the change. We add this to the set D of all diagnoses. Furthermore, we constrain the design such that in the next iteration *diag* cannot get the same value again. We repeat the process until no more traces can be found. If no trace has been found (i.e. $D = \emptyset$), we need to increase N to look for multiple

```
1  //
2  // first unwound iteration
3  runnable_count−−;
4  receiver_main_status = RUNNING;
5  receiver_main(); // first scheduled process
6  if (runnable_count == 0) {
7    // delta notification
8    ...
9  }
10 if (runnable_count == 0) {
11   // timed notification
12   ...
13 }
14 //
15 // second unwound iteration
16 runnable_count−−;
17 sender_main_status = RUNNING;
18 sender_main(); // second scheduled process
19 if (runnable_count == 0) {
20   // delta notification
21   ...
22 }
23 if (runnable_count == 0) {
24   // timed notification
25   ...
26 }
27 ...
```

Fig. 6. Forcing the process schedule of the running example

faults. For the instrumentation phase this basically means that we extend *diag* to a N-dimensional vector and modify the if-conditions respectively.

For the running example, our approach provides only one diagnosis, where the variable *diag* is assigned to 7. The diagnosis means that if the transaction *sub* is initiated instead of *add*, the property violation is eliminated. This is indeed a correct bug-fix for the design.

V. Experiments

In this section the experimental results are presented and discussed. All experiments have been carried out on a 3 GHz Intel Xeon system with 4 GB RAM running Linux. Furthermore, CBMC v3.6 [15] has been used.

A. Sender receiver TLM design

In comparison to the running example, the functionality of the sender receiver TLM design is much more complex. The sender has a memory addressed from 0 to 1023 which is modeled as an array of 1024 integers. The sender can initiate a transaction *send(x,y)* which sends the content of the memory from address *x* to address *y* (exclusive) to the receiver. The data is transported to the receiver through a FIFO of size 32. The FIFO implementation has been taken from the official SystemC distribution. Both transactions *read* and *write* of the FIFO are blocking transactions. We have extended the implementation to include non-blocking *read* and *write*. At the start of the transaction *send*, the sender initiates another transaction *send_num* that delivers the number *M* of integers to be sent directly to the receiver (*M* = *y* - *x*) . The receiver will then try to read *M* integers from the FIFO. The transaction *send* is blocked by waiting for an event until the receiver finishes reading from the FIFO and notifies the event.

We show the quality of our approach on three slightly modified designs denoted as *SR1*, *SR2*, *SR3* in the following. Each of those designs contains one bug. A summary is provided in Table I. The first column gives the length L of the error trace measured in terms of states as report by CBMC. The second column shows the number of diagnoses and the last column provided the total run-time to compute all diagnoses.

TABLE I
RESULTS OF THE SENDER RECEIVER TLM DESIGN

Design	L	$\|D\|$	Time
SR1	1880	1	84.65s
SR2	2264	3	92.31s
SR3	1332	3	81.46s

TABLE II
RESULTS OF JPEG ENCODER DESIGN

Design	L	$\|D\|$	Time
JPEG1	1091	1	13.17s
JPEG2	1778	1	77.28s

Due to page limitation, we only discuss two of the experiments in more detail. In the design *SR1*, the incorrect value of $M = y - x + 1$ is delivered to the receiver by the transaction *send_num* because the address range from x to y is erroneously assumed to consist of all address values including the address y. This bug can be classified into the category *incorrect transaction data*. It causes the design to deadlock, because the receiver is blocked while trying to read from the empty FIFO. Our approach is able to localize the fault accurately. CBMC gives us only one diagnosis, where the value of *diag* corresponds to the faulty component *send_num*. Furthermore, the value of M has also been lower than $y - x + 1$ in the diagnosis, indicating that this value M needs to be reduced.

The bug in *SR2* is to use the non-blocking transaction *write* of the FIFO in the sender. Because of the circular behavior of the FIFO, if more than 32 (the FIFO size) integers are sent to the receiver, only the last 32 ones are kept. This incorrect behavior is detected by checking the property P "The first integer read from the FIFO should be equal to the value of the memory cell with address x at the start of the transaction *send(x, y)*". Our approach determines a total of three diagnoses. The first diagnosis identifies *send_num* as the faulty component and suggests to repair the bug by assigning a negative argument to *send_num* which is useless. The other two diagnoses identify the faulty component, the non-blocking *write*, accurately. Under the assumption that the *first* of the 32 integers kept in the FIFO was put by the k-th *write* transaction, one diagnosis tries to repair the error by changing the argument of this transaction to match the asserted value. This diagnosis can also be ignored. The other gives the correct repair that changes the non-blocking *write* to its blocking variant.

B. JPEG Encoder

The second design considered is the TLM implementation of a part of a JPEG encoder taken from [12]. The design consists of eight modules communicating through a bus. We consider two variants of the design denoted as *JPEG1* and *JPEG2*. The first (second) design contains a *write* (*read*) transaction with an incorrect bus address computation. In both cases, our approach was able to identify the faulty component accurately. The results are provided in Table II with the same table layout as Table I.

C. Summary

For all designs, the run-time to determine the fault locations was very short. In all cases, the approach was able to find the correct diagnosis and repair. For *SR2* and *SR3*, a few spurious diagnoses have also been found, but with nearly no effort, they have been ruled out.

VI. CONCLUSIONS

We have presented an automatic fault localization approach for SystemC TLM designs. The approach targets in particular typical TLM faults like e.g. the call of a blocking transaction instead of a non-blocking one (and vice versa) or the initiation of the wrong transaction as well as erroneous process synchronization. These faults are modeled in the SystemC TLM design by integrating abnormal predicates such that the corresponding components can be changed. Then, formal methods based on BMC are employed to determine possible fault locations. Among the source code position to be changed they also include a change or possible values such that the intended behavior of the design is obtained removing the contradiction given by the error trace.

For different SystemC TLM examples we have shown that our approach finds the fault locations very fast. If more than one fault locations result, the spurious cases can be identified quickly by the user. Thus, the debugging process is significantly accelerated by our approach.

REFERENCES

[1] L. Cai and D. Gajski, "Transaction level modeling: an overview," in *CODES+ISSS*, 2003, pp. 19–24.
[2] OSCI, "SystemC," 2010, available at 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] S. Boukhechem and E. Bourennane, "TLM platform based on SystemC for starsoc design space exploration," *Adaptive Hardware and Systems, NASA/ESA Conference on*, pp. 354–361, 2008.
[6] N. Bombieri, F. Fummi, and D. Quaglia, "System/network design-space exploration based on TLM for networked embedded systems," *ACM Trans. Embed. Comput. Syst.*, vol. 9, no. 4, pp. 1–32, 2010.
[7] W. Ecker, V. Esen, T. Steininger, M. Velten, and M. Hull, "Interactive presentation: Implementation of a transaction level assertion framework in SystemC," in *DATE*, 2007, pp. 894–899.
[8] N. Bombieri, F. Fummi, and G. Pravadelli, "Incremental ABV for functional validation of TL-to-RTL design refinement," in *DATE*, 2007, pp. 882–887.
[9] M. Y. Vardi, "Formal techniques for SystemC verification," in *DAC*, 2007, pp. 188–192.
[10] L. Ferro and L. Pierre, "Formal semantics for PSL modeling layer and application to the verification of transactional models," in *DATE*, 2010, pp. 1207–1212.
[11] D. Große and R. Drechsler, *Quality-Driven SystemC Design*. Springer, 2010.
[12] D. Große, H. M. Le, and R. Drechsler, "Proving transaction and system-level properties of untimed SystemC TLM designs," in *MEMOCODE*, 2010, pp. 113–122.
[13] H. M. Le, D. Große, and R. Drechsler, "Towards analyzing functional coverage in SystemC TLM property checking," in *HLDVT*, 2010, pp. 67–74.
[14] A. Griesmayer, S. Staber, and R. Bloem, "Automated fault localization for C programs," *Electr. Notes Theor. Comput. Sci.*, vol. 174, no. 4, pp. 95–111, 2007.
[15] E. M. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *TACAS*, 2004, pp. 168–176.
[16] F. Rogin, C. Genz, R. Drechsler, and S. Rülke, "An integrated SystemC debugging environment," in *FDL*, 2007, pp. 140–145.
[17] F. Rogin, R. Drechsler, and S. Rülke, "Automatic debugging of system-on-a-chip designs," in *IEEE International SOC Conference*, 2009, pp. 333–336.
[18] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Trans. Software Eng.*, vol. 28, no. 2, pp. 183–200, 2002.
[19] E. Cheung, P. Satapathy, V. Pham, H. Hsieh, and X. Chen, "Runtime deadlock analysis of SystemC designs," in *HLDVT*, 2006, pp. 187 –194.
[20] A. Sen, V. Ogale, and M. S. Abadir, "Predictive runtime verification of multi-processor SoCs in SystemC," in *DAC*, 2008, pp. 948–953.
[21] S. Kundu, M. Ganai, and R. Gupta, "Partial order reduction for scalable testing of SystemC TLM designs," in *DAC*, 2008, pp. 936–941.
[22] N. Blanc and D. Kroening, "Race analysis for SystemC using model checking," in *Int'l Conf. on CAD*, 2008, pp. 356–363.
[23] A. Smith, A. G. Veneris, M. F. Ali, and A. Viglas, "Fault diagnosis and logic debugging using boolean satisfiability," *IEEE Trans. on CAD*, vol. 24, no. 10, pp. 1606–1621, 2005.
[24] G. Fey, S. Staber, R. Bloem, and R. Drechsler, "Automatic fault localization for property checking," *IEEE Trans. on CAD*, vol. 27, no. 6, pp. 1138–1149, 2008.
[25] *Accellera Property Specification Language Reference Manual, version 1.1*, http://www.pslsugar.org, 2005.
[26] D. Tabakov, M. Vardi, G. Kamhi, and E. Singerman, "A temporal language for SystemC," in *FMCAD*, 2008, pp. 1–9.
[27] N. Bombieri, F. Fummi, and G. Pravadelli, "A mutation model for the SystemC TLM 2.0 communication interfaces," in *DATE*, 2008, pp. 396–401.