J Electron Test (2012) 28:435–448
DOI 10.1007/s10836-012-5303-6

On the Reuse of TLM Mutation Analysis at RTL

Valerio Guarnieri · Giuseppe Di Guglielmo · Nicola Bombieri · Graziano Pravadelli · Franco Fummi · Hanno Hantson · Jaan Raik · Maksim Jenihhin · Raimund Ubar

Received: 22 August 2011 / Accepted: 24 April 2012 / Published online: 25 May 2012

Abstract Mutation analysis has gained consensus during the last decades as being an efficient technique for measuring the quality of SW testbench. More recently, it has been efficiently applied for validating testbenches of embedded system models implemented in hardware description language (HDL) at different abstraction levels (i.e., RTL, TLM). This article analyzes how mutation analysis performed at TLM can be reused at RTL and, in particular, how such a reuse can help designers in (i) optimizing the time spent for simulation at RTL, and (ii) improving the RTL testbench quality. Two alternatives of TLM mutation analysis reuse are presented and investigated for proposing an efficient methodology of RTL mutation analysis. Through experimental results, the proposed methodology is compared to the standard RTL mutation analysis to confirm its efficiency in terms of both simulation time and reached mutation coverage.

Responsible Editor: L. M. Bolzani Pöhls

V. Guarnieri · G. Di Guglielmo · N. Bombieri · G. Pravadelli · F. Fummi
Dipartimento di Informatica, Università di Verona, Strada Le Grazie 15, 37134 Verona, Italy

V. Guarnieri
e-mail: valerio.guarnieri@univr.it

G. Di Guglielmo
e-mail: giuseppe.diguglielmo@univr.it

N. Bombieri
e-mail: nicola.bombieri@univr.it

G. Pravadelli
e-mail: graziano.pravadelli@univr.it

F. Fummi
e-mail: franco.fummi@univr.it

H. Hantson · J. Raik (✉) · M. Jenihhin · R. Ubar
Department of Computer Engineering, Tallinn University of Technology, Raja 15, 12618 Tallinn, Estonia
e-mail: jaan@ati.ttu.ee

H. Hantson
e-mail: hanno@ati.ttu.ee

M. Jenihhin
e-mail: maksim@ati.ttu.ee

R. Ubar
e-mail: raiub@ati.ttu.ee

Keywords Mutation analysis · Mutation testing · SystemC · Transaction-level modeling · Register-transfer level

1 Introduction

Mutation analysis and mutation testing have definitely gained consensus during the last decades as being important techniques for software testing [10, 17, 21, 23]. Such approaches rely on the creation of several versions of the program to be tested, *mutated* by introducing syntactic changes. The purpose of such mutations consists of perturbing the behavior of the program to see if the test suite is able to detect the difference between the original program and the mutated versions. Mutation analysis measures the effectiveness of the test suite by computing the percentage of detected mutations (mutation coverage), while mutation testing aims at increasing the mutation coverage by generating a larger set of high quality testbenches.

Similar concepts are implemented also in HW testing [1], where high-level fault simulation is applied to measure the quality of testbenches, and test pattern generation is used to improve fault coverage. In this case, mutations introduced in the HW descriptions are refered to as *faults*.

In the recent years, the close integration between HW and SW parts in modern embedded systems and the development of high-level languages suited for modeling both HW and SW (e.g., SystemC and SystemC TLM) have required the definition of mutation analysis-based strategies that work at system level, where HW and SW functionalities are not partitioned yet. In particular, some works have been proposed to apply mutation analysis to SystemC TLM [6–8, 26, 33, 34], since transactional level modeling (TLM) has become the reference modeling style for system-level design and verification of modern system-on-chips (SoCs) [12]. Experimental results have shown that TLM mutation analysis greatly speeds up the design process by allowing designers to model and verify complex systems early in the design flow with respect to RTL approaches.

However, applying mutation analysis only at TLM is not enough. In fact, once verified, the TLM implementation must be then refined into a more detailed RTL implementation, where the verification process must be repeated before the tape-out. In such a TLM-to-RTL refinement step, the TLM IP functionality is synthesized into a cycle accurate implementation, while the TLM interface is replaced by a pin accurate interface composed of all the data I/O ports with the addition of some control ports for implementing handshaking mechanisms specific to the target platform (e.g., bus compliant protocols, enabling flags, etc.).

In this context, high-level synthesis (HLS) is considered the reference paradigm for automatically generating RTL descriptions starting from the system level (i.e., TLM) models [15] and different HLS tools are emerging on the market for TLM-to-RTL synthesis [13, 16].

After the TLM-to-RTL synthesis, mutation analysis must be applied to the RTL code in order to check: (i) whether all the high-level functionality originally checked at TLM are correctly preserved at RTL, (ii) whether all the architectural details typical of RTL implementations (e.g., pipelined behaviors, clock gating, clock-based delay, etc.) have been correctly introduced by the synthesis process. In this way, mutation analysis gives also useful information to identify any specific functionality wrongly synthesized at RTL.

Nevertheless, although mutation analysis can be performed by a fast and efficient simulation at TLM, it sensibly slows down at RTL due to the amount of accuracy details of both the RTL description and the RTL mutation model. In addition, the traditional mutation analysis directly applied to synthesized RTL would not always give back useful information to find and classify design bugs.

In this context, this article aims at investigating the actual benefits of reusing the TLM mutation analysis (i.e., TLM testbenches and TLM mutants) for improving the mutation analysis performed at RTL. As a results, the article proposes a methodology that, through the reuse of TLM mutation analysis, aims at better exploiting the time spent in simulation for RTL mutation analysis and improving the result readability for enhancing the RTL testbenches.

Starting from a TLM model, we investigate and compare two alternatives (see Fig. 1):

1. going down synthesizing the TLM model to an RTL implementation, then moving to the right to inject mutants on the RTL code (we call this path *native RTL mutation analysis*), or
2. going right to inject TLM mutants and then synthesizing the mutated TLM model to a mutated RTL implementation (we call this path *TLM-derived mutation analysis*).

Such a comparison and the related experimental results underline that:

– By applying the *TLM-derived mutation analysis*, it can be observed that any functionality verified at TLM by the TLM testbench can be considered verified at RTL when the TLM testbench is reused at RTL. This suggest that there is no need to further improve the RTL testbench for stressing the IP functionality.

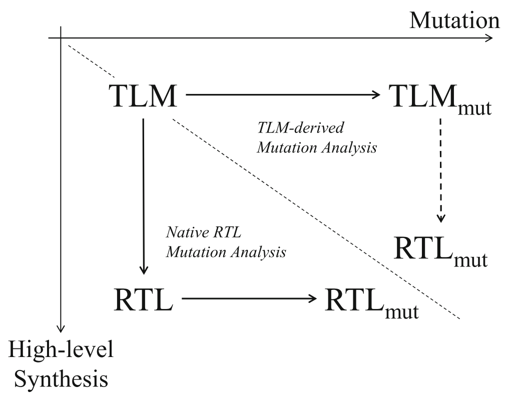


Fig. 1 High-level synthesis and mutation analysis may be combined in different ways

- By considering the *standard RTL mutation analysis* (i.e., native RTL mutation analysis) and reusing the TLM testbench:
  - the RTL mutation coverage is low (as expected);
  - the RTL mutation coverage easily grows up by enriching the testbench with stimuli randomly generated;
  - the remaining "unkilled" mutants can be classified in three different categories. This information helps designers to improve the RTL testbench. In particular, the first two categories are general and apply to every TLM-to-RTL synthesis process. In contrast, the third category helps designers to improve the RTL testbench by taking into account the micro-architectural details chosen during the synthesis process.

The article is organized as follows. Section 2 presents an analysis of related works. Section 3 introduces an overview of the key concepts needed for understanding the methodology. Section 4 presents the main methodology while Section 5 reports the obtained experimental results and the analysis of them. Section 6 is devoted to the concluding remarks.

2 Related Work

The initial concept of mutation analysis was first proposed by Richard Lipton in 1971 [25]. However, major work was not published until the end of 1970s [10, 17, 21].

In general, 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 [14, 29], 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, [2] focused on determining a comprehensive number of mutant operator categories for the C programming language. The operators were divided into four 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. [30] 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 article is based on these key operator categories.

Mutation analysis has been applied also to Java [24] and SQL [28, 35]. Several approaches [3, 4], empirical studies [27] and frameworks [9] have been presented in the literature for mutation analysis of such languages.

Hantson et al. [22] propose a technique to apply mutation analysis to high-level decision diagrams (HLDD). It produces good results for RTL designs converted into HLDDs but does not support SystemC and higher abstraction levels, including TLM.

Only in the recent years mutation analysis has been applied to languages for system-level design and verification such as SystemC [6–8, 26, 33, 34]. Bombieri et al. and Sen [6, 7] propose mutation models for perturbing SystemC TLM descriptions. In particular, these works present different analysis of the main constructs provided by the SystemC TLM 2.0 library and a set of mutants to perturb the primitives related to the TLM communication interfaces.

Sen [33] propose a fault model by developing mutation operators for concurrent SystemC designs. In particular it aims at verifying SystemC descriptions by facing non-determinism and concurrency problems such as starvation, interference and deadlock typical of such language.

Bombieri et al. [8] introduces the concept of functional qualification for measuring the quality of functional verification of TLM models. Functional qualification is based on the theory of mutation analysis but considers a mutation to have been killed only if a testbench fails. A mutation model of TLM behaviors is proposed to qualify a verification environment based on both testbenches and assertions. The presentation describes at first the theoretic aspects of this topic and shows advantages and limitations of the application of mutation analysis to TLM.

Sen and Abadir [34] proposes to attack the verification quality problem for concurrent SystemC programs by developing novel mutation testing based coverage metrics. The approach involves a comprehensive set of mutation operators for concurrency constructs in SystemC and defines a novel concurrent coverage metric considering multiple execution schedules that a concurrent program can generate.

Lisherness and Cheng [26] presents SCEMIT, a tool for the automated injection of errors into C/C++/ SystemC models. A selection of mutation style errors are supported, and injection is performed though a plugin interface in the GNU compiler collection (GCC), which minimizes the impact of the proposed tool on existing simulation flows. The results show the value of high-level error injection as a coverage measure compared to conventional code coverage measures.

Different aspects concerning hardware or software implementation are analyzed in all these works. All these approaches are suited to target basic constructs, low-level synchronization primitives as well as high-level primitives typically used for modeling TLM communication protocols.

The reuse of TLM testbenches for RTL fault simulation has been proposed in [5]. In this work the authors show that if a fault is detectable by an RTL test bench then it can be detected also by a TLM test bench filtered by a transactor. However, the authors do not elaborate about the differences between injecting mutants before or after TLM-to-RTL synthesis, as we do in this article.

To the best of our knowledge, there is no work in literature that faces the reuse of mutation analysis through the different refinement steps of a TLM-based design flow as done in the following sections. This article extends the work presented in [19] and presents a comprehensive work on mutation analysis for system level descriptions (i.e., SystemC TLM) and how such analysis can be reused once such descriptions are synthesized at RTL.

3 Background

In this Section, we provide background information on mutation analysis, high-level synthesis and the TLM-2.0 standard.

3.1 Mutation Analysis

Mutation analysis [17] has definitely gained consensus during the last decades as being an important technique for software (SW) testing [18]. Such testing approaches rely on the creation of several versions of the program to be tested, "mutated" by introducing syntactically-correct functional changes.

The purpose of such mutations lies in perturbing the behaviour of the program to see if the test suite is able to detect the difference between the original program and the mutated versions. Thus, the effectiveness of the test suite is measured by computing the percentage of detected mutations.

In the traditional mutation analysis the output of the design-under-test is compared with and without the mutation [20]. If there is a difference observed in the output, then the mutant is considered to have been killed. If no difference is observed, the mutant is said to be live. This is due to one of three possibilities:

- the testbench is not able to detect a change in the output, so it needs to be improved and extended to detect the mutant;
- the mutant operates on dead code, i.e. in code that is never reached (and thus activated), so any change introduced will never be executed during simulation;
- the mutant is functionally equivalent to the original code, i.e. the mutant does not introduce any change in the computation.

The first fundamental hypothesis of mutation analysis is that if the system contains non-killed mutants then the system also could contain real bugs (or coding mistakes) that cannot be found by the existing tests. If the testing is improved so as to kill live mutants, then these same tests can expose the vast majority of previously unknown bugs in the original program. According to this first hypothesis, mutation analysis permits to evaluate the quality of test benches for functional verification.

The second fundamental hypothesis of mutation analysis is the "competent-programmer hypothesis". The design is considered to be largely correct, i.e., the majority of the code is assumed to not contain bugs. This is important because the mutation analysis assesses the ability of the verification environment to measure the quality of the current design implementation. When mutations are introduced, they take the design slightly out of specification. This second hypothesis explains why by detecting artificially-introduced design errors the test benches are able to exercise most of the system functionalities.

Similar concepts are applied also for testing of hardware descriptions at RTL, where verification engineers use high-level fault simulation to measure the quality of testbenches, improve fault coverage, and, thus, providing more effective test suites. In this case, mutations introduced in the HW descriptions are refered to as *faults* [1].

Nowadays, (i) the close integration between HW and SW parts in modern system-on-chips (SoCs), (ii) the development of languages suited for modeling systems at a higher level of abstraction (i.e., SystemC TLM), and (iii) the need of developing verification strategies to be applied early in the design flow, require the definition of mutation-based strategies that work at the electronic system-level (ESL).

3.2 High-Level Synthesis

High-level synthesis (HLS) is considered the reference paradigm for the automatic generation of RTL descriptions starting from high-level algorithmic models. An HLS-based flow typically starts with designers developing a high-level description which captures the desired functionality.

This description is written in a high-level language, such as C/C++, and is untimed, thus not including any timing notion. At this abstraction level, variables and data types are not accurate enough for the HW design domain. As such, conversion of floating-point and integer data types into bit-accurate data types is required. This is usually done by analyzing all operations performed in the starting description.

HLS tools generate a fully timed RTL implementation of the initial untimed high-level description. Designers are given a vast array of architectural choices pertaining to the RTL domain, such as pipelining, delay/area optimization, and so on. This allows for the creation of a custom architecture that best suits the desired functionality. Different RTL implementations can be explored until a satisfying one is found.

4 Reuse of TLM Mutation Analysis at RTL

In the context of analyzing the effect of reusing TLM testbenches and mutants at RTL, the following sections are devoted to present: the TLM to RTL synthesis process (Section 4.1), the TLM mutation analysis infrastructure and the related mutation model (Section 4.2), and, finally, considerations about the comparison between TLM-derived mutation analysis and the native RTL mutation analysis showed in Fig. 1 (Section 4.3).

4.1 TLM-to-RTL Synthesis

An HLS tool, i.e, Mentor Graphics CatapultC [13], is used in the proposed flow to automatically perform TML-to-RTL synthesis. CatapultC takes as input C/C++ code providing a high-level description of the desired system behavior. In particular, the preliminary step in the synthesis process consists of isolating a procedure which wraps up the system functionality.

Procedure parameters are used to provide inputs and retrieve outputs, and they are translated into corresponding input/output RTL ports during the synthesis process. A basic handshaking protocol is added to provide a convenient means to achieve communication with a testbench during the simulation phase. Thanks to this addition, the reuse of the TLM testbench at RTL is possible with an almost effortless transition. Otherwise, a transactor would be required, thus increasing the complexity of the simulation environment and introducing other possible sources of errors.

Functionality is synthesized by decomposing TLM operations into smaller basic operations (i.e., sums and concatenations) at RTL, which are usually performed on ranges of bits. Inner signals are introduced to store intermediate results, which are then combined to produce the final outputs.

Finally, in the case of TLM-derived mutation analysis, the mutated design must still be synthesizable. This implies that the chosen mutation operators shall not introduce non-synthesizable constructs.

4.2 Mutation Analysis Infrastructure

Mutation analysis relies on a set of operators to perform syntactic changes to the description. These operators can be conveniently classified into categories, according to what they alter. Although mutation analysis is a powerful approach for modelling design errors, it is computationally expensive. In particular, the main expense of mutation is the high number of variants of the original design, that must be repeatedly executed. Thus, in this work, we adopt a simplified subset of the "sufficient" mutation operators from [31], themselves a subset of those proposed in [11]. In particular the five categories of operators are: arithmetic operator replacement (AOR), bitwise operator replacement (BOR), relational operator replacement (ROR), shift operator replacement (SOR) and unary operator injection (UOI). Table 1 shows the list of possible replacements for each mutation operator category. Whenever an operator belonging to a given category is found, it is replaced with all the others in its respective group. These categories of mutation operators easily apply to both TLM and RTL SystemC descriptions. In particular, these modifications do not create problems to the HLS tool, since the resulting description is still synthesizable.

Moreover, mutation coverage subsumes other structural-coverage metrics. For example, the branch coverage requires that each branches of a conditional statement (e.g., `IF`) are executed. The relational-operator replacement (ROR), among other modifications, replaces each decision by TRUE or FALSE. To kill the TRUE mutant, a test case must

Table 1 Categories of mutation operators

Mutation operator category	List of replacements
Arithmetic operator replacement (AOR)	Addition (ADD), subtraction (SUB), Multiplication (MUL), division (DIV), modulo (MOD)
Bitwise operator replacement (BOR)	AND, OR, XOR
Relational operator replacement (ROR)	Equal (EQ), not equal (NEQ), greater than (GT), less than (LT), greater than or equal (GE), less than or equal (LE)
Shift operator replacement (SOR)	Left shift (SL), right shift (RS)
Unary operator insertion (UOI)	Negation (NEG)

take the FALSE branch, and to kill the FALSE mutant, a test must take the TRUE branch. Thus the ROR operator subsumes branch coverage. By extending the adopted set of mutation operators all the most used structural-coverage criteria can be subsumed [32].

Figure 2 provides an overview of the adopted mutant-injection process. Injection is carried out by first scanning the input description to identify locations to be injected. Then for each identified injection location, mutations are produced by replacing the involved operator with all the other operators belonging to the same category, one-by-one.

To facilitate the following simulation phase and reduce the compilation time of all the generated mutants, only one injected system description is created, instead of creating and compiling a separate one for each injected mutant. The essence of this method lies in the creation of a specially parameterized program called *meta-mutant*: the unique injected description includes all the code produced by the injection phase, and allows to selectively activate one mutant at a time through the use of the `mutant_id` variable. Such a variable is properly driven by the testbench during the simulation phase. As for the choice of mutation operators, the meta-mutant apply to both RTL and TLM SystemC, and the resulting code is easily synthesized by the HLS tool.

4.3 Coverage at RTL: A Cautionary Tale

As anticipated in Fig. 1, RTL mutation analysis can be performed in two alternative ways:

1. TLM-derived mutation analysis, i.e., mutation followed by synthesis: the starting TLM description is first mutated, and then synthesized to produce a mutated RTL description denoted by RTL'_{mut} in Fig. 1.
2. Native RTL mutation analysis, i.e., synthesis followed by mutation: the starting TLM description is first synthesized to produce a mutant-free RTL description. Then, the obtained RTL is mutated, obtaining RTL description denoted by RTL_{mut} in Fig. 1.

It is worth noting that, at first, these two scenarios serve the same purpose, i.e., performing RTL mutation analysis by reusing TLM testbenches, but the difference in the abstraction level on which mutants are injected

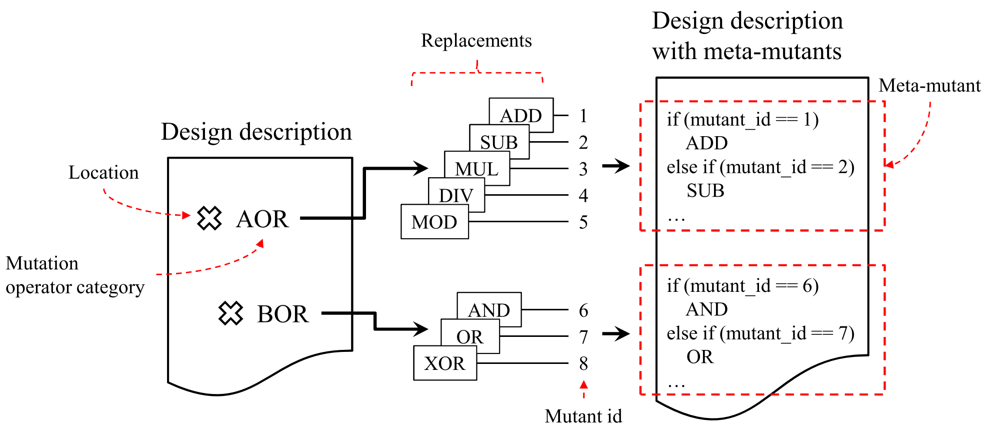


Fig. 2 The mutant-injection process. For each candidate location in the original design description, replacements are identified according to Table 1. The replacements generate mutants of the code and the resulting unique meta-mutated description is parameterized over the mutant id

leads to a number of implications and considerations that will be described in the following.

The first scenario (at the left-hand side of Fig. 3) features the reuse of the TLM testbench in order to detect possible errors introduced by the synthesis process, which is bound to be slow given the presence of injected mutants within the input code. In this case, mutants being tested are synthesized, and correspond to altered blocks of functionality passed down to a lower abstraction level. The generation of stimuli at TLM is fast and it provides a test bench for verifying the functionality. If these stimuli were simulated at the RTL then the simulation time would be significantly higher. Thus, the left-hand of Fig. 3 suggests that functionality verification should be addressed at TLM.

On the other hand, the second scenario (the right-hand side of Fig. 3) focuses on the synthesized mutant-free description, which is perturbed by injecting mutants altering the low-level functionality. Synthesis in this case is bound to be much faster, but the mutation process may not be accurate enough to focus on design errors at the level of RTL architecture. The same stimuli permit to kill part of the mutants injected at TLM, but significantly less with respect to the left-hand scenario in terms of percentage. In order to increase this percentage, the test bench has to be improved in order to kill the mutants representing the RTL architectural constructs. In other words, on the right-hand scenario, where we inject mutants at RTL, the verification is focusing on the "chosen architecture", or better, on the result of the high-level synthesis.

Moreover, following the native-RTL mutation analysis (right side of Fig. 3) we can observe that mutant injection at RTL results generally in a larger number of mutants with respect to TLM injection, because of an increase in the number of candidate locations in the RTL description. On the contrary, mutants in the RTL code generated by synthesizing the TLM mutated design (left side of Fig. 3) is unchanged with respect to the number of mutants injected in the original TLM code. However, the mutation coverage achieved by the native-RTL mutation analysis tends to be lower with respect to one achieved by the TLM-derived mutation analysis, when the TLM testbench is reused at RTL. Typically, verification engineers manually generate TLM testbenches as test scenarios from the design specifications and test plans: such testbenches stress the design by applying valid values from the input domains.

Figure 4 provides a visual explanation of why the number of mutants injected at RTL increases. According to the complexity of the statement blocks making up the high-level description of the functionality, the synthesis process may produce corresponding portions of RTL code at different levels of abstraction.

The synthesis tool may be able to decompose a given C++ statement into a proper connection of basic RTL components such as multiplexers and adders (upper-left corner of Fig. 4). In this scenario, the corresponding generated RTL code is expected to be much larger, given the addition of implementation details and the mapping to these basic components. As such, the number of injection locations will greatly increase, thus leading to the injection of a greater number of mutants with respect to the TLM description.

On the other hand, there may be cases where only a partial decomposition may be possible. In these cases, the generated RTL code will contain a mixture of constructs representing basic RTL components and high-level constructs (e.g. if-then-else statements). The right corner of Fig. 4 represents this scenario. Even in this case, the number of injection locations will increase as a consequence of this mixture.

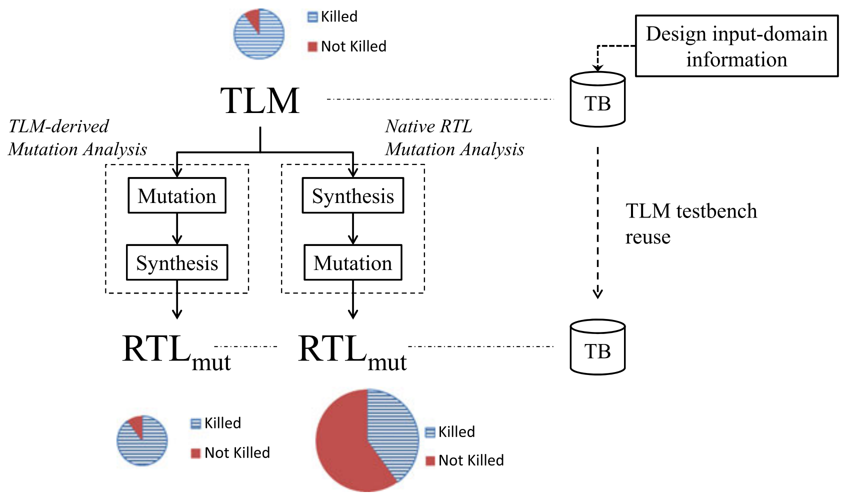
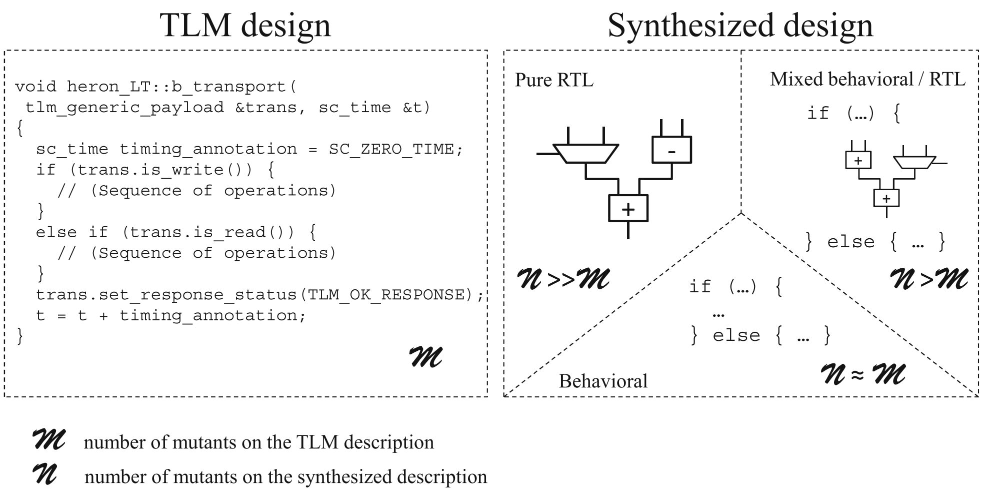


Fig. 3 TLM testbench reuse scenarios and corresponding mutation coverage. A higher number of mutants occurs along the native RTL mutation analysis, i.e., synthesis followed by mutation. Testbenches are generated at transaction level starting from the design specifications and test plan, which typically report input-domain information, e.g., valid values for the inputs parameters



```
void heron_LT::b_transport(
 tlm_generic_payload &trans, sc_time &t)
{
  sc_time timing_annotation = SC_ZERO_TIME;
  if (trans.is_write()) {
    // (Sequence of operations)
  }
  else if (trans.is_read()) {
    // (Sequence of operations)
  }
  trans.set_response_status(TLM_OK_RESPONSE);
  t = t + timing_annotation;
}
```

TLM design
Synthesized design
Pure RTL
Mixed behavioral / RTL
if (…) {
} else { … }
$\mathcal{N} \gg \mathcal{M}$
$\mathcal{N} > \mathcal{M}$
if (…) {
…
} else { … }
Behavioral
$\mathcal{N} \approx \mathcal{M}$
$\mathcal{M}$
$\mathcal{M}$ number of mutants on the TLM description
$\mathcal{N}$ number of mutants on the synthesized description

Fig. 4 Relationship between number of mutants on the TLM and the synthesized descriptions. According to the complexity of the TLM statements, the synthesis process may produce RTL code at different levels of abstraction. Typically, a pure RTL, i.e., low level, description provides an significantly higher number of possible mutations than the corresponding high-level code

If even a partial decomposition is not feasible or necessary, the generated RTL code will contain high-level constructs providing a behavioral description of the statement (bottom corner of Fig. 4). In this case, it is perfectly reasonable to expect the number of injection locations to stay at least the same as in the TLM description. A minor increase may still be possible due to transformations introduced at the level of operations and sub-expressions.

Overall, this justifies the increase in the number of injected mutants at RTL. Experimental evidence of this claim is to be found in Section 5.

Concerning the low mutation coverage achieved by reusing TLM testbeches on the RTL code obtained according to the left flow of Fig. 4, a possible motivation lies in the way the TLM testbench is built. A further reason for having many live mutants is to be found in the implementation of RTL architectural choices provided by designers to the HLS tool. In particular, analysing the live mutants (i.e., mutants that have not been detected by the testbenches) we observed that they fall under one of the following categories:

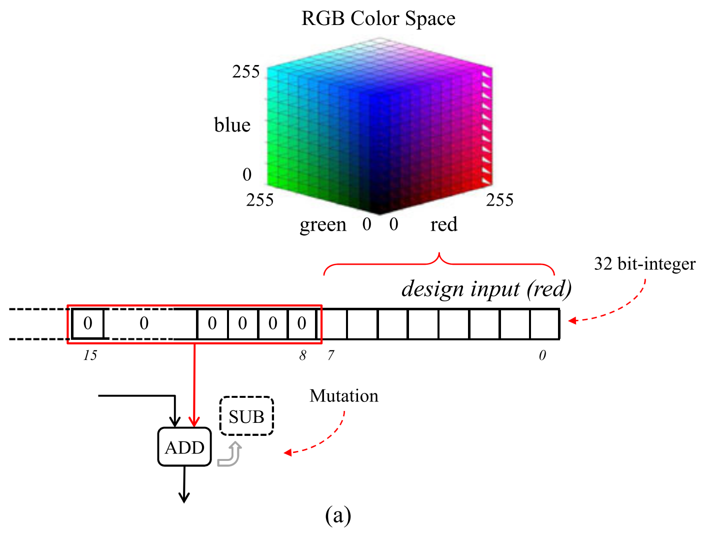
– the mutant depends (directly or indirectly) on a range of bits of input ports which are never set by the testbench;
– the mutant operates on a range of bits of an intermediate result which does not propagate to the outputs, because of subsequent operations discarding such range;
– the mutant alters code deriving from RTL architectural choices.

The first category is a consequence of the way a testbench provides input data. It relies on design input-domain information which limits the range of values provided to inputs, in order to simulate feasible use cases pertaining to a real application. Hence, it provides values within this range for all the inputs. However, this information is not provided to the automatic synthesis tool in the starting TLM description, since it is hardly available yet. This is mostly due to the lack of bit-accuracy information at TLM. As such, the synthesized RTL description does not reflect such a knowledge. This results in overestimating the RTL space of computation, thus introducing blocks of code which are not activated by the inputs provided by the testbench. Mutants introduced in these blocks are therefore bound to be never activated and then never killed.

The second category is directly related to the way automatic synthesis is performed. As previously stated, any HLS tool adds signals to the design in order to store and to accumulate intermediate results. Bit ranges of these signals are then used to compute the final results. Many times, these signals have a larger bit width than the final result, to provide better accuracy while performing intermediate computations. As such, a mutant may operate on a bit range of one of these intermediate signals, and this range ends up being discarded during the operations that lead to the final result.

The third category pertains to those code blocks that are introduced by the HLS tool to properly implement the desired RTL architectural choices. Changes applied in this context may not produce a corresponding alteration to the functional behavior. Furthermore, the testbench may not have been built to accurately stimulate such features.

A module performing conversion from a color space to another one provides an example for the first category, as shown in Fig. 5a. A mutant may replace the addition operator with the subtraction operator in an expression being assigned to an inner signal. However, the second operand contains the bit range



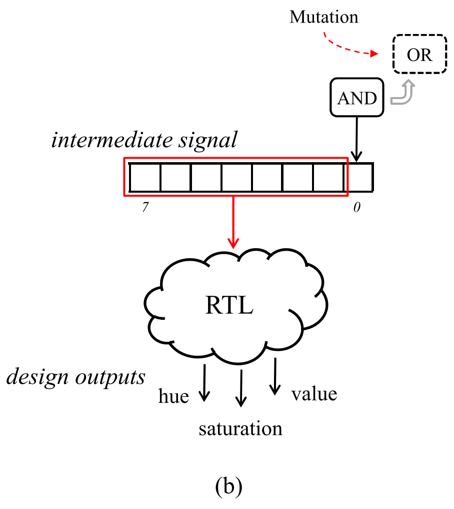


Fig. 5 Examples of live-mutants causes for native RTL mutation analysis. **a** The mutant depends on a bit range of the integer input port which is never assigned by the testbench: by default integers in C/C++ are initialized to 0. **b** The mutant operates on a bit which does not propagate to the outputs

15 down-to 8 (i.e., [256,32767]) of one of the input ports. Since the design under verification performs conversion from a color space to another, the inputs—as well as the outputs—are limited to a specific range, namely 7 down-to 0 (i.e, [0,255]). As such, the second operand in the expression will always evaluate to 0, thus making it impossible to detect the mutant under these circumstances. In this case, the problem lies in the lack of proper bit-accuracy information provided to the HLS tool. In the TLM version, inputs and outputs were represented by using the generic **int** type, even though a much smaller subset of its possible values were being used. As such, the HLS tool conservatively synthesizes by assuming that all the 32 bits of the **int** type may be used. This leads to overestimating bit widths for inputs, outputs and inner signals.

Color-space-conversion module provides an example for the second category as well, as shown in Fig. 5b. A mutant may alter only the least significant bits of an intermediate signal, in position 7 down-to 0. The intermediate signal being written is among the first ones in the code, i.e., its assignment is one of the first operations being performed. As computation progresses, new intermediate signals are written, taking into account bit ranges of previously assigned intermediate signals. By the time output ports are written, the least significant bits altered by the mutant may end up being ignored because of subsequent operations performed to produce the final result. In this case, the mutant cannot be detected, since the change performed in the intermediate signal does not propagate to the output.

Given these premises, the lower mutation coverage achieved at RTL has to be properly interpreted. This decrease seems to suggest a weakness in the testbench being employed, but the analysis of the reasons for live mutants points into a different direction. The automatic synthesis process overestimates bit widths because of the lack of bit-accuracy information in the starting TLM description. This produces code blocks which are not stimulated by the testbench being used.

Moreover, changes introduced by mutants injected directly at RTL operate on a too fine granularity to be mapped back to possible design errors. As previously stated, C++ statements are decomposed into smaller and simpler operations at RTL by the synthesis process. RTL mutants alter only one of these operations at a time. As such, the change in behavior they introduce may only affect a small fraction of the original computation, and most likely cannot be traced back to a corresponding alteration in the starting C++ code. This

Table 2 Experimental results for the TLM mutation analysis

Design	hsv2rgb	rgb2hsv	rgb2ycbcr	ycbcr2rgb	Line	Heron	Dayofweek
# of mutants	87	95	53	43	33	48	43
# killed	83	84	49	43	33	48	43
Mut. coverage	98%	88%	92%	100%	100%	100%	100%
# of lines	413	712	272	242	289	255	257
Sim. time (s)	0.004	0.008	0.004	0.004	0.004	0.004	0.008

Table 3 Experimental results for the TLM-derived mutation analysis

Design	hsv2rgb	rgb2hsv	rgb2ycbcr	ycbcr2rgb	Line	Heron	Dayofweek
# of mutants	87	95	53	43	33	48	43
# killed	86	83	49	43	33	48	43
Mut. coverage	98%	87%	92%	100%	100%	100%	100%
# of lines	5282	7188	7628	3206	1183	3417	13296
Sim. time (s)	0.784	2.502	0.304	0.088	0.028	0.184	0.116
Synth. time (s)	117.04	553.56	698.18	89.35	19.49	175.10	5355.22

is because no operator having such a slightly deviated behavior is defined and available at this higher abstraction level. Hence, these mutants cannot represent actual design errors.

For these reasons, live mutants belonging to the first two categories end up being not relevant to the purpose of mutation analysis focused on possible design errors.

Mutants belonging to the third category actually go beyond the scope of this article. Future work will explore a way to selectively inject mutants on code blocks generated by the HLS tool to implement architectural choices, in order to investigate on possible errors introduced in this context. In this way, also low-level behaviors strictly related to the RTL implementation can be actually verified.

5 Experimental Confirmation

The confirmation of observations reported in Section 4.3 has been carried out by performing mutation analysis on the following seven designs:

- *hsv2rgb*: performs color conversion from the HSV color space to the RGB color space;
- *rgb2hsv*: performs color conversion from the RGB color space to the HSV color space;
- *rgb2ycbcr*: performs color conversion from the RGB color space to the YCbCr color space;
- *ycbcr2rgb*: performs color conversion from the YCbCr color space to the RGB color space;
- *line*: computes the standard equation of the line passing between two points;
- *heron*: computes the area of a triangle by using Heron's formula;
- *dayofweek*: computes the day of week for a given date.

For each design, the following three versions were considered:

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

Experiments were carried out by injecting mutants on each version for each design, and then simulating them to compute mutation coverage.

We have adopted a random approach for testbench generation and we use mutation analysis to judge the adequacy of testbenches. The testbenches are judged adequate only if at least one of the non-yet-killed mutants compute outputs different from the original design. In that sense, adopting manually- or automatically-generated approach for testbench generation is only an efficiency matter. In any case, the mutation-analysis-based approach guarantees that the final testbench is able to stress most of the system functionalities.

Results for the *TLM mutation analysis* are shown in Table 2. Table 3 lists results for the *TLM-derived mutation analysis*, while Table 4 details results for the *native RTL mutation analysis*.

Table 4 Experimental results for the native-RTL mutation analysis

Design	hsv2rgb	rgb2hsv	rgb2ycbcr	ycbcr2rgb	Line	Heron	Dayofweek
# of mutants	220	243	180	168	33	58	112
# killed	101	122	154	118	33	58	108
Mut. coverage	45%	50%	86%	70%	100%	100%	96%
# of lines	3356	6241	5803	4452	518	1014	4092
Sim. time (s)	5.108	7.841	4.512	2.188	0.016	0.060	0.416
Synth. time (s)	13.65	8.72	12.18	8.68	6.77	9.01	10.87

Table 5 Speed-up of mutation analysis for TLM derived and native TLM with respect to native RTL

Design	TLM derived	Native TLM
Dayofweek	13.5	60,290
Heron	0.30	2,246
hsv2rgb	8.87	89,111
Line	0.70	665
rgb2hsv	0.01	183
rgb2ycbcr	4.70	13,317
ycbcr2rgb	9.96	32,961

In each Table, rows # *of mutants* and # *killed* indicate the number of injected mutants and the number of killed mutants, respectively. Row *Mut. coverage* shows the mutation coverage, while row # *of lines* indicates the number of lines of code in the description. Rows *Sim. time* and *Synth. time* provide simulation and synthesis time, elapsed in seconds.

Number of injected mutants and mutation coverage are the same in Tables 2 and 3. This was expected, since Mentor Graphics Catapult C preserves functional equivalence in its synthesis process. As such, if the injected TLM description is provided as input, its RTL synthesized version will reproduce its complete behavior, thus including all previously injected mutants.

Since mutation analysis based on meta-mutants introduces a significant number of additional computational paths in control flow, the synthesis time for the RTL synthesized from injected TLM version (Table 3) is much higher than the one for the RTL directly injected version (Table 4). The same applies to the number of lines of code. In fact, the mutant-free TLM description in most cases contains a single control flow, which is much easier to follow during the synthesis process than having to deal with all the possible branches in control flow introduced by injected mutants.

Simulation time in the three versions is a direct consequence of three factors, abstraction level, number of injected mutants and mutation coverage. TLM simulation is much faster than its RTL counterpart. Table 5 reports the speed-ups of mutation analysis for TLM derived and native TLM with respect to native RTL. An increase in the number of injected mutants corresponds to an increase in simulation time, since each mutant requires the simulation of at least one test vector to be killed. Simulation is performed so that once a difference in the outputs is observed between the mutant-free version and the mutated version, the mutant is reported to be killed and the testbench moves to the activation of the next mutant. As such, live mutants take up the vast majority of simulation time, since they require the simulation of all the test vectors provided by the testbench.

For each design, mutation coverage in the native RTL mutation analysis is less than or equal to the other two versions. The decrease in coverage is more sensible in the first four designs, which perform conversion between color spaces.

Results suggest that the TLM-derived mutation analysis flow provides a solid verification of the high-level functionality. On the other hand, the native RTL mutation analysis flow is affected by observations made in Section 4.3, which are confirmed and backed up by these experimental results.

Experiments point out that the native RTL mutation analysis flow does not provide useful information about correctness of the synthesis. This is essentially due to the automatic synthesis process producing large and unreadable code. As such, any link to high-level functionality is bound to be lost, making it almost impossible to establish a relationship between a mutant directly injected at RTL and the change it causes in the high-level functionality. Therefore, mutation coverage in the native RTL flow is bound to be not quite as meaningful as in the TLM-derived flow.

In this context, experiments highlight the need for a further test generation phase for RTL-native mutation analysis. This is summarized in Table 6. Designs *hsv2rgb*, *rgb2hsv*, *rgb2ycbcr*, and *ycbcr2rgb* present low native-RTL coverage (see Table 4). For them, we performed an additional test campaign at RT-level focusing on the mutations associated with the architectural choices of the high-level synthesis process. Row # *killed* reports the total number of killed mutants and, in brackets, the extra mutants killed by RT test. Row *Mut. coverage* reports the final mutation coverage. Finally, *TGen. time* reports the additional test-generation

Table 6 Experimental results for the additional native-RTL mutation analysis which addresses the architectural aspects introduced by the high-level synthesis process

Design	hsv2rgb	rgb2hsv	rgb2ycbcr	ycbcr2rgb	Line	Heron	Dayofweek
# of mutants	220	243	180	168	33	58	112
# killed	178 (77)	243 (121)	163 (9)	161 (43)	33 (0)	58 (0)	108 (0)
Mut. coverage	81%	100%	91%	96%	100%	100%	96%
TGen. time (s)	1381.22	16.59	434.13	280.17	–	–	–

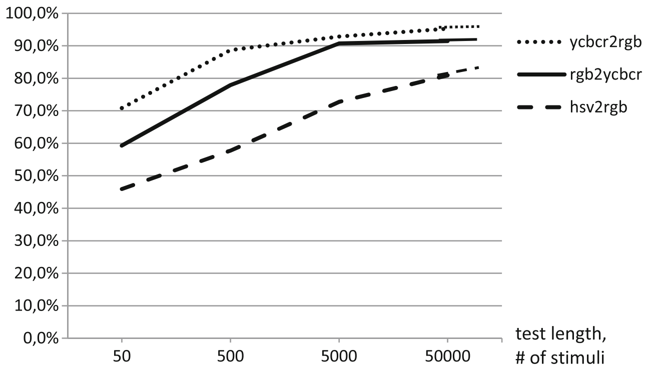


Fig. 6 RTL-native mutation coverage depending on the test length

time required by the RT-level test phase. As well, Fig. 6 reports the trends of the RTL-native mutation coverage during the RTL-test campaign. At RT-level the generation of test is significantly slower than at TLM. Although we adopted a random approach for test generation, the use of mutation analysis to judge the adequacy of testbenches require significantly time. Thus, the proposed re-use of TLM testbenches at RT level confirms its efficiency in terms of both simulation time and reached mutation coverage.

6 Concluding Remarks

RTL mutation analysis can be done by injecting mutants directly on the RTL models (native RTL mutation analysis), or by injecting mutants on the TLM descriptions and then synthesizing the corresponding RTL mutated models (TLM-derived mutation analysis). This paper showed that the second alternative provides several advantages with respect to the first.

At the cost of a slower synthesis process, the TLM-derived mutation analysis has faster simulation time. Moreover, we showed that TLM testbenches can be efficiently reused in TLM-derived mutation analysis. They achieve the same mutant coverage at RTL as it is achieved on the TLM design. On the contrary, the reuse of TLM testbenches in the native RTL mutation analysis provides us with apparently worse results. However, the decrease observed in native RTL mutant coverage has to be properly interpreted: it does not mean that the quality of TLM testbenches is low. Indeed, it is mainly due to the bit width overestimation performed by the automatic synthesis process, caused by the lack of bit-accuracy information in the initial TLM description, as detailed in Section 4.3.

Finally, we elaborated about the capability of TLM-derived mutation analysis of preserving the mapping between TLM and RTL mutants, thus, allowing to more easily identify possible problems in the synthesis process. Contrary to the TLM-derived mutation, in the native RTL mutation analysis the link to TLM functionality is lost, making it almost impossible to establish a relationship between a mutant directly injected at RTL and the change it causes with respect to to the original TLM functionality.

Acknowledgements The work has been supported by FP7 DIAMOND project, CEBE Center of Excellence and Estonian SF grants 8478 and 9429.

References

1. Abramovici M, Breuer M, Friedman A (1990) Digital systems testing and testable design. Computer Science Press, New York
2. Agrawal H, DeMillo RA, Hathaway B, Hsu W, Hsu W, Krauser EW, Martin RJ, Mathur AP, Spafford E (1989) Design of mutant operators for the C programming language. Purdue University, West Lafayette, Indiana, techreport SERC-TR-41-P
3. Alexander RT, Bieman JM, Ghosh S, Bixia J (2002) Mutation of Java objects. In: Proc. of IEEE ISSRE, pp 341–351
4. Belli F, Budnik C-J, Wong W-E (2006) Basic operations for generating behavioral mutants. In: Proc. of IEEE ISSRE, pp 10–18
5. Bombieri N, Fummi F, Pravadelli G (2006) On the evaluation of transactor-based verification for reusing TLM assertions and testbenches at RTL. In: Proc. of ACM/IEEE conference on design, automation and test in Europe, DATE, pp 1007–1012
6. Bombieri N, Fummi F, Pravadelli G (2008) A mutation model for the SystemC TLM 2.0 communication interfaces. In: Proc. of ACM/IEEE conference on design, automation and test in Europe, DATE, pp 396–401
7. Bombieri N, Fummi F, Pravadelli G (2009) On the mutation analysis of SystemC TLM-2.0 standard. In: Proceedings of IEEE international workshop on microprocessor test and verification, MTV, pp 32–37
8. Bombieri N, Fummi F, Pravadelli G, Hampton M, Letombe F (2009) Functional qualification of TLM verification. In: Proc. of the ACM/IEEE conference on design, automation and test in Europe, DATE, pp 190–195
9. Bradbury JS, Cordy JR, Dingel J (2006) Mutation operators for concurrent Java (J2SE 5.0). In: Proc. of IEEE ISSRE workshops, pp 11–20
10. Budd TA, Sayward FG (1977) Users guide to the Pilot mutation system. Yale University, New Haven, Connecticut, Technical report 114
11. Budd TA, DeMillo RA, Lipton R, Sayward F (1980) Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In: Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on principles of programming languages. ACM, pp 220–233
12. Cai L, Gajski DD (2003) Transaction level modeling: an overview. In: ACM/IEEE CODES+ISSS, pp 19–24
13. Catapult C Synthesis (2010). Mentor graphics. http://www.mentor.com/esl/catapult

14. Choi BJ, DeMillo RA, Krauser EW, Martin RJ, Mathur AP, Offutt AJ, Pan H, Spafford EH (1989) The Mothra tool set. In: Proceedings of the 22nd annual Hawaii international conference on system sciences (HICSS), pp 275–284
15. Coussy P, Gajski DD, Meredith M, Takach A (2009) An introduction to high-level synthesis. IEEE Des Test Comput 13(24):8–17
16. Cynthesizer - TLM Synthesis (2008). Forte Design Systems. http://www.forteds.com/products/tlmsynthesis.asp
17. DeMillo RA, Lipton RJ, Sayward FG (1978) Hints on test data selection: help for the practicing programmer. Computer 11(4):34–41
18. Do H, Rothermel G (2006) On the use of mutation faults in empirical assessments of test case prioritization techniques. IEEE Trans Softw Eng 32(9):733–752
19. Guarnieri V, Bombieri N, Pravadelli G, Fummi F, Hantson H, Raik J, Jenihhin M, Ubar R (2011) Mutation analysis for SystemC designs at TLM. In: Proc. of IEEE Latin-American test workshop (LATW), pp 27–30
20. Guderlei R, Just R, Schneckenburger C, Schweiggert F (2008) Benchmarking testing strategies with tools from mutation analysis. In: International conference on software testing verification and validation workshop. IEEE, pp 360–364
21. Hamlet RG (1977) Testing programs with the aid of a compiler. IEEE Treans Softw Eng 3(4):279–290
22. Hantson H, Raik J, Jenihhin M, Chepurov A, Ubar R, di Guglielmo G, Fummi F (2010) Mutation analysis with high-level decision diagrams. In: Test workshop (LATW), 2010 11th Latin American, pp 1–6
23. Hyunsook D, Rothermel G (2006) On the use of mutation faults in empirical assessments of test case prioritization techniques. IEEE Trans Softw Eng 32(9):733–752
24. Irvine SA et al (2007) Jumble Java byte code to measure the effectiveness of unit tests. In: Mutation testing workshop, pp 169–175
25. Lipton R (1971) Fault diagnosis of computer programs. Carnegie Mellon University, Student report
26. Lisherness P, Cheng K-T (Tim) (2010) SCEMIT: a SystemC error and mutation injection tool. In: Proc. of ACM/IEEE design automation conference (DAC), pp 228–233
27. Lyu M-R, Zubin H, Sze SKS, Xia C (2003) An empirical study on testing and fault tolerance for software reliability engineering. In: Proc. of IEEE ISSRE, pp 119–130
28. Ma Y-S, Offutt AJ, Kwon YR (2005) MuJava: an automated class mutation system: research articles. Softw Test Verif Reliab 15:97–133
29. Offutt AJ, King KN (1987) A Fortran 77 interpreter for mutation analysis. In: Papers of the symposium on interpreters and interpretive techniques. SIGPLAN '87, pp 177–188
30. Offutt AJ, Rothermel G, Zapf C (1993) An experimental evaluation of selective mutation. In: Proceedings of the 15th international conference on software engineering (ICSE'93), Baltimore, Maryland, pp 100–107
31. Offutt AJ, Lee A, Rothermel G, Untch R, Zapf C (1996) An experimental determination of sufficient mutant operators. ACM Trans Softw Eng Methodol 5(2):99–118
32. Offutt AJ, Voas J (1996) Subsumption of condition coverage techniques by mutation testing. Department of Information and Software Systems Engineering, George Mason University, Technical report ISSE-TR-96-01
33. Sen A (2009) Mutation operators for concurrent SystemC designs. In: Proc. of IEEE international workshop on microprocessor test and verification. MTV, pp 27–31
34. Sen A, Abadir MS (2010) Coverage metrics for verification of concurrent SystemC designs using mutation testing. In: Proc. of IEEE international high-level design, validation, and test workshop, pp 75–81
35. Tuya J, Suarez-Cabal MJ, De La Riva C (2006) SQLMutation: a tool to generate mutants of SQL database queries. In: Mutation testing workshop, pp 39–43

Valerio Guarnieri received the Master degree in Computer Science from the Università di Verona, in 2009. He has been a PhD student in the Dipartimento di Informatica at the Università di Verona since 2010. His main research interests are in electronic design automation methodologies for design and verification of TLM-based designs. He received a "best paper award" at IEEE FDL '11. He is member of the IEEE.

Giuseppe Di Guglielmo received the Laurea degree in Computer Science in 2005 and the PhD degree in Computer Science in 2009, both from the Università di Verona. He has been a research assistant at the Dipartimento di Informatica of the Università di Verona since 2009. His research interests include verification of embedded systems and EDA methodologies for hardware/software system modeling. He is a member of the IEEE.

Nicola Bombieri received the laurea degree and the PhD in Computer Science from the University of Verona in 2004 and 2008, respectively. Since 2008, he is researcher and professor assistant at the Department of Computer Science of the University of Verona. His research activity focuses on design and verification of embedded systems in the design flows based on transaction level modeling (TLM), and automatic generation and optimization of embedded SW. He has been involved in several national and international research projects and has published more than 40 papers on conference proceedings and journals. He is a member of the IEEE.

Graziano Pravadelli received the Laurea degree and the PhD degree in computer science from the Università di Verona, respectively, in 2001 and 2004. He has been an associate professor in the Dipartimento di Informatica at the Università di Verona since January 2011. His main research interests are in hardware description languages and electronic design automation methodologies for modeling and verification of hardware/software systems, in which area he has published more than 80 papers. He is a member of the IEEE.

Franco Fummi received the Laurea degree in Electronic Engineering in 1990 and the PhD degree in Electronic Engineering in 1995, both from the Politecnico di Milano. He has been a full professor in the Dipartimento di Informatica of the Università di Verona since 2001. His main research interests are in hardware description languages and electronic design automation methodologies for modeling, verification, testing, and optimization of hardware/software systems. He has published more than 230 papers in the EDA field; three of them received "best paper awards" at, respectively, IEEE EURODAC '96, IEEE DATE '99, and IEEE FDL '11. Since 1995, he has been with the Dipartimento di Elettronica e Informazione of the Politecnico di Milano with the position of assistant professor. In July 1998, he was promoted to the position of associate professor in computer architecture in the Dipartimento di Informatica at the Università di Verona. He is a member of the IEEE and a member of the IEEE Test Technology Committee.

Hanno Hantson is a PhD student at Tallinn University of Technology and a member of IEEE. His research interest is mutation analysis methods. He has co-authored two papers.

Jaan Raik received his M.Sc. and Ph.D. degrees in Computer Engineering from Tallinn University of Technology in 1997 and in 2001, respectively, where he currently holds the position of a senior research fellow. He is a member of IEEE Computer Society, a member of program committees for several top-level conferences and has co-authored more than 100 scientific publications. In 2004, he was awarded the national Young Scientist Award. Starting from 2010 he acts as the scientific co-ordinator of the EU FP7 DIAMOND research project. His main research interests include high-level test generation and verification.

Maksim Jenihhin received his M.Sc. and PhD degrees in Computer Engineering from Tallinn University of Technology (TUT) in 2004 and in 2008, respectively. He is a member of IEEE Computer Society. His research interests include verification and debug as well as test and EDA methodologies. Currently he is a senior research fellow at TUT. He has co-authored more than 50 papers.

Raimund Ubar received his Ph.D. degree in 1971 at the Bauman Technical University in Moscow. He is a professor of Computer Engineering at Tallinn University of Technology. His research interests include computer science, electronics design, design verification, test generation, fault simulation, design-for-testability, fault-tolerance. He has published more than 200 papers and two books. R. Ubar has given seminars or lectures in 20–25 universities in more than ten countries. In 1993–1996 he was the Chairman of the Estonian Science Foundation and a member of the Estonian Science Council. He is a Golden Core Member of the IEEE, a member of ACM, SIGDA, Gesellschaft der Informatik (Information Society, Germany), European Test Technology Technical Committee and Estonian Academy of Sciences.