Mutation Analysis for SystemC Designs at TLM

Valerio Guarnieri, Nicola Bombieri, Graziano Pravadelli, Franco Fummi

Department of Computer Science
University of Verona
Verona, Italy
{firstname.lastname}@univr.it

Hanno Hantson, Jaan Raik, Maksim Jenihhin, Raimund Ubar

Department of Computer Engineering
Tallinn University of Technology
Tallinn, Estonia
{hanno|jaan|maksim|raiub}@ati.ttu.ee

***Abstract* – Mutation analysis has been borrowed from the software testing domain as a technique for evaluating the quality of testbenches in validating digital systems. This paper presents a new method for applying mutation analysis on SystemC hardware designs at Transaction-Level Modeling (TLM). The method injects mutants by directly perturbing the SystemC code. Five key categories of mutation operators are implemented in order to speed up the analysis process. In the paper, a comparison of mutation analysis at two different abstraction levels – TLM and Register-Transfer Level (RTL), is carried out. The experiments show that mutation analysis is considerably faster at TLM than it is at RTL while achieving almost equal mutant coverage. Last but not least, TLM mutation analysis provides also more readable feedback for the engineer to improve the testbench. To the best of our knowledge this is the first method for mutation analysis directly working on uncompiled SystemC TLM code.**

Keywords – mutation analysis, SystemC, TLM, RTL;

I. INTRODUCTION

Mutation analysis and mutation testing are important techniques for software testing. Similarities with procedural programming languages have also brought the idea to hardware domain. The goal of mutation analysis is to assess the quality of the testbenches utilized in hardware simulation.

The approach proposed in this paper relies on creating multiple modifications of the correct code – all of which have a different, but syntactically correct functional change in the code. Such modifications are called *mutations*. The objective of the process is to see whether the testbench is able to detect the change or not. In case the testbench detects it, the mutant is said to be *killed*.

In the opposite case the mutant is *live* and either we need to improve the test suite or the mutant produces output that is equivalent to the original code. Such *equivalent* mutants seem to indicate a weakness in the test suite, but they actually do not, because no test can detect them.

In order for a functional verification or testing method to detect a mutant, three conditions have to be satisfied:

- It must be activated (the corresponding code must be exercised);
- It must be propagated to an observable point;
- It must be detected, i.e. a value mismatch has to be observed at an output.

Mutation analysis is divided into weak and strong mutation. *Weak mutation* requires that only the first of previously described conditions is satisfied. *Strong mutation* requires that all of them are fulfilled. The method proposed in this paper is based on the strong mutation approach.

A. *Overview of previous work*

The initial concept of mutation analysis was first proposed by Richard Lipton in 1971 [1]. However, major work was not published until the end of 1970s [2], [3], [4].

The results of mutation analysis greatly depend on the categories of mutation operators used. Previous research has determined many different categories to use in specific cases. The mutation testing tool Mothra [5], developed in the middle of 1980s to inject and execute mutants on Fortran 77 programs, used three categories of operators: operand replacement, expression modification and statement modification. In total there were 22 elements in the categories. However, many of them were very specific to Fortran language.

Following the approach of Mothra, [6] focused on determining a comprehensive number of mutant operator categories for the C programming language. The operators were divided into 4 categories: statement mutations, operator mutations, variable mutations and constant mutations. In total there were 77 mutant operators, which were again very specific, taking into account errors that alter the expected statement execution flow. The increase in the number of operators with respect to Mothra, comes from the greater complexity and expressiveness of the C language.

Offutt et al. [7] showed experimentally that a selected set of five so called key operator categories provide almost the same coverage as non-selective mutation, with cost reductions of at least four times with small programs, and up to 50 times with larger programs. The approach proposed in this paper is based on these key operator categories.

978-1-4577-1490-0/11/$26.00 ©2011 IEEE

In our previous work [8] we implemented a tool for mutation analysis on high-level decision diagrams (HLDD) [9]. The tool was integrated into APRICOT framework, which is developed at Tallinn University of Technology [10]. It produces good results for RTL designs converted into HLDDs but does not support SystemC and higher abstraction levels, including TLM.

To the best of our knowledge, the only work on mutation analysis that takes into account SystemC models is the tool called SCEMIT [11], which supports also plain C and C++ programs. The tool works on compiled version of the code through a plug-in to the GCC compiler. Injection is done at compile-time and there is no need to modify source code. To prevent an explosion in the number of mutants, the authors focus is set to smaller categories, leaving out those operators which produce great number of mutations and those which operate on language and syntax-specific constructs. Since SCEMIT operates on intermediate code, there is also no need to include them. Therefore, four categories of mutation operators were employed: operator replacement, variable-to-constant replacement, constant replacement and relational operator replacement.

However, the above-mentioned approach offers no feedback for the designer who is unable to interpret the live mutants from the compiled code. In this paper we present TLM mutation analysis that produces readable feedback by directly perturbing the SystemC code. In the paper, a comparison of mutation analysis at two different abstraction levels – TLM and RTL – is carried out. The experiments show that mutation analysis is considerably faster at the TLM than it is at the RTL while achieving almost equal mutant coverage.

The paper is organized as follows. Section 2 presents SystemC Transaction-level modeling. Section 3 explains the SystemC-based mutation analysis method. Section 4 provides experimental results and finally conclusions are drawn in Section 5.

II. TRANSACTION-LEVEL MODELING WITH SYSTEMC

In this section, we provide background information on TLM and the TLM-2.0 standard.

A. TLM overview

TLM is the reference modeling style for design and verification of modern system-on-chips (SoCs) at the electronic system-level. The main advantage of TLM lies in the great speed-up it provides to the design process. In fact, it allows designers to write a fully functional system-level description, which can be simulated at much greater speed than RTL models. This enables feedback at the early phases of the design process, thus producing a better starting point to further refine and elaborate.

The Open SystemC Initiative (OSCI) [12] committee has been developing a reference standard for TLM in the last years to ensure interoperability between suppliers and users. As such, TLM-2.0 has become the final reference standard for SystemC TLM [13].

B. SystemC TLM-2.0 coding styles and communication interfaces

TLM presents a variety of *use cases*, such as software development, software performance analysis, architectural analysis and hardware verification. Rather than creating a specific abstraction level for each use case, the TLM-2.0 standard describes a number of coding styles that are appropriate for, but not locked to, the different use cases.

Two examples of TLM-2.0 coding styles proposed by OSCI are the following:

- *Loosely-timed.* The loosely-timed coding style is appropriate for software development using a virtual platform model of an MPSoC, where the software may, for example, include one or more operating systems. This coding-style implies a loose dependency between timing and data. Models implemented with this coding style do not have to rely on the passing of time to generate a response. Usually, resource contention and arbitration are not considered.
- *Approximately-timed.* The approximately-timed coding style is appropriate for architectural exploration and performance analysis. This coding-style implies a much stronger dependency between timing and data. Since models implemented with this coding style must synchronize transactions before processing them, they are forced to incur multiple context switches during simulation, which lead to performance penalties. On the other hand, they easily model resource contention and arbitration.

The best-suited coding style is applied depending on the target use case and each coding style is implemented by using a specific TLM interface. The TLM-2.0 standard defines the following interfaces:

- *Blocking interface.* It allows a simplified coding style for models that complete a transaction in a single function call, by exploiting the blocking primitive `b_transport(payload, time)`. Timing annotation is performed through the `time` parameter of the primitive. The blocking interface suits the implementation of the loosely-timed coding style.
- *Non-blocking interface.* It allows the association of multiple timing points with a single transaction. It relies on the use of non-blocking primitives `nb_transport_fw(payload, time,`

`phase)` and `nb_transport_bw(payload, time, phase)`. The `time` parameter supports timing annotation as well, while parameter `phase` is used to break down the communication process, thus allowing to implement more accurate protocols, such as the four phases approximately-timed coding style.

- *Direct memory interface (DMI) and debug transport interface.* They are specialized interfaces that provide direct access and debug access to an area of memory owned by a target. They bypass the usual path through interconnect components used by the transport interface. DMI accelerates regular memory transactions in a loosely-timed simulation, while the debug transport interface provides debug access without the delays or side-effects associated with regular transactions.

Communication in TLM is generally achieved by exchanging packets containing data and control values (i.e., payload objects) through a channel (e.g., a socket) between an initiator module (master) and a target module (slave).

III. Method for Mutation Analysis on SystemC

This section presents mutation analysis method implemented for SystemC designs.

The first step of mutation analysis is to find the optimal categories of mutation operators. This task is fairly complicated because of the wide range of possible changes that can be made in the source code. Determining the best operator categories for a given example usually involves code analysis to find the potential modification possibilities.

When designing the categories of mutation operators to be used, the following guidelines have been followed:

- Mutant operators should accurately model the errors that may be introduced by developers and engineers;
- Each mutant operator should change only one syntactic entity of a program;
- Each mutant operator should generate a syntactically correct program (i.e., the mutants can be compiled and executed);
- The categories should not generate too many mutants in order to have reasonable execution times, but it should provide the best coverage of possible design errors;
- The categories should minimize the possibility of generating an equivalent mutant.

The focus of this work was not to propose new operators or operator categories. Therefore, we used a slightly modified set of five key operator categories, proposed in [8]. In our experiments, those five categories have provided almost the same coverage as non-selective mutation, with cost reductions of at least four times with small programs, and up to 50 times with larger programs [7]. The categories of operators used in our work are as follows: arithmetic operator replacement (AOR), logical connector replacement (LCR), shift operator replacement (SOR), relational operator replacement (ROR) and unary operator injection (UOI).

Table 1 shows the list of replacements for each mutation operator category. In every case the operator is substituted by another operator from the group. This is done until all operators are covered.

TABLE I. CATEGORIES OF MUTATION OPERATORS.

Mutation operator category	List of replacements
AOR (arithmetic operator replacement)	Addition (ADD), subtraction (SUB), multiplication (MULT), division (DIV), modulo (MOD)
LCR (logical connector replacement)	AND, NAND, OR, NOR, XOR
SOR (shift operator replacement)	Left shift (SL), right shift (SR)
ROR (relational operation replacement)	Equal (EQ), not equal (NEQ), greater than (GT), less than (LT), greater than or equal (GE), less than or equal (LE)
UOI (unary operation insertion)	Negative (NEG), inversion (INV)

The injection process can be carried out in two ways:

- Fault simulation-based;
- Testbench-based.

In the fault simulation-based approach firstly the original, fault-free code is simulated. After this, all mutants are injected one at a time, simulated and compared against the result of the original code. This method was used in our previous work [8].

In the testbench-based approach firstly the whole mutant set is added to the code and a counter is introduced for selecting mutants. Next the original code and all mutants are simulated, one after another. For every mutant the result is compared against the result of the original code. Testbench-based method is used in current work and will be described more thoroughly in the next paragraphs.

Concerning the injection process, the original system description is first analyzed and injection locations are identified. Then for each location a proper mutation operator is applied, resulting in different versions of the current statement being created.

In order to keep the following simulation phase easier and the result of the injection more manageable, only one

injected system description is created. Instead of creating one separate description for each injected mutant, we generate a system description that includes all the code produced by the injection phase, and that allows to selectively activate one mutant at a time through the use of a `fault_number` variable, properly driven by the testbench during the simulation phase. Figure 1 illustrates the whole injection process.

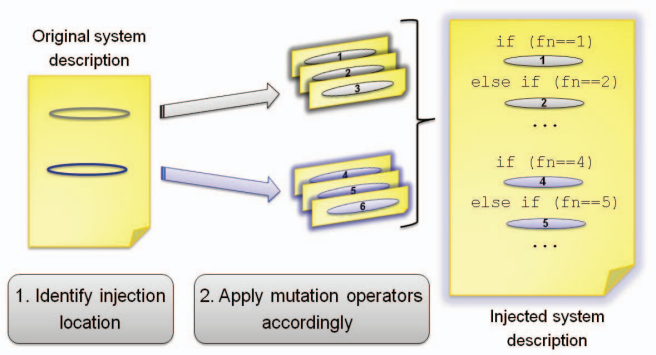

FIGURE I. MUTANT INJECTION OVERVIEW.

IV. EXPERIMENTAL RESULTS

In order to validate the efficiency, in terms of speed and coverage differences of the proposed method at different abstraction levels, we performed mutation analysis on a number of designs, three versions for each of them:

- TLM with mutant injection in the functionality part, which consists of C++ code (*TLM injected*);
- RTL version obtained by synthesizing the injected functionality part (from the previous step) with Mentor Graphics Catapult C [14] (*RTL synthesized from injected*);
- RTL version obtained by synthesizing the fault-free functionality part (from the original design description) with Mentor Graphics Catapult C, and then injecting mutants directly at this level (*RTL directly injected*).

Designs used for the experiments are as follows:

- *adpcm*: performs adaptive differential pulse code modulation to compress audio packets;
- *div*: filter for similarity analysis of image pixels;
- *gcd*: computes the greatest common divisor for two unsigned integers.

Experiments were carried out by injecting mutants on each version for each design and then simulating them to compute mutation coverage. In total nine experiments were made, and the results are shown in Table 2.

The results confirmed that mutants injected at TLM were preserved during synthesis to RTL and the number of mutants remained exactly the same on the *TLM injected* and *RTL synthesized from injected* versions of the designs.

From the perspective of simulation time, the results were completely different. Simulation times of the *RTL synthesized from injected* version were drastically increased, as Figure 4 shows. This again confirmed our expectations, as moving to a more detailed abstraction level should result in longer run-times.

This highlights a benefit from injecting mutants directly to the TLM version, because a very good simulation speed is achieved without losing accuracy, and sufficiently accurate feedback is available even in the early phases of the design process.

On the other hand, injecting mutants directly at RTL (*RTL directly injected*), produces slightly better results in terms of mutation coverage, but at the price of slower simulation times. Figure 5 details mutation coverage results.

It is important though that a major drawback of such an approach is code readability, as TLM code is much easier for a human being to understand and modify than the automatically generated RTL code. Examples of TLM and generated RTL code are shown on Figure 2 and Figure 3 respectively.

```
rem = a % b;
while (rem != 0) {
  a = b;
  b = rem;
  rem = a % b;
}
```

FIGURE II. SYSTEMC CODE EXAMPLE AT TLM.

TABLE II. EXPERIMENTAL RESULTS.

	TLM injected			**RTL synthesized from injected TLM**			**RTL directly injected**		
Design	adpcm	div	gcd	adpcm	div	gcd	adpcm	div	gcd
# of mutants	66	45	21	66	45	21	61	16	18
# of killed mutants	23	44	19	23	44	19	25	16	17
Mutation coverage	35,0%	98,0%	90,0%	35,0%	98,0%	90,0%	41,0%	100,0%	94,0%
# of code lines	835	441	284	4031	1586	919	788	347	378
Simulation time (ms)	5	4	4	1651	84	3312	134	15	271

The difference between the two pieces of source code should be immediately striking. The generated RTL code suffers from the lack of readability deriving from being automatically generated by a high-level synthesis tool. In this context, correctness and automatic code translation are the main priorities. In fact, the most common scenario in high-level synthesis consists of obtaining the synthesized description and providing it to other tools responsible for the physical implementation. As such, the generated code is not really meant to be clearly understandable by human beings, nor to be manually edited.

Somewhat surprising was the fact that at *RTL directly injected* the number of possible mutations decreased compared to *TLM injected* and *RTL synthesized from injected*. This can be explained by the optimizations introduced by Catapult C during the synthesis process, which often result in using less assignments and operators than the corresponding description at TLM. Nevertheless, it must be stressed that this version suffers from the readability problem we outlined before.

```
if (( mc_bool(rst.read()))) goto gcdAndLcm_main;

// C-Step 1 of Loop 'gcdAndLcm_while'

gcdAndLcm_rem_sva =
CONV_STD_LOGIC_VECTOR(CONV_UNSIGNED(UNSIG
NED(gcdAndLcm_b_sva_read_dft) %
UNSIGNED(gcdAndLcm_rem_sva), 32), 32);

gcdAndLcm_rem_sva = gcdAndLcm_rem_sva_1;
```

FIGURE III. SYSTEMC CODE EXAMPLE AT GENERATED RTL.

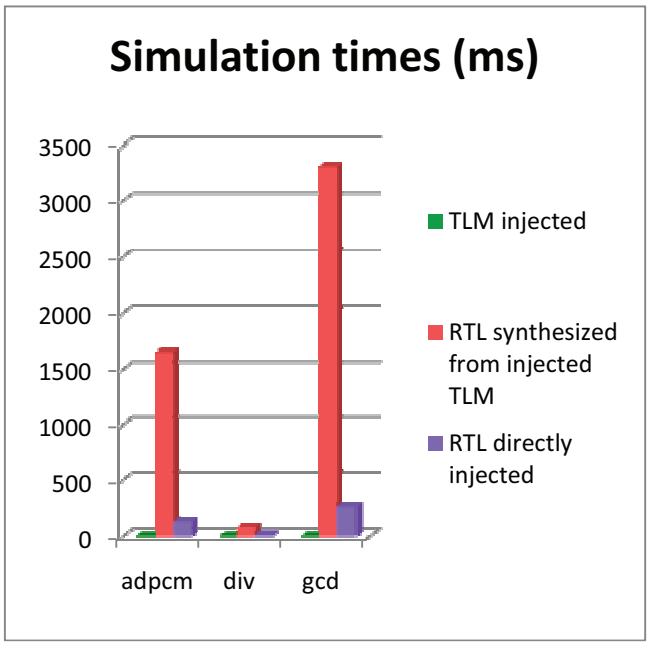

FIGURE IV. SIMULATION TIMES.

It is worth noting that these experiments and the subsequent analysis led us to an improvement of the testbenches employed, making them more comprehensive by considering corner cases which were not taken into account before. In one case we also discovered a bug in the design description when investigating the reasons for low mutation coverage.

Thus, we can definitely claim that mutation analysis allowed us to evaluate the quality of the verification environment and to verify the correctness of a design through simulation.

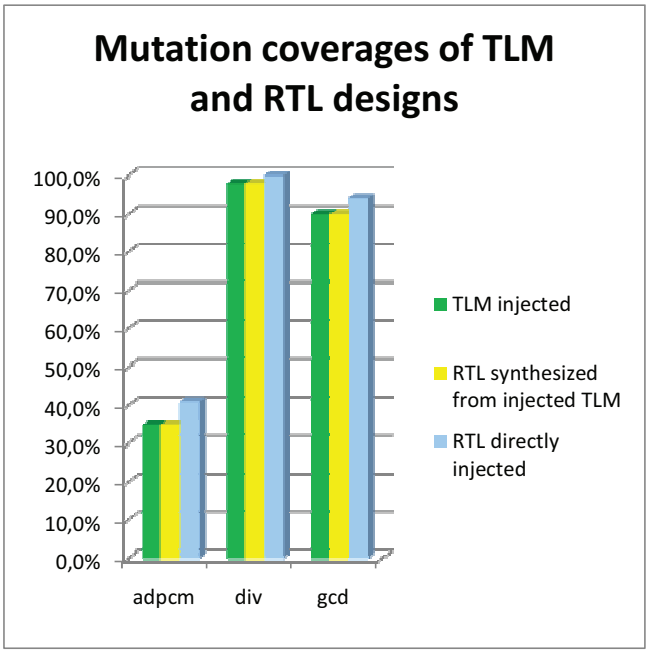

FIGURE V. MUTATION COVERAGES.

V. CONCLUSIONS

We proposed a new method to automatically inject faults into the functionality of system descriptions that works at different abstraction levels (TLM and behavioral RTL). To the best of our knowledge this is the first method for mutation analysis directly working on uncompiled SystemC TLM code. Five key categories of mutation operators were used to simulate the faults.

Experimental results with different versions of different designs showed that injecting faults directly to RTL code provides slightly better mutation coverage. However, this does not mitigate the loss in readability and simulation times when compared to TLM.

Future work includes improving the set of mutant operators in order to cover more design errors, performing additional experiments, implementing a tool for automatic fault injection and extending the work to the field of design error correction with mutants.

ACKNOWLEDGMENTS

The work has been supported by European Commission Framework Program 7 projects FP7-REGPOT-2008-1 CREDES and FP7-ICT-2009-4-248613 DIAMOND, by European Union through the European Regional Development Fund, by Estonian Science Foundation grants 7068, 7483 and 8478.

REFERENCES

[1] R. Lipton, “Fault Diagnosis of Computer Programs”, 1971.

[2] T. Budd and F. Sayward, "Users guide to the Pilot mutation system", technical report 114, Dept. of Comp. Science, Yale University, 1977.

[3] R. G. Hamlet, "Testing programs with the aid of a compiler", IEEE Transactions on Software Engineering, vol. 3, pp. 279-290, July 1977.

[4] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer", IEEE Computer, vol. 11, pp. 34-41, April 1978.

[5] B. Choi, R. DeMillo, E. Krauser, R. Martin, A. Mathur, A. Offut, H. Pan, E.Spafford, “The Mothra Toolset”, Proceedings of Hawaii International Conference on System Sciences, 1989.

[6] H. Agrawal, R. DeMillo, B. Hathaway, W. Hsu, E. Krauser, et al., “Design of mutant operators for the C programming language”, Software Engineering Research Center, Purdue University, 1989.

[7] A. J. Offutt, G. Rothermel, and C. Zapf, "An experimental evaluation of selective mutation", in Proceedings of the Fifteenth International Conference on Software Engineering, (Baltimore, MD), pp. 100-107, IEEE, May 1993.

[8] H. Hantson, J. Raik, G. di Guglielmo, M. Jenihhin, A. Chepurov, F. Fummi, R.Ubar, “Mutation Analysis on High-Level Decision Diagrams”, IEEE Latin-American Test Workshop, pp. 1-6, March 28-31, 2010.

[9] R.Ubar, "Test Synthesis with Alternative Graphs", IEEE Design & Test of Computers, Spring 1996, pp. 48-57

[10] M.Jenihhin, J.Raik, A.Chepurov, R.Ubar. Simulation-based Verification with APRICOT Framework using High-Level Decision Diagrams. IEEE East-West Design & Test Symposium, Sept.18-21, 2009, pp.13-16.

[11] P. Lisherness, K.-T. Cheng, “SCEMIT: A SystemC Error and Mutation Injection Tool”, Proceedings of the 47th Design Automation Conference, p. 228-233, 2010.

[12] OSCI, 2009, http://www.systemc.org.

[13] OSCI, OSCI TLM-2.0 Language Reference Manual, 2009, http://www.systemc.org.

[14] Mentor Graphics, “Catapult C Synthesis - Full-Chip High-Level Synthesis”, http://www.mentor.com/esl/catapult/