# Scalable Fault Localization for SystemC TLM Designs

Hoang M. Le[1] Daniel Große[1] Rolf Drechsler[1,2]
[1]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
{hle,grosse,drechsle}@informatik.uni-bremen.de

***Abstract*—SystemC and Transaction Level Modeling (TLM) have become the de-facto standard for Electronic System Level (ESL) design. For the costly task of verification at ESL, simulation is the most widely used and scalable approach. Besides the Design Under Test (DUT), the TLM verification environment typically consists of stimuli generators and checkers where the latter are responsible for detecting errors. However, in case of an error, the subsequent debugging process is still very time-consuming.**

**In this paper, we present a scalable fault localization approach for SystemC TLM designs. The approach targets the described standard TLM verification environment and can be easily integrated into one. Our approach is inspired by software diagnosis techniques. We extend the concept of execution profiles of software programs, also known as program spectra, to handle the TLM simulation. The whole simulation consists of several runs; each run corresponds to the request-DUT-response path. During simulation our approach individually collects spectra for each run. Then, based on analyzing the differences of passed and failed runs we determine possible fault locations.**

**We demonstrate the quality of our approach by several experiments including TLM-2.0 designs. As shown in the experiments, the fault locations are identified accurately and very fast.**

## I. Introduction

The move from implementation-driven design at RTL to higher levels of abstraction represents a paradigm shift in the development of electronic systems. This shift has been put into practice as *Electronic System Level* (ESL) design. The main advantages of ESL are to enable greater complexity, virtual prototyping, faster architectural exploration and the reduction of costs through higher productivity.

An important foundation in the ESL arena has become reality with the *Transaction Level Modeling* (TLM) standard TLM-2.0 which has been originally developed for, and now included in the system modeling language SystemC [1], [2]. The core of TLM is a clear separation of computation and communication where for communication standardized interfaces are provided. They abstract implementation details like protocols using function calls. In this context, a transaction models the data exchange of a payload between an initiator and a target. By this, TLM enables efficient virtual prototyping, model interoperabilty and IP re-use.

However, due to the steadily increasing complexity the effort spent for verification dominates the design effort. According to a recent verification study [3], in the time period from 2007 to 2010 the number of designers increased by only 4% whereas in the same time period the number of verification engineers increased by 58%. Hence, a considerable effort has been and is put into ESL verification methodologies and approaches, see e.g. [4], [5], [6], [7], [8].

The most widely used and scalable ESL verification approaches are based on simulation. Besides the *Design Under Test* (DUT), the TLM verification environment typically consists of stimuli generators and checkers where the latter are responsible for detecting the errors. However, in case of an error, the subsequent debugging process is still very time-consuming, in particular since it is highly manual. A few approaches to automate debugging of SystemC designs have been proposed [9], [10], [11]. These approaches focus on very specific faults (e.g. faults related to the SystemC process scheduling), do not determine a fault location, or are limited in the design size as they rely on formal methods.

In this paper, we present a scalable fault localization approach for SystemC TLM designs which targets the described standard TLM verification environment. Our approach is inspired by software diagnosis techniques [12], [13]. We extend the concept of execution profiles of software programs, also known as *program spectra*, to support the TLM simulation. The whole TLM simulation consists of several runs; each run corresponds to the request-DUT-response path. Using this concept of a run, our approach individually collects spectra for each run during the simulation. Then, based on analyzing the differences of passed and failed runs we determine possible fault locations. Furthermore, the runs can overlap due to the concurrency of TLM models. This leads to inaccurate spectra and affects the fault localization accuracy. Our approach also addresses this problem and is able to distinguish overlapping runs while collecting spectra.

In summary, the main contributions of this paper are:

- First scalable fault localization approach for SystemC TLM designs
- Extension of successful concepts from the software domain for TLM simulation
- Very fast and highly accurate TLM fault localization

The remainder of this paper is structured as follows: Section II briefly reviews spectrum-based fault localization for software. The proposed approach is introduced in Section III. In Section IV the experiments are given and finally the paper is concluded in Section V.

## II. Spectrum-based Fault Localization

We follow the terminology used in [13]. A software program is *transformational*, i.e. it transforms an input to an output in a single run. A run is declared as *passed* if no fault has been detected during the run, or as *failed* otherwise.

*Program spectra* present execution profiles of software programs, i.e. a set of data collected during the execution. Many different forms of program spectra exist, such as block

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project SANITAS under contract no. 16M3088 and by the German Research Foundation (DFG) within the Reinhart Koselleck project DR 287/23-1.
978-3-9815370-0-0/DATE13/ 

hit, path hit, etc. Here we only focus on the so-called *block hit spectra*, which is also used in our approach. A block hit spectrum contains for each code block of a program a flag indicating whether the block has been visited in the considered run. For examples of code blocks please see the rightmost part of Fig. 1.

Based on the collected spectra of the passed and failed runs, a *suspiciousness* value for each code block can be calculated. This value corresponds to the the likelihood that the block contains the fault. In the experimental study of [13] several formulas for suspiciousness have been evaluated. Overall, the best results have been achieved with *Ochiai formula*:

$$Suspiciousness(b) = \frac{Failed(b)}{\sqrt{TotalFailed \times (Failed(b) + Passed(b))}}$$

In the formula, $b$ is a code block, $Failed(b)$ ($Passed(b)$) presents the number of failed (passed) runs that executed $b$, whereas $TotalFailed$ is the total number of failed runs. In the next section our TLM fault localization approach is introduced which also uses the Ochiai formula.

## III. Scalable Fault Localization for SystemC TLM

This section presents our fault localization approach in detail. First, the concept is introduced and demonstrated on a concrete SystemC TLM example. Then, the algorithm is given including the support for TLM-2.0 models.

### A. Concept

Foremost, the terminology and definitions described in Section II cannot be used directly for SystemC TLM.

The most substantial difference between a software program and a SystemC TLM design (which is a high-level abstraction of a hardware/software system) is that a TLM design is not transformational. That means it does not calculate an output from a given input in a single run and then stops. Instead, a TLM design repeatedly performs the following cycle: receive requests from the environment, update its internal states and produce responses. Therefore, the terminology must be adapted for SystemC TLM models. This includes the notion of passed and failed runs in the context of SystemC TLM.

Now, consider todays standard simulation-based TLM verification environment. Besides the DUT, the environment also includes a stimuli generator and a checker.[1] The generator continuously produces constrained random and/or directed stimuli, converts them to transaction objects and delivers these to the DUT. The checker serves as the error detection mechanism. It monitors the internal states as well as the responses of the DUT while processing the transaction objects and checks whether these monitored values satisfy the DUT specification. Based on this observation, a *run* can be naturally defined as the propagation of a transaction object from the generator to the checker through the DUT. The predicate *passed* and *failed* is defined based on the corresponding result reported by the checker for the run.

Furthermore, we need to define when and how the (block hit) spectrum of each run is collected during runtime. The straight-forward solution for the first part is to start (stop) recording block hits when a transaction object is delivered to the DUT (the checker). However, this solution is problematic in many cases because of possible overlapping runs, i.e. a new transaction object is delivered before the processing of current transaction objects finishes. This leads to inaccurate spectra and affects the accuracy of fault localization. This problem will be demonstrated by an example in the following section. The ideal solution requires to accurately identify the start and the end of each run as well as all of its suspensions and resumptions. We denote the accuracy grade of this identification step as *recording resolution*. In the following we refer to the resolution used by the straight-forward and ideal solution as the *basic* and *perfect* resolution, respectively. In general, better recording resolution will lead to more accurate fault locations.

With respect to how to collect the spectra of runs, the basic idea is to instrument the DUT adding two types of function calls. Each function call of the first type is inserted at each point identified in the recording resolution. These function calls update a data structure of currently active runs (i.e. not ended or suspended). Each function call of the second type is inserted at the beginning of each code block. These function calls update the block hit spectra of the currently active runs.

### B. Example

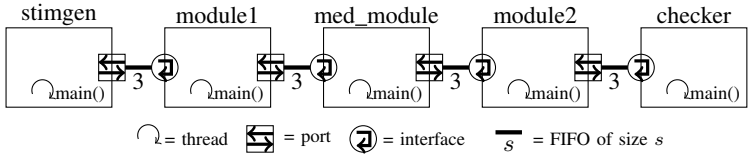
The example in this section shows the importance of the recording resolution. In Fig. 1 parts of a simplified but complete TLM verification environment are shown. For simplicity, we use an untimed FIFO-based TLM model here. TLM-2.0 models are used in the experiments. The DUT repeatedly receives a transaction object (payload) with three integers *x*, *y*, and *z*, calculates the median of these (and some other values), then write the results back into the corresponding fields of the payload. Here we focus only on the submodule *med_module* which calculates the median (function *calculate_med*). The calculation of *module1* and *module2* is performed in the functions *calculate1* and *calculate2* (not shown), respectively. Every payload is generated and delivered to the DUT by the module *stimgen* (see Line 7-8 in Fig. 1). After the requested values have been calculated, the corresponding payload is sent to the module *checker* to check for correctness. As depicted in Fig. 1 the modules are connected using four instances of *tlm_fifo* of size 3. Each module also has a *main* SC_THREAD containing a loop that gets a payload from the incoming FIFO, processes and puts it into the outgoing FIFO.

Now we focus on the spectra collected during simulation. For the sake of simplicity, we only consider the code blocks of the function *calculate_med* which contains a bug (Line 51 should be $p.med = p.x$;). The body of this function has been used in [12] to demonstrate fault localization for software programs. The blocks are numbered as shown in Fig. 1. An excerpt of the simplified trace of a simulation with the first nine random payloads is shown in Fig. 2.

From the trace, the order of execution of the runs (identified by the payload number) as well as the hit blocks during the simulation can be observed. The trace also shows clearly that the runs overlap as the start (end) of each run is indicated by the corresponding text "DUT receives (finishes) payload #i". Before we discuss the spectra we describe for this example which fragments of the trace belong to run #1 for the two resolutions: In Fig. 2 we have marked the fragments of run #1 for the basic resolution using a "normal" line, whereas we have marked the fragments of run #1 for the perfect resolution using a dotted line.

Coming to the spectra, we summarize the results in Table I.

[1] This is a somewhat simplified view, but does not affect the generality.




```
1  class stimgen : public sc_module {
2    sc_port< tlm_fifo_put_if<payload> > out;
3    ...
4    void main() {
5          ...
6          while (true) {
7            generate_payload(p);
8            out->put(p);
9          }
10   }
11 };
12 class moduleX : public sc_module { // X = 1, 2
13   ...
14   void main() {
15         while (true) {
16           in->get(p);
17           calculateX(p);
18           out->put(p);
19         }
20   }
21 };
```

```
22 class checker : public sc_module {
23   ...
24   void main() {
25         while (true) {
26           in->get(p);
27           bool result = check(p);
28           report_result(p, result);
29         }
30   }
31 };
32 class med_module : public sc_module {
33   sc_port< tlm_fifo_get_if<payload> > in;
34   sc_port< tlm_fifo_put_if<payload> > out;
35   ...
36   void main() {
37         while (true) {
38           in->get(p);
39           calculate_med(p);
40           out->put(p);
41         }
42   }
```

```
43 };
44 void med_module::calculate_med(payload& p) {
45   p.med = p.z;
46   if (p.y < p.z) { /* block 1 */
47     if (p.x < p.y)
48       { /* block 3 */ p.med = p.y; }
49     else { /* block 4 */
50       if (p.x < p.z)
51         { /* block 5 */ p.med = p.y; // bug   }
52     }
53   }
54   else { /* block 2 */
55     if (p.x > p.y)
56       { /* block 6 */ p.med = p.y; }
57     else { /* block 7 */
58       if (p.x > p.z)
59         { /* block 8 */ p.med = p.x; }
60     }
61   }
62 }
```

Fig. 1. SystemC TLM example

```
1  DUT receives payload #1 (22, 5, 41)
2  DUT receives payload #2 (5, 31, 11)
3  DUT receives payload #3 (16, 1, 38)
4  calculate_med payload #1
5  HIT BLOCK 1 4 5
6  calculate_med payload #2
7  HIT BLOCK 2 7
8  calculate_med payload #3
9  HIT BLOCK 1 4 5
10 DUT receives payload #4 (26, 19, 3)
11 DUT receives payload #5 (30, 29, 10)
12 DUT receives payload #6 (3, 3, 14)
13 DUT finishes payload #1
14 DUT finishes payload #2
15 DUT finishes payload #3
16 calculate_med payload #4
17 HIT BLOCK 2 6
18 calculate_med payload #5
19 HIT BLOCK 2 6
20 calculate_med payload #6
21 HIT BLOCK 1 4 5
22 Result of payload #1: FAILED
23 Result of payload #2: PASSED
24 ...
```

Fig. 2. Simplified simulation trace

The left half of the table shows the spectra and the suspiciousness of each code block calculated under the basic resolution. In the rows each block is listed and an "x" denotes that the block has been hit in the respective run. The final row shows whether the respective run passed or failed. Column $S$ gives the suspiciousness calculated using the Ochiai formula. As can be seen, using the basic resolution would mean that for example the spectra of the first three runs are identical (blocks 1, 2, 4, 5 and 7 were hit; compare also above mentioned marking in Fig. 2). Thus, the hit of the faulty block 5 is mistakenly covered in the spectra of the passed runs. Overall, as can be seen, the faulty block does not have the highest suspiciousness.

The perfect resolution additionally includes for each run two suspension and resumption points, respectively. A run is suspended when the associated payload is sent to the next module (from *module1* to *med_module* and from *med_module* to *module2*, therefore two suspensions) and then resumed when this module receives the payload. Therefore, a block hit is only collected in the spectrum of a run if the hit takes place during the processing of the associated payload. More precisely, this means the spectrum of the run #k is only updated when *calculate_med* is called to process payload #k. As shown in the right half of Table I, the spectra are accurate and thus the faulty block is correctly identified.

### C. Algorithm

In this section we present the fault localization algorithm for SystemC TLM designs. Thereby, we focus on the three most widely used modeling styles: untimed FIFO-based (e.g. our example), TLM-2.0 loosely-timed (LT), and TLM-2.0 approximately-timed (AT). First, we introduce the auxiliary C++ functions to support the recording of spectra:

- *start_run(id)* and *end_run(id)*: indicate the start/resumption and the end/suspension of a run identified by *id*, respectively and inserts *id* into the set of active runs.
- *record_hit(X)* adds block *X* to the spectra of active runs.
- *record_result(id, result)* assigns the result reported by the checker (passed or failed) to the run identified by *id*. It also updates the number of passed/failed runs for each block, as well as the total number of passed/failed runs. Those numbers are needed to apply the Ochiai formula.

As can be seen, these functions require for each run an identifier *id*. For designs where transaction objects are always passed by reference such as the generic payload of TLM-2.0 models, these objects can be directly used to identify the associated runs. In case that transaction objects are passed by value (for example in many untimed FIFO-based designs), each transaction object is required to have a unique id. The fault localization algorithm consists of the following steps:

1) Instrument the DUT at the recording points (i.e. start, end, suspension, resumption of runs) by inserting calls to *start_run* and *end_run* at appropriate code positions. This step depends mostly on the underlying modeling style of the DUT as detailed below.
2) Instrument the DUT to insert calls to *record_result* in the checker. If the checker is automatically generated, this can be easily integrated into the generation. Otherwise, it takes a minimum effort to manually insert those calls.
3) Instrument the DUT to number all code blocks and insert a call to *record_hit* at the beginning of every block. This instrumentation step can be fully automated.
4) Compile and simulate the instrumented DUT until completion or interruption.
5) Calculate the suspiciousness of all blocks and report the list sorted in descending order of suspiciousness.

The instrumentation can be done as follows for the three above mentioned modeling styles:

TABLE I
SPECTRA AND SUSPICIOUSNESS IN TWO DIFFERENT RESOLUTIONS

| | Runs in Basic Resolution | | | | | | | | | | Runs in Perfect Resolution | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | #1 | #2 | #3 | #4 | #5 | #6 | #7 | #8 | #9 | S | #1 | #2 | #3 | #4 | #5 | #6 | #7 | #8 | #9 | S |
| Block 1 | x | x | x | x | x | x | x | x | x | 0.577 | x | | x | | | x | x | x | x | 0.707 |
| Block 2 | x | x | x | x | x | x | | | | 0.471 | | x | | x | x | | | | | 0 |
| Block 3 | | | | | | | | | | 0 | | | | | | | | | | 0 |
| Block 4 | x | x | x | x | x | x | x | x | x | 0.577 | x | | x | | | x | x | x | x | 0.707 |
| **Block 5** | x | x | x | x | x | x | x | x | x | 0.577 | x | | x | | | x | | | x | 0.866 |
| Block 6 | | | | x | x | x | | | | 0 | | | | x | x | | | | | 0 |
| Block 7 | x | x | x | | | | | | | 0.667 | | x | | | | | | | | 0 |
| Block 8 | | | | | | | | | | 0 | | | | | | | | | | 0 |
| Passed/Failed | F | P | F | P | P | P | P | P | F | | F | P | F | P | P | P | P | P | F | |

x = block hit

TABLE II
RESULTS FOR TLM-2.0 DESIGNS

| Design | LOC | Block | Passed | Failed | Rank |
|---|---|---|---|---|---|
| $lt\_ab\_1$ | 1508 | 118 | 120 | 8 | 1 |
| $lt\_ab\_2$ | 1508 | 118 | 102 | 24 | 1 |
| $lt\_ab\_3$ | 1508 | 118 | 64 | 64 | 1-4 |
| $lt\_db\_1$ | 1508 | 118 | 98 | 30 | 1 |
| $lt\_db\_2$ | 1508 | 118 | 90 | 38 | 4 |
| $at\_mixed\_ab\_1$ | 2536 | 220 | 124 | 4 | 1 |
| $at\_mixed\_ab\_2$ | 2536 | 220 | 92 | 36 | 1 |
| $at\_mixed\_ab\_3$ | 2536 | 220 | 80 | 48 | 5 |
| $at\_mixed\_db\_1$ | 2536 | 220 | 96 | 32 | 1 |
| $at\_mixed\_db\_2$ | 2536 | 220 | 76 | 52 | 1 |
| $at\_mixed\_pb$ | 2536 | 220 | 38 | 2 | 1-2 |
| $dma\_example\_ab$ | 882 | 106 | 582 | 18 | 1 |
| $dma\_example\_db$ | 882 | 106 | 562 | 38 | 1 |

- Untimed FIFO-based: In this modeling style, each FIFO is being polled in a loop. The loop body first gets a transaction object from the incoming FIFO, then processes this object and finally puts it into the outgoing FIFO. Thus, we insert *start_run* after each call of the FIFO interface *get* and *end_run* before each call of *put*.
- TLM-2.0 LT: Insert *start_run* before and *end_run* after each call to the blocking transport interface *b_transport*.
- TLM-2.0 AT: Handle the non-blocking transport interfaces *nb_transport_fw* and *nb_transport_bw* similar to *b_transport*, and the payload event queues similar to FIFOs, since they are also being polled in a loop.

As can be seen, the instrumentation of the DUT only introduces calls to very simple functions. Hence, the overhead added to the simulation will be very small. In the next section the experimental evaluation is presented.

## IV. EXPERIMENTS

In this section we present the experimental results. All experiments have been carried out on a 3 GHz AMD Opteron system with 32 GB RAM running Linux. Since the runtimes of all experiments were negligible as expected, we only focus on the accuracy of the fault localization.

We have considered in our experiments three different designs using both loosely-timed and approximately-timed modeling styles: $lt$ and $at\_mixed$ from the official TLM-2.0 distribution and $dma\_example$ from the ARM AMBA-PV extensions to TLM-2.0.

We have injected bugs into the designs and then applied our approach to locate these bugs. The majority of bugs manipulate either the address or the data of some transaction objects so that the corresponding runs can be classified as failed. These bugs are categorized as *ab* and *db*, respectively. The corresponding bug category is appended to the name of the design as shown in Table II which presents a summary of the results. For $at\_mixed$ we have also introduced a bug that violates the base protocol (hence denoted as *pb*).

The first column of Table II gives the name of the design. The next columns present the lines of code (LOC) and the number of blocks of the design, the number of passed runs and failed runs during the simulation. Finally the rank of the faulty block in the final list sorted by suspiciousness is given in the last column. If several blocks have the same suspiciousness, we show the rank of the first and the last block with this suspiciousness (e.g. 1-4 means that the first four blocks in the sorted list have the highest suspiciousness). As can be observed from Table II, in most cases the block containing the injected bug has been accurately localized (i.e. Rank = 1).

The main limitation of the approach is the granularity of the determined fault locations. These are code blocks that might include the fault. If a highly suspicious block is large (i.e. it contains many statements), it is still time-consuming to find the faulty statement.

## V. CONCLUSIONS

In this paper we have presented the first scalable fault localization approach for SystemC TLM. Inspired by spectrum-based fault localization approaches – developed for software programs – we have extended the concept of program spectra to support the TLM simulation. Collecting accurate spectra requires to identify and distinguish possibly overlapping runs in the TLM simulation such that the transactions resulting from different DUT requests can be unambiguously associated to each respective request.

Overall, a very fast and accurate fault localization has been achieved. We successfully demonstrated the quality of our approach for several TLM-2.0 design examples.

## REFERENCES


[1] Accellera Systems Initiative. (2012) SystemC 2.3 (includes TLM). [Online]. Available: http://www.accellera.org
[2] *IEEE Standard SystemC Language Reference Manual*, IEEE Std. 1666, 2011.
[3] Wilson Research Group and Mentor Graphics, "2010-2011 Functional Verification Study," 2011.
[4] W. Ecker, V. Esen, T. Steininger, M. Velten, and M. Hull, "Implementation of a transaction level assertion framework in SystemC," in *DATE*, 2007, pp. 894–899.
[5] L. Ferro and L. Pierre, "ISIS: Runtime verification of TLM platforms," in *FDL*, 2009, pp. 1–6.
[6] 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.
[7] B. Bailey, F. Balarin, M. McNamara, G. Mosenson, M. Stellfox, and Y. Watanabe, *TLM-Driven Design and Verification Methodology*. Lulu Enterprises Inc., 2010.
[8] H. M. Le, D. Große, and R. Drechsler, "Towards analyzing functional coverage in SystemC TLM property checking," in *HLDVT*, 2010, pp. 67–74.
[9] F. Rogin, C. Genz, R. Drechsler, and S. Rülke, "An integrated SystemC debugging environment," in *FDL*, 2007, pp. 140–145.
[10] F. Rogin, R. Drechsler, and S. Rülke, "Automatic debugging of system-on-a-chip designs," in *IEEE International SOC Conference*, 2009, pp. 333–336.
[11] H. M. Le, D. Große, and R. Drechsler, "Automatic TLM fault localization for SystemC," *IEEE Trans. on CAD*, vol. 31, no. 8, pp. 1249–1262, Aug. 2012.
[12] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *ASE*, 2005, pp. 273–282.
[13] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund, "A practical evaluation of spectrum-based fault localization," *Journal of Systems and Software*, vol. 82, no. 11, pp. 1780 – 1792, 2009.