THESIS ON INFORMATICS AND SYSTEM ENGINEERING C106

Mutation-Based Verification and Error Correction in High-Level Designs

HANNO HANTSON

TUT PRESS

TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology
Department of Computer Engineering

Dissertation was accepted for the defence of the degree of Doctor of Philosophy in Computer and System Engineering on October 14, 2015.

Supervisor: Prof. Jaan Raik
Department of Computer Engineering
Tallinn University of Technology, Estonia

Opponents: Dr. Maria K. Michael
Department of Electrical and Computer Engineering
University of Cyprus, Cyprus

Prof. Heinrich Theodor Vierhaus
Department of Computer Engineering
Brandenburg University of Technology, Cottbus, Germany

Defence of the thesis: November 12, 2015

Declaration:
Hereby I declare that this doctoral thesis, my original investigation and achievement, submitted for the doctoral degree at Tallinn University of Technology has not been submitted for any academic degree.

/Hanno Hantson/

ISSN 1406-4731
ISBN 978-9949-23-853-8 (publication)
ISBN 978-9949-23-854-5 (PDF)

INFORMAATIKA JA SÜSTEEMITEHNIKA C106

Mutatsioonidel põhinev verifitseerimine ja vigade parandamine kõrgtaseme skeemides

HANNO HANTSON

TTÜ
KIRJASTUS

To my wife Kätlin,
daughter Sandra and son Hugo

Table of Contents

List of Publications

Publications included in the Appendices with author's contributions

I. Hantson, Hanno; Raik, Jaan; di Guglielmo, Giuseppe; Jenihhin, Maksim; Chepurov, Anton; Fummi, Franco; Ubar, Raimund. "Mutation Analysis with High-Level Decision Diagrams". Proceedings of the 11th Latin-American Test Workshop, IEEE Computer Society Press, 2010, pp. 1–6.

Contributes to Section 3.1 of this Thesis. The author's contributions are: participating in development of the HLDD-based mutation analysis method, implementing mutation analysis tool to the Apricot framework and presenting the paper at 11th Latin-American Test Workshop.

II. Guarnieri, Valerio; Di Guglielmo, Giuseppe; Bombieri, Nicola; Pravadelli, Graziano; Fummi, Franco; Hantson, Hanno; Raik, Jaan; Jenihhin, Maksim; Ubar, Raimund. "On the Reuse of TLM Mutation Analysis at RTL". Journal of Electronic Testing-Theory and Applications, 28(4), 2012, pp. 435–448.

Contributes to Section 3.2 of this Thesis. The author's contributions are: participating in development of the RTL-TLM-based mutation analysis method, performing experiments using Mentor Graphics CatapultC software and presenting the paper at 12th Latin-American Test Workshop. Paper II was an extended version of the latter.

III. Raik, Jaan; Repinski, Urmas; Tšepurov, Anton; Hantson, Hanno; Ubar, Raimund; Jenihhin, Maksim. "Automated design error debug using high-level decision diagrams and mutation operators". Microprocessors and Microsystems: Embedded Hardware Design, 37(4), 2013, pp. 1–10.

Contributes to Section 4.1 of this Thesis. This paper was based on author's work on mutation analysis developed in Paper I.

IV. Jenihhin, Maksim; Tšepurov, Anton; Tihhomirov, Valentin; Hantson, Hanno; Raik, Jaan; Ubar, Raimund; Bartsch, Gu¨nter; Meza-Escobar, Jorge Hernan; Wuttke, Heinz-Dietrich. "Automated Design Error Localization in RTL Designs". IEEE Design & Test of Computers, 1, 2014, pp.83–92.

Contributes to Section 4.2 of this Thesis. The author's contributions are: performing experiments using Apricot software and presenting the paper at 13th Latin-American Test Workshop (Best Paper Award). Paper IV was an extended version of the latter.

V. Raik, Jaan; Repinski, Urmas; Hantson, Hanno; Jenihhin, Maksim; Di Guglielmo, Giuseppe; Pravadelli, Graziano; Fummi, Franco. "Combining Dynamic Slicing and Mutation Operators for ESL Correction". Proceedings of the 17th IEEE European Test Symposium, IEEE Computer Society Press, 2012, pp. 1–6.

Contributes to Section 4.3 of this Thesis. The author's contributions are: developing the mutation-based fault model in cooperation with Giuseppe Di Guglielmo from the Univesity of Verona during the author's stay in Verona, proposing an improved classification of faults.

Full list of author's publications

Journals

1. IV. Jenihhin, Maksim; Tšepurov, Anton; Tihhomirov, Valentin; Hantson, Hanno; Raik, Jaan; Ubar, Raimund; Bartsch, Gu¨nter; Meza-Escobar, Jorge Hernan; Wuttke, Heinz-Dietrich. "Automated Design Error Localization in RTL Designs". IEEE Design & Test of Computers, 1, 2014, pp.83–92.
2. III. Raik, Jaan; Repinski, Urmas; Tšepurov, Anton; Hantson, Hanno; Ubar, Raimund; Jenihhin, Maksim. "Automated design error debug using high-level decision diagrams and mutation operators". Microprocessors and Microsystems: Embedded Hardware Design, 37(4), 2013, pp. 1–10.
3. II. Guarnieri, Valerio; Di Guglielmo, Giuseppe; Bombieri, Nicola; Pravadelli, Graziano; Fummi, Franco; Hantson, Hanno; Raik, Jaan; Jenihhin, Maksim; Ubar, Raimund. "On the Reuse of TLM Mutation Analysis at RTL". Journal of Electronic Testing-Theory and Applications, 28(4), 2012, pp. 435–448.

Conferences

4. Hantson, Hanno; Repinski, Urmas; Raik, Jaan; Jenihhin, Maksim; Ubar, Raimund. “Diagnosis and Correction of Multiple Design Errors Using Critical Path Tracing and Mutation Analysis”. Proceedings of the 13th IEEE Latin-American Test Workshop. IEEE Computer Society Press, 2012, pp. 27–32, *Best Paper Award*.
5. Raik, Jaan; Repinski, Urmas; Hantson, Hanno; Jenihhin, Maksim; Di Guglielmo, Giuseppe; Pravadelli, Graziano; Fummi, Franco. “Combining Dynamic Slicing and Mutation Operators for ESL Correction”. Proceedings of the 17th IEEE European Test Symposium, IEEE Computer Society Press, 2012, pp. 1–6.
6. Guarnieri, Valerio; Hantson, Hanno; Raik, Jaan; Jenihhin, Maksim; Bombieri, Nicola; Pravadelli, Graziano; Fummi, Franco; Ubar, Raimund. “Mutation Analysis for SystemC Designs at TLM”. Proceedings of the 12th IEEE Latin-American Test Workshop Proceedings (1 - 6), IEEE Computer Society Press, 2011, pp. 1–6.
7. Hantson, Hanno; Raik, Jaan; di Guglielmo, Giuseppe; Jenihhin, Maksim; Chepurov, Anton; Fummi, Franco; Ubar, Raimund. “Mutation Analysis with High-Level Decision Diagrams”. Proceedings of the 11th Latin-American Test Workshop, IEEE Computer Society Press, 2010, pp. 1–6.

List of Abbreviations

ADD	Assignment Decision Diagram
ALU	Arithmetic-Logic Unit
AST	Abstract Syntax Tree
BDD	Binary Decision Diagram
CRC	Cyclic Redundancy Check
DD	Decision Diagram
DFT	Design For Testability
DUV	Design Under Verification
FIFO	First In, First Out
FSM	Finite State Machine
GCC	GNU Compiler Collection
GCD	Greatest Common Divisor
GNU	GNU Is Not Unix (Operating System)
HDL	Hardware Description Language
IC	Integrated Circuit
IG	Instantiation Graph
IoT	Internet of Things
ISA	Instruction Set Architecture
ITRS	International Technology Roadmap for Semiconductors
ESL	Electronic System Level
HLDD	High-Level Decision Diagram
K*BMD	Kronecker Multiplicative Binary Moment Diagram
LoC	Lines of Code
MTDD	Multi-Terminal Decision Diagram

OSCI	Open SystemC Initiative
PCB	Printed Circuit Board
RISC	Reduced Instruction Set Computer
ROBDD	Reduced Ordered Binary Decision Diagram
ROBSY	Reconfigurable On Board self test SYstem
RTL	Register-Transfer Level
SAT	SATisfiability
SISD	Single Instruction Single Data
SMT	SAT Modulo Theory
SoC	System-on-Chip
TLM	Transaction Level Modeling
UML	Unified Modeling Language
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
WB	Wishbone protocol
ZDB	zamiaCAD Data Base

List of Figures

1 Introduction

Immense development of technology has led us to an era where computer electronics is part of virtually everything. New data from Juniper Research has revealed that the number of IoT (Internet of Things) connected devices will number 38.5 billion in 2020, up from 13.4 billion in 2015: a rise of over 285% (Juniper Research, 2015).

One thing common to all of those devices is that everyone expects them to work. And not only work but do it in a way that is useful for us. In other words do their job like they are supposed to. As devices and systems have grown extremely complex, it is not an easy task to make it happen. For example the IBM z13 microprocessor consists of 7.1 billion transistors (Warnock, J., 2015). The fact that technology evolves with every passing day makes things even worse because it is difficult to come up with new and better solutions with the same pace.

The increasing complexity of devices has resulted in emergence of the design methodologies at higher levels of abstraction such as Register-Transfer Level (RTL) and Electronic System Level (ESL).

This Thesis focuses on verification and design error correction at high abstraction levels in order to contend the challenges mentioned above. The underlying method applied is mutation analysis.

1.1 Verification of computing systems

With the growth of the complexity and extensive usage of computing systems the importance of verification has risen greatly. Nowadays electronics is applied everywhere – from pets to space technology. Failures in electronics range from merely being annoying to the loss of lives in extreme cases. Some examples of major accidents and incidents are described in the following paragraphs.

For example, from the middle of 1985 to January 1987, lack of verification led to the death of several people. It was caused by a computer-controlled radiation therapy machine, called the Therac-25, which massively overdosed at least six people with radiation. Some patients died and others received serious injuries. These accidents are known as the worst in the 35-year history of medical accelerators (Leveson N., 1995).

Widely known example is the bug in Intel Pentium processor that was discovered in May 1994. It became known as Pentium FDIV bug as the bug

appeared in floating-point division. At first Intel claimed the bug would affect only a few users but later studies had worse estimations. Public pressure led Intel to note that everyone who complains will get a replacement processor. The cost of replacement was $475 million (Nicely T., 2011).

On the morning of 4 June 1996 the flight of Ariane 5 launcher ended up with complete disintegration. About 40 seconds after the launch, the launcher veered off its flight path, broke up and exploded. The use of Ariane 4 software caused the accident due to the differences in early part trajectory of Ariane 5 compared to Ariane 4. This was not taken into account during the development of the software. More particularly, data conversion from 64-bit floating point to 16-bit signed integer value resulted in an operand error that ended up with the explosion of the launcher. Fortunately, no one was injured in the accident. Cost of the accident was at least $370 million (Lions J. L., 1996; Dowson M., 1997).

Sometimes it is very important to verify all corner cases. Although they might be unlikely to happen, the lack of verification might easily end up in loss of lives. Such thing happened in Panama between August 2000 and March 2001 with the loss of at least 17 people. The users of computerized treatment planning system who found an alternative configuration method to fulfil their needs caused this. As computer output gave the impression that the calculation results were correct, no one suspected anything. The result was that patients received a proportionately higher dose of radiation than prescribed. 28 patients in total were involved before mistreatment was stopped (Mettler F.A. Jr., Ortiz López P., et.al., 2001).

Toyota has made several recalls due to software bugs. In February 2010, Toyota called back 397,000 vehicles worldwide to fix an anti-lock brake software glitch and in 2005, Toyota repaired 75,000 Priuses to fix software glitches that caused the engine to stall (Manning S., Krisher T., 2010). Last recall was in February 2014 when 1.9 million third-generation Prius cars were recalled due to a programming glitch in their hybrid system. The setting of the software could cause higher thermal stress in certain transistors within the booster converter, resulting with deformation or damage to the transistors, which could end up with the hybrid system's shut down and the vehicle stopping suddenly (Kim C.-R., 2014).

Finally, there are bugs that have been present for a long time but no one has noticed them despite the fact that software is open-source. Good examples are Shellshock and Heartbleed, which were present for almost 20 years in millions of devices. It is something on a completely different scale. Compared to previous car industry examples, it is like understanding that tires are fundamentally flawed and all of them need a fix (CNN Money, 2014).

All these examples have one thing in common. They might have not happened if more advanced verification methods had been used.

In 1965 Gordon E. Moore wrote that the complexity for minimum component cost has increased at a rate of roughly a factor of two per year and there is no reason to believe it will not remain constant for at least 10 years. Moore revisited the subject ten years later and redrew his plot of component densities by a gentler slope, one in which density doubled every 18 months. Shortly after this, his plot was dubbed Moore's Law. A simple extrapolation from a simple observation has remained true throughout the decades (Schaller R. R., 1997).

The vast progress in the semiconductor industry has led to 10 nm technology node that raises gate density to over 6 million per square millimeter compared to roughly 190 transistors per mm^2 inside Intel 4004 microprocessor in 1971 (ITRS, 2013; Computer History Museum, 2015).

The complexity and cost of design and verification of integrated circuits have rapidly increased to the point where thousands of engineer-years (and a design team of hundreds) are devoted to a single design, yet processors reach market with hundreds of bugs. This aspect is leading to decreasing emphasis on heavy customization and exotic circuit topologies, and increasing use of design automation tools such as logic synthesis and automatic circuit tuning. The resulting productivity increases have allowed processor development schedules and team sizes to flatten out. Improvements in tools for analysis of timing, noise and power, and for verification of physical and electrical design rules, have also contributed to a steady increase in design quality (ITRS, 2013).

An important message in the ITRS roadmap (ITRS, 2013) is that design cost is the greatest threat to continuation of the semiconductor industry's phenomenal growth. According to Intel, a 1981 leading edge chip required 100 designer months, contained 10,000 transistors, which makes 100 transistors per month. A 2002 leading edge chip required already 30,000 designer months and it contains 150,000,000 making it 5000 transistors/month. However, the design costs have increased from $1M to $300M during the same period. Thus, the chip development capacity has increased 50 times, and design costs have increased 300 times at the same time. The same trend has continued over the recent years.

This dramatic increase in cost has mainly been due to the fact that traditionally the IC capacity has grown 58 %/ year, while the designer's productivity grows only 21 % annually. The phenomenon is shown in Figure 1.1, and it is known as the *design productivity gap* (Keutzer K., Newton R., 2015). It is the productivity gap that pushes the chip-making companies to exploit more and more engineering resources in order to reach the limits of what can be achieved in modern technology resulting in ever-increasing costs. Obviously, this gap could be contended and costs reduced only if more effective design approaches would be developed in the future to increase designer's productivity.

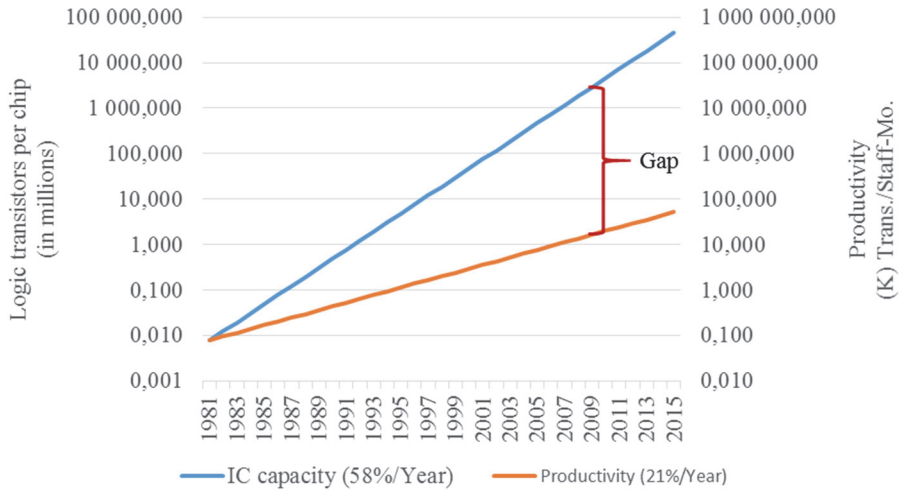


Figure 1.1. Design productivity gap

1.2 Error localization and correction

Designing a microelectronic chip is a very expensive task and excessive design costs are the greatest threat to the continuation of the semiconductor industry's growth. In order to contain this threat, the increasing gap between the complexity of new systems and the productivity of system design methods must be mitigated by developing new and more efficient design methods and tools. Functional correctness of systems is becoming ever more difficult to attain and it is becoming the main bottleneck in the systems' development process. Better verification techniques must be the focus in research and development if one wants to keep increasing the scale of electronics design. Detection of mistakes, however, offers only a partial solution to the correctness issue. Once that has been ascertained, the difficult task of discovering the sources of mistakes (faults) and subsequently locating and correcting them remains (Ubar R., Raik J., Vierhaus H. T., 2011).

It is a well-acknowledged fact that verification is forming a major part in the total product design cycle (Lam W. K., 2005), and this trend is increasing. At the same time when there have been numerous research works on verification methods identifying the occurrences of errors, the problem of diagnosing the causes of errors and correcting them has been largely neglected. Yet a large part of the verification cycle is consumed inside the design loops between debugging and correction. It is estimated that fault location and correction constitute roughly a half of the total time spent on verification and debug (FP6 PROSYD, 2004). Verification and debug (i.e. assuring the correctness of the design), in turn,

represent the main reason of the excessive costs accounting for about 70 % of design expenses (Lam W. K., 2005). Location and correction costs therefore form about 1/3 of the total design time. Figure 1.2 visualizes the amount of time spent on specification, design, fault detection, location, and correction in a typical design process (FP6 PROSYD, 2004; Ubar R., 2011).

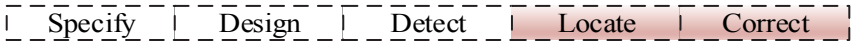


Figure 1.2. Time spent on different tasks in a design process

Every design must be verified throughout the whole design process in order to make sure that that the functionality matches the specification. In case too little effort is spent on verification, the results may be disastrous as the outcome might behave completely different from what was expected.

There are several methods for the task. Depending on the abstraction level they may additionally vary. The current Thesis focuses on using mutation analysis as one possible solution to the problem. Mutation analysis is addressed on two abstraction levels – Register-Transfer Level (RTL) and Electronic System Level (ESL).

In the traditional debug flow (Figure 1.3) a designer gets feedback from verification tools in form of counter-examples. On one hand, the designer is faced with too much information contained in the large counter-examples. On the other hand, there is not enough information in order to unambiguously locate the bug. As manual bug localization is very time-consuming it should be automated.

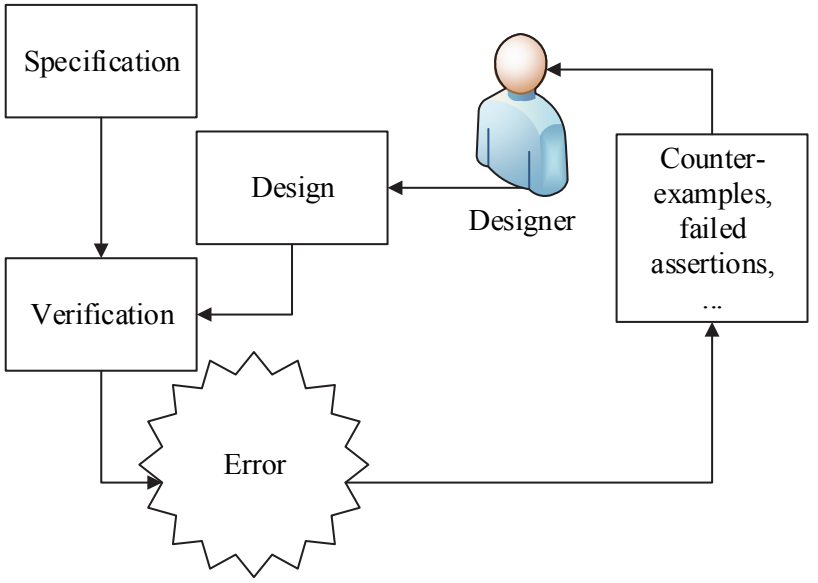


Figure 1.3. Traditional debug flow

Automation of the debug process consists of two steps. Once it is clear that the design and specification are not the same, or in other words, there is an error in the design, only one step is done. It must be followed by another one, which usually is error localization. After locating the error, the work is still not finished because generally it is reasonable to try and fix the error (Figure 1.4). Mutations can help here as every mutation can be viewed as a possible fix from the design view. The current Thesis addresses this topic and provides solutions on two abstraction levels – RTL and ESL.

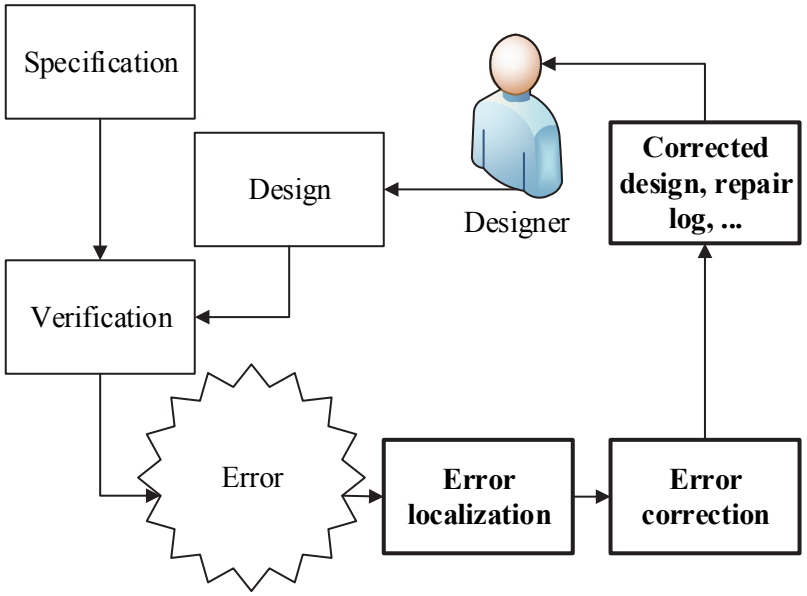


Figure 1.4. Automated debug flow

1.3 Main contributions

The main contribution of the Thesis is to propose new tools, case studies and methods to enable the designer automatically locate hard-to-detect bugs thereby offering solutions to save time and effort in integrated circuit design.

The contributions of this Thesis are:

- A new tool for mutation testing in hardware description languages using HLDDs.
- A new method to automatically inject faults into the functionality of system descriptions that performs mutation analysis at different abstraction levels.

- A case study of automatic localization of design errors (bugs) in processor designs.
- A method for statistical localization and mutation-based correction of design errors at the source-level of hardware description language code using HLDDs.
- A method for mutation-based correction of design errors in algorithmic descriptions of system-level hardware.

1.4 Outline of the Thesis

The presented Thesis is organized into four chapters. The introductory chapter is followed by background information with overview of functional verification, mutation analysis, design error correction, high-level decision diagrams and ESL modelling in SystemC.

The third chapter is divided into two main topics. In the first part mutation analysis is applied to high-level decision diagrams via an automated tool. The second part concentrates on mutation analysis at higher abstraction level with comparison of the two levels. Both methods are supported by experimental results.

The fourth chapter begins with design error correction at lower RTL level with thorough focus of backtrace, localization and correction. It is followed by an automated tool for design error correction at higher abstraction level.

Finally conclusions are presented.

2 Background

This Chapter provides the background for the topics that form the basis of the developments in the Thesis. The topics include functional verification, mutation analysis, error localization and correction, high-level decision diagrams and ESL modeling in SystemC.

2.1 Functional verification

A design process transforms a set of specifications into an implementation of the specifications. At the specification level, the specifications state the functionality that the design executes but do not indicate how it executes. An implementation of the specifications spells out the details of how the functionality is provided. Both a specification and an implementation are a form of description of functionality, but they have different levels of concreteness or abstraction. A description of a higher level of abstraction has fewer details; thus, a specification has a higher level of abstraction than an implementation. In an abstraction spectrum of design, a decreasing order of abstraction is seen: functional specification, algorithmic description, register-transfer level (RTL), gate netlist, transistor netlist, and layout (Figure 2.1). Along this spectrum a description at any level can give rise to many forms of a description at a lower level. For instance, an infinite number of circuits at the gate level implements the same RTL description. When moving down the ladder, a less abstract description adds more details while preserving the descriptions at higher levels. The process of turning a more abstract description into a more concrete description is called refinement. Therefore, a design process refines a set of specifications and produces various levels of concrete implementations (Lam W. K., 2005).

Design verification is the reverse process of design. Design verification starts with an implementation and confirms that the implementation meets its specifications. Thus, at every step of design, there is a corresponding verification step. For example, a design step that turns a functional specification into an algorithmic implementation requires a verification step to ensure that the algorithm performs the functionality in the specification. Similarly, a physical design that produces a layout from a gate netlist has to be verified to ensure that the layout corresponds to the gate netlist. In general, design verification encompasses many areas, such as functional verification, timing verification, layout verification, and electrical verification, just to name a few. In this Thesis only functional verification is considered and referred to as design verification.

Figure 2.2 shows the relationship between the design process and the verification process.

On a finer scope, design verification can be further classified into two types. The first type verifies that two versions of design are functionally equivalent. This type of verification is called equivalence checking. One common scenario of equivalence checking is comparing two versions of circuits at the same abstraction level. For instance, compare the gate netlist of a prescan circuit with its postscan version to ensure that the two are equivalent under normal operating mode (Lam W. K., 2005).

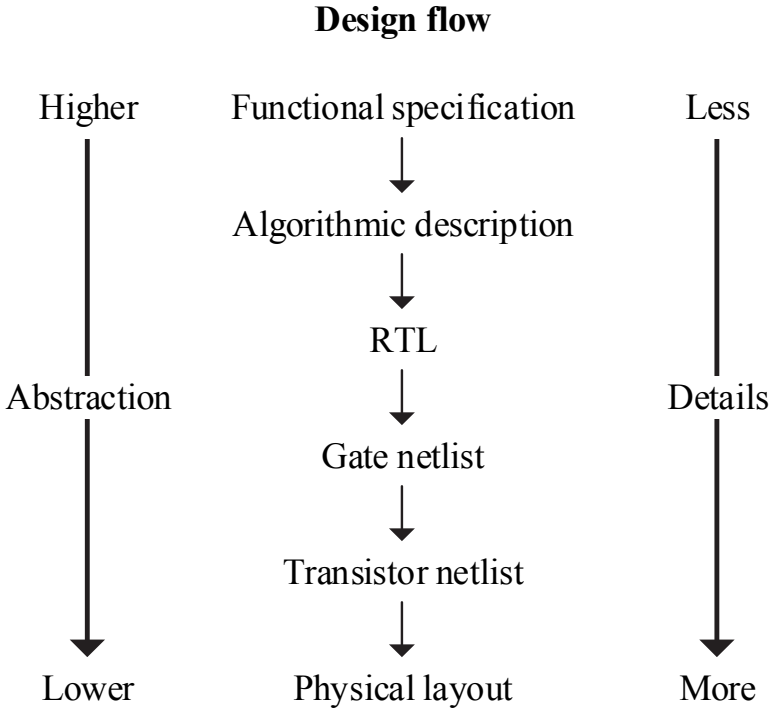


Figure 2.1. A ladder of design abstraction

However, the two versions of the design differ with regard to abstraction level. For example, one version of the design is at the level of specification and the other version is at the gate netlist level. When the two versions differ substantially with regard to the level of abstraction, they may not be functionally equivalent, because the lower level implementation may contain more details than allowed, but that are unspecified, at the higher level. For example, an implementation may contain timing constraints that are not part of the original specification. In this situation, instead of verifying the functional equivalence of the two versions, it is verified

whether the implementation satisfies the specifications. Note that equivalence checking is two-way verification, but this is a one-way verification because a specification may not satisfy an unspecified feature in the implementation. This type of verification is known as implementation verification, property checking, or model checking. Based on the terminology of property checking, the specifications are properties that the implementation must satisfy. Based on the terminology of model checking, the implementation or design is a model of the circuit and the specifications are properties. Hence, model checking means checking the model against the properties.

There are two types of design errors. The first type of error exists not in the specifications but in the implementations, and it is introduced during the implementation process. An example is human error in interpreting design functionality. To prevent this type of error, one can use a software program to synthesize an implementation directly from the specifications. Although this approach eliminates most human errors, errors can still result from bugs in the software program, or usage errors of the software program may be encountered. Furthermore, this synthesis approach is rather limited in practice for two reasons. First, many specifications are in the form of casual conversational language, such as English, as opposed to a form of precise mathematical language, such as Verilog or C++. It is known that automatic synthesis from a loose language is infeasible. Second, even if the specifications are written in a precise mathematical language, few synthesis software programs can produce implementations that meet all requirements. Usually, the software program synthesizes from a set of functional specifications but fails to meet timing requirements (Lam W. K., 2005).

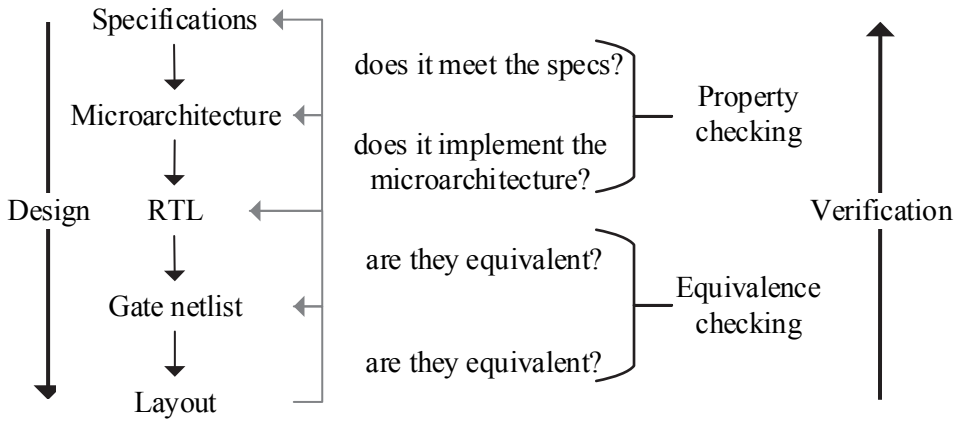


Figure 2.2. Relationship between the design and the verification processes

A more widely used method to uncover errors of this type is through redundancy. That is, the same specifications are implemented two or more times using different approaches, and the results of the approaches are compared. In theory, the more times and the more different ways the specifications are implemented, the higher the confidence produced by the verification. In practice, more than two approaches are rarely used, because more errors can be introduced in each alternative verification, and costs and time can be insurmountable.

The design process can be regarded as a path that transforms a set of specifications into an implementation. The basic principle behind verification consists of two steps. During the first step, there is a transformation from specifications to an implementation. Let us call this step verification transformation. During the second step, the result from the verification is compared with the result from the design to detect any errors. This is illustrated in Figure 2.3 (A). Oftentimes, the result from a verification transformation takes place in the head of a verification engineer, and takes the form of the properties deduced from the specifications. For instance, the expected result for a simulation input vector is calculated by a verification engineer based on the specifications and is an alternative implementation (Lam W. K., 2005).

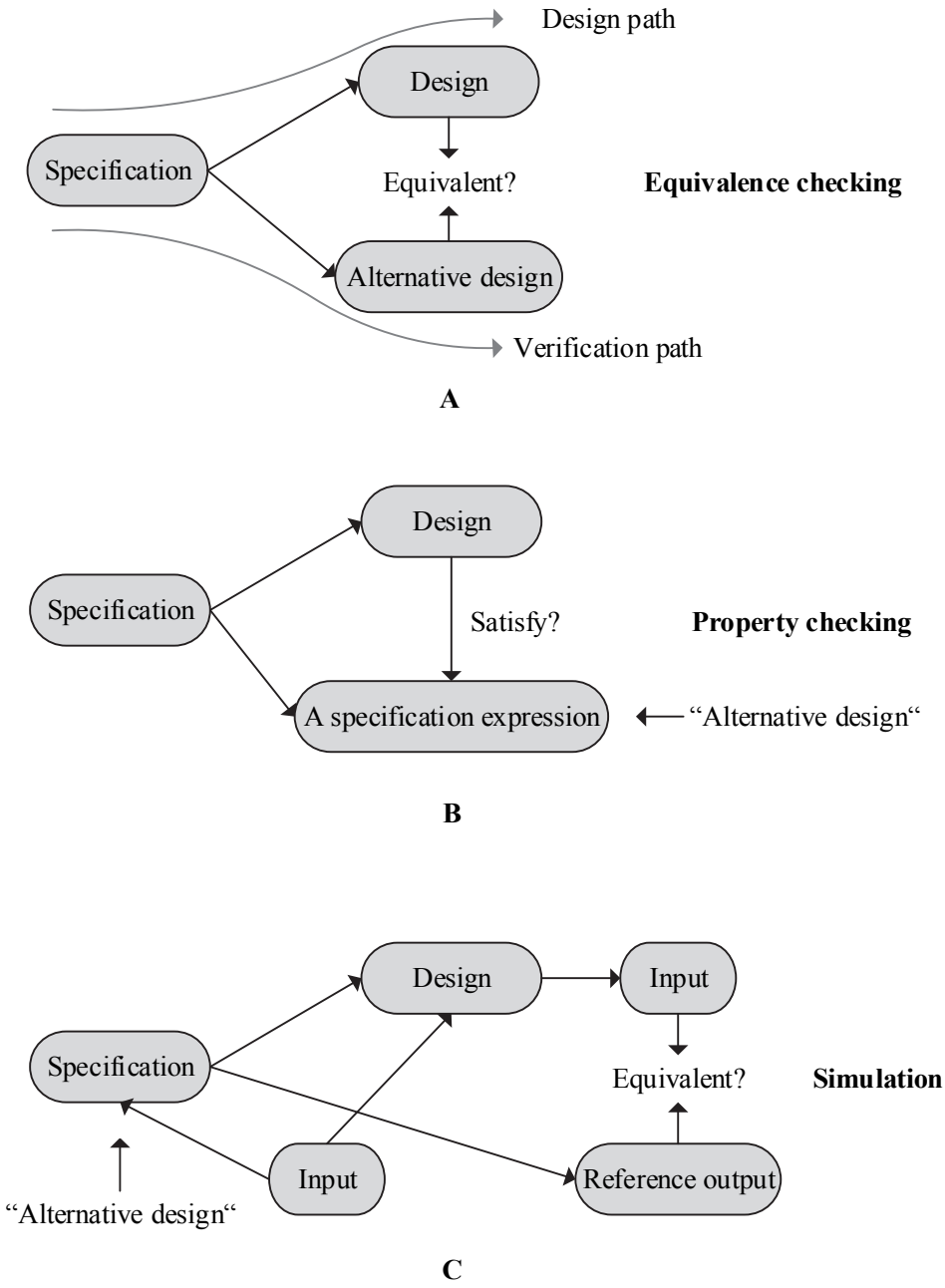


Figure 2.3. The basic principle of design verification

Obviously, if verification engineers go through the exact same procedures as the design engineers, both the design and verification engineers are likely to arrive at the same conclusions, avoiding and committing the same errors. Therefore, the more different the design and verification paths, the higher confidence the verification produces. One way to achieve high confidence is for verification engineers to transform specifications into an implementation model in a language different from the design language. This language is called verification language, as a counterpart to design language. Examples of verification languages include Vera, and C/C++. A possible verification strategy is to use C/C++ for the verification model and Verilog/VHSIC Hardware Description Language (VHDL) for the design model.

During the second step of verification, two forms of implementation are compared. This is achieved by expressing the two forms of implementation in a common intermediate form so that equivalency can be checked efficiently. Sometimes, a comparison mechanism can be sophisticated for example, comparing two networks with arrival packets that may be out of order. In this case, a common form is to sort the arrival packets in a predefined way. Another example of a comparison mechanism is determining the equivalence between a transistor-level circuit and an RTL implementation. A common intermediate form in this case is a binary decision diagram (Lam W. K., 2005).

Here it is seen that the classic simulation-based verification paradigm fits the verification principle. A simulation-based verification paradigm consists of four components: the circuit, test patterns, reference output, and a comparison mechanism. The circuit is simulated on the test patterns and the result is compared with the reference output. The implementation result from the design path is the circuit, and the implementation results from the verification path are the test patterns and the reference output. The reason for considering the test patterns and the reference output as implementation results from the verification path is that, during the process of determining the reference output from the test patterns, the verification engineer transforms the test patterns based on the specifications into the reference output, and this process is an implementation process. Finally, the comparison mechanism samples the simulation results and determines their equality with the reference output. The principle behind simulation-based verification is illustrated in Figure 2.3 (C).

Verification through redundancy is a double-edged sword. On the one hand, it uncovers inconsistencies between the two approaches. On the other hand, it can also introduce incompatible differences between the two approaches and often verification errors. For example, using a C/C++ model to verify against a Verilog design may force the verification engineer to resolve fundamental differences between the two languages that otherwise could be avoided. Because the two languages are different, there are areas where one language models accurately

whereas the other cannot. A case in point is modeling timing and parallelism in the C/C++ model, which is deficient. Because design codes are susceptible to errors, verification code is equally prone to errors. Therefore, verification engineers have to debug both design errors as well as verification errors. Thus, if used carelessly, redundancy strategy can end up making engineers debug more errors than those that exist in the design plus verification errors resulting in large verification overhead costs.

As discussed earlier, the first type of error is introduced during an implementation process. The second type of error exists in the specifications. It can be unspecified functionality, conflicting requirements, and unrealized features. The only way to detect the type of error is through redundancy, because specification is already at the top of the abstraction hierarchy and thus there is no reference model against which to check. Holding a design review meeting and having a team of engineers go over the design architecture is a form of verification through redundancy at work. Besides checking with redundancy directly, examining the requirements in the application environment in which the design will reside when it has become a product also detects bugs during specification, because the environment dictates how the design should behave and thus serves as a complementary form of design specification. Therefore, verifying the design requirements against the environment is another form of verification through redundancy. Furthermore, some of these types of errors will eventually be uncovered as the design takes a more concrete form. For example, at a later stage of implementation, conflicting requirements will surface as consistencies, and features will emerge as unrealizable, given the available technologies and affordable resources (Lam W. K., 2005).

2.2 Mutation analysis

Mutation analysis has a rich and varied history, with major advances in concepts, theory, technology, and social viewpoints. This history begins with (Lipton R., 1971) proposing initial concepts of mutation in a class term paper titled “Fault Diagnosis of Different Computer Programs." It was not until the end of the 1970's, however, before major work was published on the subject (DeMillo R. A., Lipton R. J., Sayward F. G., 1978) is generally cited as the seminal reference (Offut A. J., 2000).

Mutation analysis induces faults into software by creating many versions of the software, each containing one fault. Test cases are used to execute these faulty programs with the goal of distinguishing the faulty programs from the original program. Hence the terminology; faulty programs are mutants of the original, and

a mutant is killed by distinguishing the output of the mutant from that of the original program.

Mutants either represent likely faults, a mistake the programmer could have made, or they explicitly require a typical testing heuristic to be satisfied, such as *execute every branch* or *cause all expressions to become zero*. Mutants are limited to simple changes on the basis of the coupling effect, which says that complex faults are coupled to simple faults in such a way that a test data set that detects all simple faults in a program will detect most complex faults (Offut A. J., 2000).

Mutation analysis provides a test criterion, rather than a test process. A testing criterion is a rule or collection of rules that imposes requirements on a set of test cases. Test engineers measure the extent to which a criterion is satisfied in terms of coverage; a set of test cases achieves 100% coverage if it completely satisfies the criterion. Coverage is measured in terms of the requirements that are imposed; partial coverage is defined to be the per cent of requirements that are satisfied. Test requirements are specific things that must be satisfied or covered; for example, reaching statements are the requirements for statement coverage and killing mutants are the requirements for mutation. Thus, a test criterion establishes firm requirements for how much testing is necessary; a test process gives a sequence of steps to follow to generate test cases. There may be many processes used to satisfy a given criterion, and a test process need not have the goal of satisfying a criterion. In precise terms, mutation analysis is a way to measure the quality of the test cases and the actual testing of the software is a side effect. In practical terms however, the software is tested, and tested well, or the test cases do not kill mutants. This point can best be understood by examining a typical mutation analysis process.

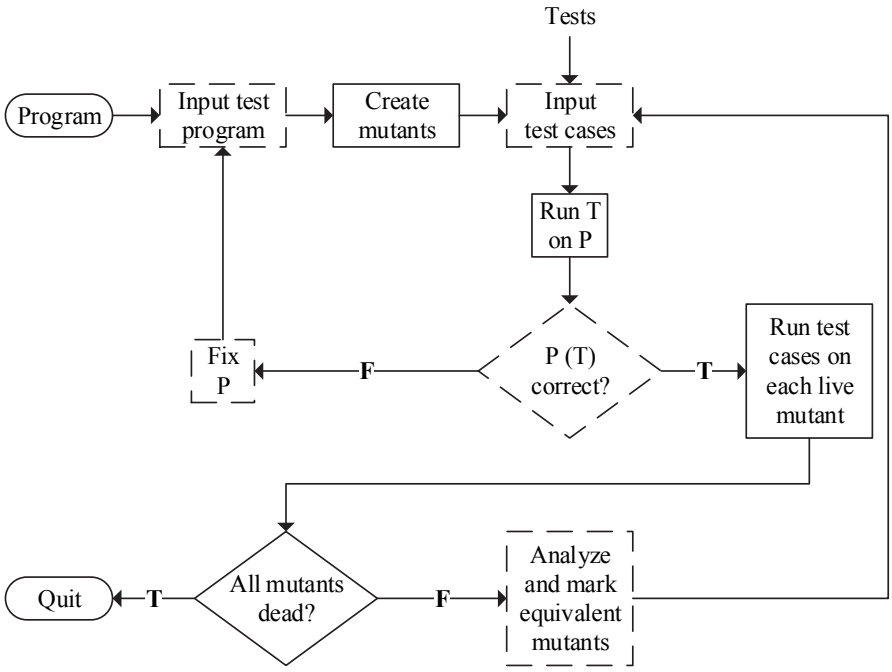


Figure 2.4. Traditional mutation process

When a program is submitted to a mutation system, the system first creates many mutated versions of the program. A mutation operator is a rule that is applied to a program to create mutants. Typical mutation operators, for example, replace each operand by every other syntactically legal operand, or modify expressions by replacing operators and inserting new operators, or delete entire statements. Figure 2.4 graphically shows a traditional mutation process. The solid boxes represent steps that are automated by traditional systems such as Mothra (DeMillo R. A., et.al., 1988), and the dashed boxes represent steps that are done manually (Offut A. J., 2000).

Next, test cases are supplied to the system to serve as inputs to the program. Each test case is executed on the original program and the tester verifies that the output is correct. If incorrect, a bug has been found and the program should be fixed before that test case is used again. If correct, the test cases are executed on each mutant program. If the output of a mutant program differs from the original (correct) output, the mutant is marked as being dead. Dead mutants are not executed against subsequent test cases.

Once all test cases have been executed, a mutation score is computed. The mutation score is the ratio of dead mutants over the total number of non-equivalent mutants. Thus, the tester's goal is to raise the mutation score to 1.00, indicating

that all mutants have been detected. A test set that kills all the mutants is said to be adequate relative to the mutants (Offut A. J., 2000).

2.3 Error localization and correction

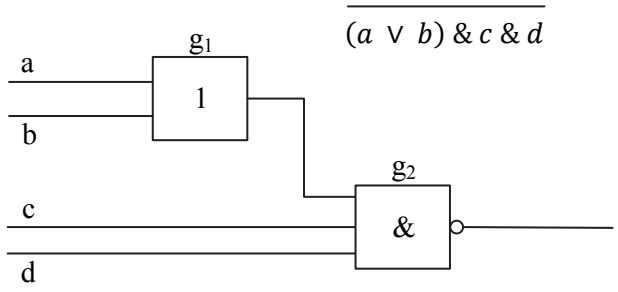
The dramatic increase in design complexity of modern electronics challenges our ability to ensure its functional correctness. While improvements in verification allow engineers to find a larger fraction of design errors more efficiently, little effort has been devoted to fixing such errors. As a result, debugging remains an expensive and challenging task. To address this problem, researchers have proposed techniques that automate the debugging process, by locating the error source within a design and/or by suggesting possible corrections (Chang K.-H., et al. 2007).

Design errors are mostly modeled in the implementation, however sometimes also in the specification. The main applications of design error localization and correction are: checking the synthesis tools, engineering changes (e.g. incremental synthesis) or debugging.

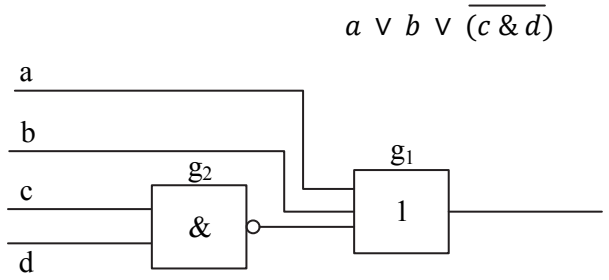
Design error localization and correction is applied when the design behavior does not match the expected behavior. Such mismatch may occur during simulation of the design, verification with formal tools (property/equivalence check) or when built-in checkers identify it.

The localization and correction methods can be classified into structure-based and specification-based ones. According to the fault model they can be divided into explicit (fault-model based) or implicit (fault-model free) methods. The advantage of explicit methods lies in the fact that they are easy to be formalized. However they are limited to enumerated bugs. On the one hand, the number of bugs to consider is very large; on the other hand, not all the possible bugs are included in the model. Further, the methods can be divided into single or multiple error assumption based, and simulation versus symbolic approaches.

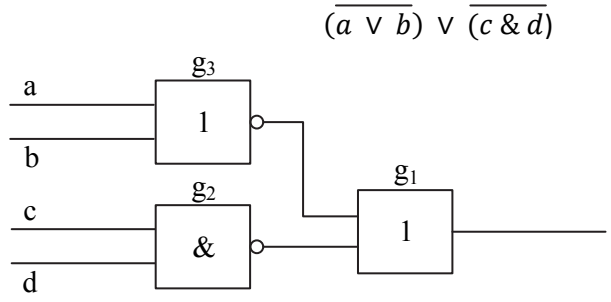
Since there is more than one way to synthesize a given function, it is possible that there is more than one way to model the error, and an incorrect implementation correction can be made at different locations. See example in Figure 2.5 (Jutman A., 1999).



a) Correct circuit



b) Wrong implementation



c) Possible correction

Figure 2.5. Ambiguity of error location

Design error diagnosis for combinational circuits has been thoroughly studied for two decades. There exist, both, fault model based (Madre J. C., Coudert O., Billon J. P., 1989; Abadir M. S., Ferguson J., Kirkland T. E., 1988) and fault-model-free (Ali M. F., et.al., 2005) approaches. There have been attempts to generalize the methods above for sequential circuits (Ali M. F., et.al., 2005; Wahba A., Borrione D., 1995), resulting in scalability problems. Some of the previous works support design error diagnosis for high-level models like the Register-Transfer Level (RTL) (Fey G., et.al., 2008; Chang K.-H., et.al., 2007).

However, these methods rely on reducing the diagnosis to logic-level formal engines. The current Thesis considers a different approach utilizing a source-level reasoning engine for the diagnosis process. This results in source-level feedback to the engineer and is therefore better understandable than logic-level debug information proposed by previous methods (Ubar R., Raik J., Vierhaus H. T., 2011).

2.4 High-Level Decision Diagrams

Different kinds of Decision Diagrams (DD) have been applied to design verification for about two decades. Reduced Ordered Binary Decision Diagrams (ROBDD) (Bryant R. E., 1986), as canonical forms of Boolean functions, have their application in equivalence checking and in symbolic model checking. In this Thesis, a decision diagram representation called High-Level Decision Diagrams (HLDDs) is used. HLDDs are word-level decision diagrams which can be considered as a generalization of BDD, where instead of single bits, computer words are considered. There exist a number of other word-level decision diagrams such as Multi-Terminal DDs (MTDDs) (Clarke E., et.al., 1993), Kronecker Multiplicative Binary Moment Diagrams (K*BMDs) (Drechsler R., Becker B., Ruppertz S., 1996) and Assignment Decision Diagrams (ADDs) (Chayakul V., Gajski D. D., Ramachandran L., 1993). However, in MTDDs the non-terminal vertices hold Boolean variables only, whereas in HLDDs the terminal vertices may be labeled by word-level variables. In K*BMDs, additive and multiplicative weights label the edges. Such representations are useful for compact canonical representation of functions on integers (especially wide integers). However, the main goal of HLDD representations described in this Thesis is not canonicity but the ease of simulation and diagnosis. The principal difference between HLDDs and ADDs lies in the fact that ADDs' edges are not labeled by activating values. In HLDDs the selection of a vertex activates a path through the diagram, which derives the needed value assignments for variables.

In this section the HLDD representation is defined, followed by an introduction of HLDD based simulation and a representation for behavioral register-transfer level VHDL descriptions.

Consider a digital system (Z, F) as a network of subsystems or components, where Z is the set of variables (Boolean, Boolean vectors or integers), which represent connections between components, primary inputs and primary outputs of the network. Let $Z = X \cup Y$, where X is the set of function arguments and Y is the set of function values where $Q = X \cap Y$ is the set of state variables. $D(z)$ denotes the finite set of all possible values for $z \in Z$ and $D(Z')$ is the set of all possible vectors in some variable set $Z' \subseteq Z$. Obviously, if $Z' = \{z_1, \ldots, z_n\}$ then $D(Z') =$

$D(z_1) \times \ldots \times D(z_n)$. Let F be the set of discrete functions: $y_k = f_k(X_k)$, where $y_k \in Y$, $f_k \in F$, and $X_k \subseteq X$ (k iterates over all elements in F).

Definition 1. High-level decision diagram representing a function $f_k : D(X_k) \rightarrow D(y_k)$ is a directed acyclic multigraph $G = (V, E)$ with a single root vertex and a set of terminal vertices where:

- V is the set of vertices and E is the set of edges.
- Each edge $e \in E$ is an ordered pair $e=(v_1, v_2) \in V^2$, where V^2 is the set of all the possible ordered pairs in the set V.
- Each non-terminal vertex is labeled by some input or control variable $x \in X$. Variable of vertex v by x_v shall be denoted.
- Each terminal vertex w is labeled by some function $g_w : D(X_w) \rightarrow D(y_k)$, where $X_w \subseteq X_k$.
- Each edge $e = (v, u)$, where v and u are vertices, is labeled by some constant $c_e \in D(x_v)$.
- Each two edges $e_1 = (v, u_1)$ and $e_2 = (v, u_2)$ starting from the same source vertex are labeled by different constants $c_{e1} \neq c_{e2}$.

If the vertex v is labeled by x_v then the number of edges starting from this vertex is $|D(x_v)|$.

Remark 1. Each BDD is HLDD as well, with two terminal vertices labeled by constant functions 0 and 1, and $D(x) = \{0, 1\}$ for every variable x.

In other words, HLDD is a data structure similar to BDD, but with many edges originating from a particular vertex, and with a number of functions at the end, instead of constants 0 and 1. One shall denote the set of terminal vertices by V^T and the set of non-terminal vertices by V^N and the set of all successors of the vertex v by $\Gamma(v)$. For non-terminal vertices $v \in V^N$ an onto function exists between the values $c \in D(x_v)$ of labels x_v and the successors $v^c \in \Gamma(v)$ of v. By v^c the successor of v for the value $x_v = c$ is denoted.

The edge (v, v^c), which connects vertices v and v^c, is called *activated* if there exists an assignment $x_v = c$. Activated edges, which connect v_i and v_j, form an *activated path* $l(v_i, v_j) \subseteq V$. An activated path $l(v_0, v^T)$ from the root vertex v_0 to a terminal vertex v^T is called the *full activated path* and v^T itself is referred to as the activated terminal vertex.

Without loss of generality it is assumed further that each variable has at least two values, i.e. $\forall z \in Z$, $|D(z)| > 1$. Let D_i designate a subset of $D(x_v)$ labeling vertex v, such that assignments from it will activate its successor vertex v_i. $D(x_v)$

is partitioned into non-intersecting sets D_1, …, D_m, where $m = |\Gamma(v)|$. More formally,

$$\bigcup_{i=1}^{m} D_i = D(x_v) \wedge \forall i, j, i \neq j \rightarrow D_i \cap D_j = \varnothing.$$

In other words, with every value assignment to variable x_v one and only one successor vertex will be activated. Further, let D_k designate a subset of $D(X)$, such that assignments from it will activate the terminal vertex v_k^T. With every value assignment to variables X, one and only one terminal vertex will be activated. Thus, $D(X)$ is partitioned into non-intersecting sets D_1, …, D_t, where $t = |V^T|$:

$$\bigcup_{k=1}^{t} D_k = D(X) \wedge \forall k, l, k \neq l \rightarrow D_k \cap D_l = \varnothing.$$

Figure 2.6 presents a HLDD G_y representing a discrete function $y=f(x_1,x_2,x_3,x_4)$. The diagram contains five vertices v_0, …, v_4. The root vertex v_0 is labeled by variable x_2, which is an integer with a range from 0 to 7. The vertex has three outgoing edges entering the vertices v_1, v_3 and v_4. The vertex v_1 is labeled by x_3 with a range from 0 to 3. It has two outgoing edges e_4 and e_5 entering terminal vertices v_2 and v_3, respectively. The edge e_4 is activated by x_3=2, while the edge e_5 is activated by x_3 having a value 0, 1 or 3. The ranges of variables x_1 and x_4 labeling terminal vertices v_3 and v_2, respectively, are not evident from the figure.

$G_y = (V, E, X)$;
$V = \{v_0, v_1, v_2, v_3, v_4\}$;
$E = \{e_1, e_2, e_3, e_4, e_5\}$, $e_1 = (v_0, v_1)$, $e_2 = (v_0, v_3)$,
$e_3 = (v_0, v_4)$, $e_4 = (v_1, v_2)$, $e_5 = (v_1, v_3)$;
$X = \{x_1 = x_{v3}, x_2 = x_{v0} = x_{v4}, x_3 = x_{v1}, x_4 = x_{v2}\}$;
$D_1(x_{v0}) = \{0\}$, $D_2(x_{v0}) = \{1, 2, 3\}$,
$D_3(x_{v0}) = \{4, 5, 6, 7\}$,
$D_1(x_{v1}) = \{2\}$, $D_2(x_{v1}) = \{0, 1, 3\}$.

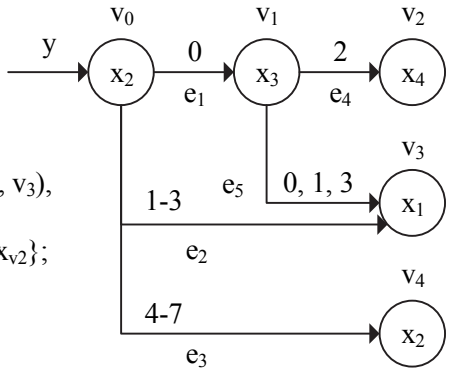


Figure 2.6. Graphical representation of a HLDD for a function $y=f(x_1,x_2,x_3,x_4)$

2.4.1 Simulation on HLDDs

HLDD models can be used for representing digital systems. In such models, the non-terminal vertices correspond to conditions or to control signals, and the terminal vertices represent arithmetic operations, variables or constants. When

representing systems by decision diagram models, in general case, a network of HLDDs rather than a single HLDD is required. During the simulation in HLDD systems, the values of some variables labeling the vertices of an HLDD are calculated by other HLDDs of the system.

Simulation on high-level decision diagrams takes place as follows. Consider a situation, where all the vertex variables are fixed to some value. According to these values, for each non-terminal vertex a certain output edge will be chosen to enter into its corresponding successor vertex. As mentioned above, such connections between vertices are referred to as the activated edges under the given values. Succeeding each other, activated edges form in turn activated paths. For each combination of values of all the vertex variables there always exists a corresponding activated path from the root vertex to some terminal vertex. Let us call this path the main activated path. The simulated value of the variable represented by the HLDD will be the value of the variable labeling the terminal vertex of the main activated path.

In Figure 2.7 simulation on the decision diagram presented in Figure 2.6 is shown. Assuming that variable x_2 is equal to 2, a path (marked by bold arrows) is activated from vertex v_0 (the root vertex) to a terminal vertex v_3 labeled by x_1. The value of variable x_1 is 4, thus, $y = x_1 = 4$. Note that this type of simulation is inherently event-driven since only those vertices have to be simulated (marked by grey color in Figure 2.7) that are traversed by the activated path.

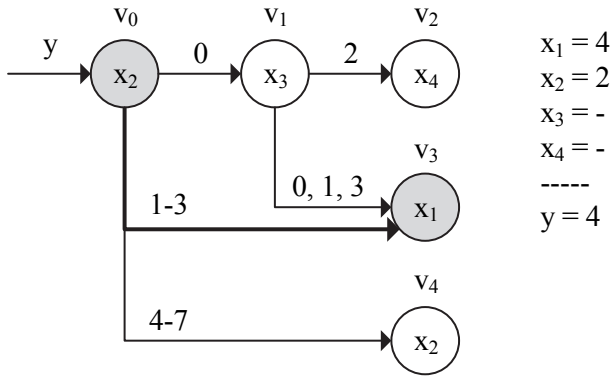


Figure 2.7. Simulation on a decision diagram

Figure 2.8 *Algorithm 1* presents simulation on HLDD models. The simulation process starts in the root vertex v_0 (line 2 of the algorithm). The vertex $v_{Current}$ is iteratively replaced by its successor vertices selected according to the value of $x_{vCurrent}$ (line 4). In order to represent feedback loops in the RTL design, the algorithm takes the previous time-step value of variable x_k labeling a vertex v_i if x_k represents a clocked variable in the corresponding HDL (lines 5, 6). Otherwise,

the present time step value will be used (line 8). $v_{Current}$ will be replaced by its successor vertex corresponding to $x_{v_{Current}} = Value$ (i.e. $v_{Current}^{Value}$) (line 10). Simulation ends when a terminal vertex is reached and the variable y corresponding to the simulated HLDD G_y is assigned the value $x_{vCurrent}$ (line 12).

1: *SimulateHLDD*(G_y)
2: $v_{Current} = v_0$
3: *While* $v_{Current} \notin V^T$
4: $x_k = x_{v_{Current}}$
5: *If* x_k *is clocked then*
6: *Value = previous time-step value of* x_k
7: *Else*
8: *Value = present time-step value of* x_k
9: *End if*
10: $v_{Current} = v_{Current}^{Value}$
11: *End while*
12: *Assign* $y = x_{v_{Current}}$
13: *End SimulateHLDD*

Figure 2.8 Algorithm 1. HLDD simulation

In the RTL style, the algorithm takes the previous time step value of variable x_k labeling a node $v_{Current}$ if x_k represents a clocked variable in the corresponding HDL. In the behavioral style, the present value of x_k will be used. In the case of behavioral HDL coding style HLDDs are generated and ranked in a specific order to ensure causality. For variables x_k labeling HLDD nodes the previous time step value is used if the HLDD calculating x_k is ranked after current decision diagram. Otherwise, the present time step value will be used.

2.4.2 Representing RTL designs by HLDDs

Consider the datapath depicted in Figure 2.9a and its corresponding HLDD representation shown in Figure 2.9b. Here, R_1 and R_2 are registers (R_2 is also a primary output), MUX_1, MUX_2 and MUX_3 are multiplexers, + and * denote addition and multiplication operations, *IN* is an input bus, SEL_1, SEL_2, SEL_3 and EN_2 serve as control signals (multiplexer selects and register enables), and *a, b, c, d* and *e* denote internal buses, respectively. In the HLDD, the control variables SEL_1, SEL_2, SEL_3 and EN_2 are labeling the internal decision vertices of the HLDD. The terminal vertices are labeled by word-level variables R_1 and R_2 (data transfers

to R_2), and by expressions related to the data manipulation operations of the network.

Consider, simulating HLDD with some values assigned to the variables. Let the value of SEL_2 be 0, the value of SEL_3 be 3 and the value of EN_2 be 1 in the current simulation run. A full activated path in the HLDD from EN_2 to R_1*R_2 is shown by bold lines and grey vertices, which corresponds to the pattern $EN_2=1$, $SEL_3=3$, and $SEL_2=0$. The activated part of the network at this pattern is denoted by grey boxes.

The main advantage and motivation of using HLDDs compared to the netlists of primitive functions is the increased efficiency of simulation and diagnostic modeling because of the direct and compact representation of cause-effect relationships. For example, instead of simulating the control word $SEL_1=0$, $SEL_2=0$, $SEL_3=3$, $EN_2 = 1$ by computing the functions $a = R_1$, $b = R_1$, $c = a + R_2$, $d = b * R_2$, e = d, and $R_2 = e$, one only needs to trace the vertices EN_2, SEL_3 and SEL_2 on the HLDD and compute a single operation $R_2 = R_1 * R_2$. In case of detecting an error in R_2 the possible causes can be defined immediately along the simulated path through EN_2, SEL_3 and SEL_2 without complex diagnostic analysis inside the corresponding RTL netlist. The activated path provides the *fault candidates*, i.e. variables that are suspected to contain faults causing the error at R_2 during current simulation run. Further reasoning should be based on analyzing sources of these signals.

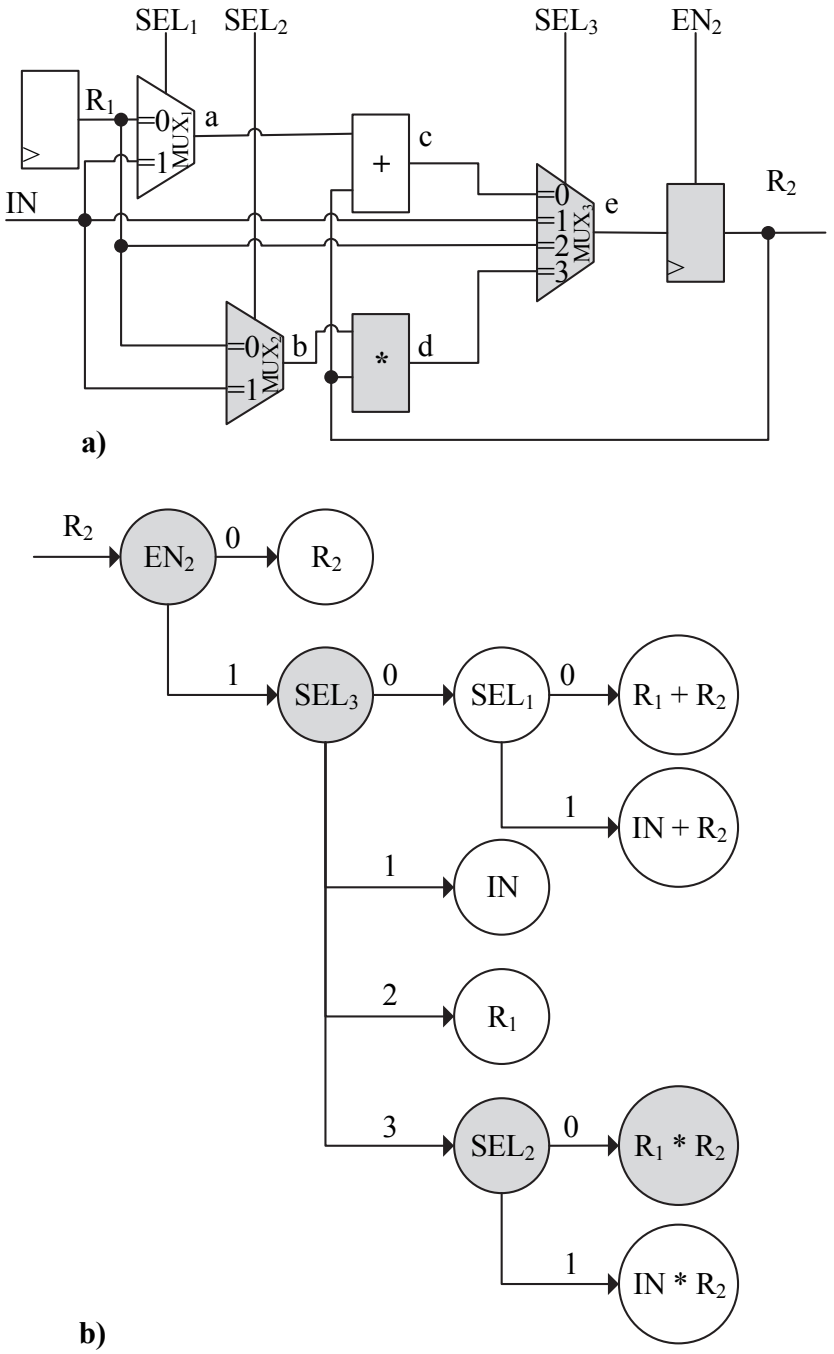
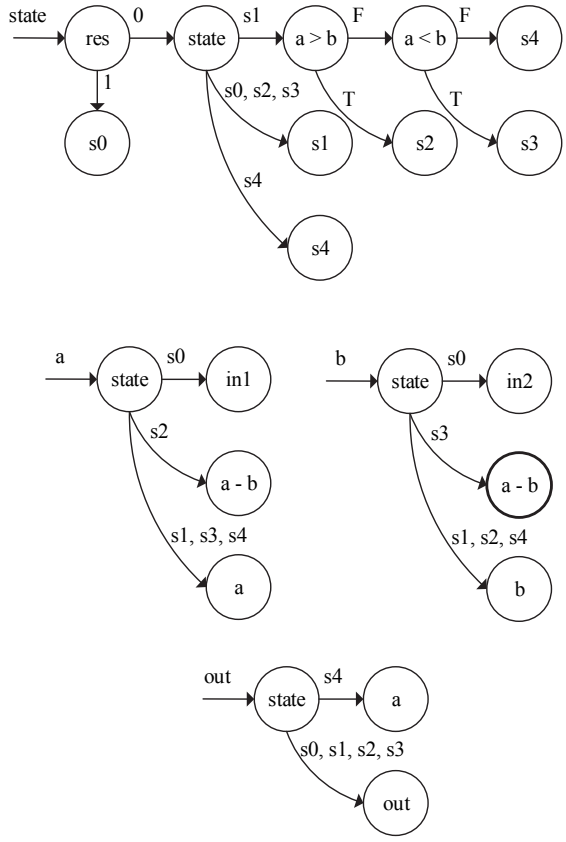


Figure 2.9. A datapath of a DUV

An example of HLDD representation (Figure 2.10b) of a VHDL code fragment of the Euclidean algorithm for calculating the Greatest Common Divisor of two unsigned variables *in1* and *in2* is presented in Figure 2.10a. The VHDL fragment contains seven variables: inputs *in1*, *in2* and *res* (the reset signal), internal variables (registers) *a, b* and *state* (for control state), and output *out*. The variable *state* is of enumeration type, variables *in1*, *in2*, *a, b* and *out* are integers and variable *res* is of bit type.

```
IF res = 1 THEN state := s0;
ELSE
  CASE state IS
  WHEN s0 =>
    a := in1;
    b := in2;
    state := s1;
  WHEN s1 =>
    IF a > b THEN state := s2;
    ELSE IF a < b THEN state := s3;
    ELSE state := s4;
    ENDIF;
  WHEN s2 =>
    a := a – b;
    state := s1;
  WHEN s3 =>
    b := a – b; -- Bug!!!
    state := s1;
  WHEN s4 =>
    out := a;
    state := s4;
  END CASE;
END IF;
```

a)



b)

Figure 2.10. RTL VHDL and its corresponding HLDD

The algorithm proceeds as follows. When the reset input *res* becomes one, the Finite State Machine (FSM) of the control part is initialized to the state *s0*. In that state, input *in1* is assigned to variable *a* and input *in2* is assigned to variable *b*. The next FSM state is *s1*, where if *a >b* one moves to state *s2*, if *a <b* one moves to state *s3*, and otherwise if *a=b* one moves to state *s4*, respectively. In state *s2*, *a-*

b is assigned to *a*, and in state *s3*, *b-a* is assigned to *b*. This guarantees that a smaller number is always subtracted from the larger one until *a* and *b* become equal and the FSM ends up in state *s4*, where the result is written to the output variable *out*.

Figure 2.10b presents the HLDD models of four variables state, a, b and out, i.e. the internal state and output variables of the design. HLDDs for design variables are generated by traversing the control flow branches of the VHDL code. Conditional statements (IF, CASE) transform into non-terminal vertices of the HLDD, control branches map to the HLDD edges and terminal vertices are created out of the right-hand side parts of value assignments to variables in corresponding control branches. In the figure, the symbols T and F labeling the HLDD edges stand for true and false, respectively.

Note that there is a bug in the VHDL description in Figure 2.10a. In the FSM state *s3*, *a-b* and not *b-a* is assigned to variable *b*. This bug will be used to illustrate the HLDD-based fault localization method explained in Section 4.1.4.

2.5 ESL modeling in SystemC

SystemC is the confluence of four streams of ideas: work at Synopsys with University of California, Irvine, and later with Infeon (formely Siemens HL) also; Frontier Design; IMEC; and work within the Open SystemC Initiative (OSCI) Language Working Group (References from System Design with SystemC), (Grötker T., et.al., 2002).

It is important to recognize that SystemC does not impose a top-down or bottom-up or even middle-out design flow. In fact, it is recognized that most design flows are iterative, and that it is rare that all modules within a system are modeled at the same level of abstraction. Commonly it is heard from designers in the industry that real designs hardly ever start with a "clean sheet of paper", so the need to model testbenches and preexisting hardware and software implementations at various levels of abstraction, is quite common.

Let's list a few simple design scenarios where different modeling levels might be used (Grötker T., et.al., 2002):

- A designer might use a very detailed implementation-level model for a design under test while using abstract models within the testbench to generate the design's stimulus and check the response.
- With a detailed implementation-level model as a starting point, a designer might create a more abstract model in order to increase

simulation speed and perhaps to protect intellectual property that might otherwise be exposed within the more detailed model.

- A designer might refine a module from a high-level functional specification down to a cycle-accurate RTL model while other modules in the system remain at higher levels of abstraction.

When considering a particular SystemC model and comparing it to an existing or proposed real-world implementation, one notes that there are several independent axes which can be used to gauge the model's accuracy. These include (Grötker T., et.al., 2002):

- *structural accuracy*: The extent to which the model reflects the structure of the actual implementations.
- *timing accuracy:* The extent to which the model reflects the timing of the actual implementation.
- *functional accuracy:* The extent to which the model reflects the function of the actual implementation.
- *data organization accuracy:* The extent to which the model reflects the actual data organization used within the implementation.
- *communication protocol accuracy:* The extent to which the model reflects actual communication protocols used within the target implementation.

For each of the different modeling accuracy aspects above, it sometimes also needs to be distinguished whether one is talking about the particular accuracy aspect only at a module's boundaries (i.e. at the module's ports), or whether the accuracy aspect also extends to all child modules contained within the parent module.

It must be noted that the modeling aspects listed above apply to software as well as hardware models. With software models it is important to identify the model accuracy in terms of structure, timing, function, data organization, and communication protocols (Grötker T., et.al., 2002).

Now that some of the important aspects which determine model accuracy have been identified, one can look into some of the terms that describe different modeling levels.

An *executable specification* is a model that is a direct translation of a design specification into SystemC. Executable specifications model the intended functionality of a design in a manner that is completely independent of any proposed implementation. If time delays are present in an executable

specification, they represent timing constraints to be imposed on the implementation.

An *untimed functional model* is similar to an executable specification, but no time delays at all are present in the model. Communication between modules within an untimed functional model is point-to-point (i.e., no shared communication links such as buses are modeled). Usually the communication is modeled using FIFOs with blocking write and read methods so that data items are reliably delivered between modules.

A *timed functional model* is similar to an untimed functional model in that communication between modules is still point-to-point (i.e., still no modeling of shared communication links) and in that it typically uses FIFOs with blocking read and write methods. However, in a timed functional model timing delays are added to processes within the design to reflect the timing constraints of the specification and processing delays of a particular target implementation.

Note that executable specifications and both untimed and timed functional models do not have any direct structural correspondence to a target implementation (Grötker T., et.al., 2002).

In a *transaction-level model* communication between modules is modeled using function calls. In such models, the communication is typically modeled in a way that is accurate in terms of functionality and often in terms of timing, but the communication is not modeled in a way that is structurally accurate.

When the term *platform transaction-level model* is used one is indicating that a model uses both the transaction-level modeling style and that the modules within such design structurally correspond to blocks within the target implementation.

A *behavioral hardware model* is a model that is pin-accurate and functionally accurate at its boundaries, but which is not considered to be clockcycle accurate at its boundaries.

The internal structure of an RTL model accurately reflects the registers and combinational logic of a target implementation.

Transaction-Level Modeling (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 for further refining and elaborating.

The Open SystemC Initiative (OSCI, 2009) 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 (OSCI, 2009).

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.

3 RTL and ESL mutation analysis methods

Mutation analysis is a known method in software domain. However, similarities with hardware and software design have brought the idea also to the hardware domain. This chapter introduces novel solutions using mutation analysis on Register-Transfer Level (RTL) with High-Level Decision Diagrams (HLDDs) and compares RTL and Electronic System Level (ESL) mutation analysis.

Subsection 3.1 starts with an overview of state-of-the-art mutation analysis at the RTL. Thereafter mutation analysis method is presented and implemented on RTL HLDDs. The method is followed by experimental results.

Subsection 3.2 describes state-of-the-art mutation analysis at the ESL, followed by the respective method on SystemC Transaction-Level Modeling (TLM) and experimental results.

3.1 RTL mutation analysis on HLDDs

The subsection presents a new tool for mutation analysis using the system model of HLDDs. The tool is integrated into the APRICOT verification environment. It is based on HLDD simulation and graph perturbation. A strategy that relies on a restricted set of five key mutation operators is developed in order to speed up the mutation analysis. Experiments on several ITC99 benchmarks and an industrial example show the feasibility of the mutation analysis approach.

This subsection is based on Paper I:

Hantson, Hanno; Raik, Jaan; di Guglielmo, Giuseppe; Jenihhin, Maksim; Chepurov, Anton; Fummi, Franco; Ubar, Raimund. “Mutation Analysis with High-Level Decision Diagrams”. Proceedings of the 11th Latin-American Test Workshop, IEEE Computer Society Press, 2010, pp. 1–6.

3.1.1 State-of-the-art

The observability problem of traditional coverage methods is widely analyzed in (Tao L., et.al., 2006). In particular the authors present an observability model and an algorithm to evaluate observability-based statement coverage for hardware designs. As in (Harris I. G., 2006), it is clearly stated that hardware designs are highly concurrent, while code software coverage metrics do not address this essential characteristic. Hence it is far from sufficient to achieve complete code coverage during verification (Tasiran S., Keutzer K., 2001).

Despite of being originally a software testing technique, obvious similarities with procedural programming languages suggested tailoring some software analysis techniques to Hardware Description Language (HDL) behavioral description analysis (Bolchini C., Baresi L., 1997). In particular, an adaptation of the mutation analysis to test VHDL functional descriptions is proposed in (Hayek G., Robach C., 1996). A VHDL language functional description can be assimilated to a software program, so it can be validated against (software) design faults using the mutation testing techniques. The methodology covers VHDL concurrent statements as block statement, process statement, and concurrent signal assignment statement. The VHDL code is translated into Fortran, and Mothra (DeMillo R. A., et.al, 1988) is applied to generate test sequences. In the proposed approach, however, concurrent constructs are merely translated to a sequential language and not targeted explicitly. In addition to academic attempts to bring mutation testing into hardware domain, a commercial functional qualification tool (Certitude, 2009) based on mutation analysis is available from Synopsys.

The adopted HLDD model provides fast simulation. Very efficient HLDD based simulation algorithms, which outperform commercial event-driven HDL simulators in 12 - 30 times and cycle-based simulators in 4 to 6 times, have been proposed (Ubar R., Morawiec A., Raik J., 2000). This is due to the fact that HLDD simulation essentially combines event-driven (path activation in the HLDD graphs) and cycle-based (HLDDs are synthesized into cycle-accurate models) paradigms.

This Thesis presents mutation analysis on the high-level decision diagram model. It is shown on an industrial example that high quality tests receiving near-hundred-percent code coverage result only in 21 % mutation coverage. This indicates a clear advantage of the mutation testing over the coverage approach, due to considering fault observation.

3.1.2 Mutation analysis method

The method presented in this Thesis is based on strong mutation. The five *key* operators proposed in (Offutt A. J., Rothermel G., Zapf C., 1993) have been implemented according to the *do fewer* strategy. In experiments, those five operators 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 (Offutt A. J., Rothermel G., Zapf C., 1993). The 5 sufficient operators are ABS, which forces each arithmetic expression to take on the value 0, a positive value, and a negative value, AOR, which replaces each arithmetic operator with every syntactically legal operator, LCR, which replaces each logical connector with several kinds of logical connectors, ROR, which replaces

relational operators with other relational operators, and UOI, which inserts unary operators in front of expressions.

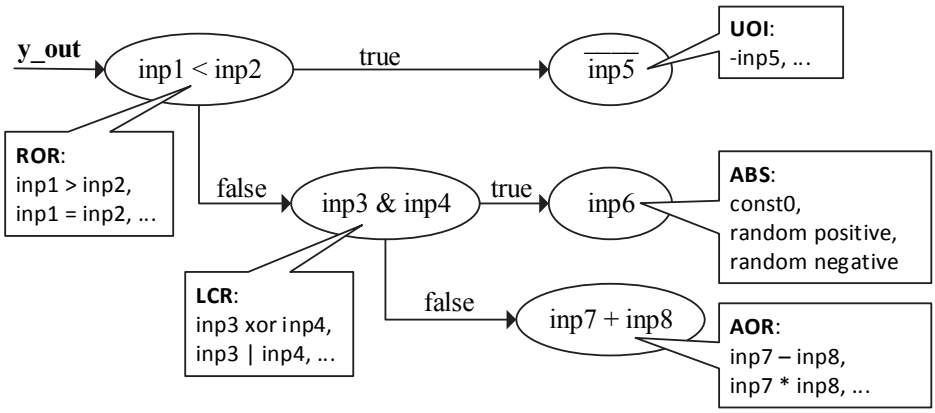


Figure 3.1. "Key" mutation operators as HLDD perturbations

The five operators have been implemented with the following constraints and specifics. UOI currently replaces only unary operators with other unary operators and ABS is applied to variables only, and not to expressions. Note also that in HLDD there are no signed/unsigned variables, but signed and unsigned relational operators exist. Therefore ROR replaces, both, signed and unsigned relational operators. In AOR mutations are also allowed by division and mod operations and a check for the case of divide-by-zero has been included. In the future, the goal is to gradually extend the set of mutation operators and select the most optimal set for hardware programs. The reduced-5-key-operator strategy represents a do fewer strategy. The purpose would be to reduce the mutation analysis cost as much as possible.

Figure 3.1 illustrates the HLDD graph perturbations for implementing the five key mutation operators on a sample diagram Gy_out. In HLDD models, the perturbation means simply replacement of an operator, variable or constant labeling the HLDD node by another operator, variable or constant.

Table 3.1 shows the list of replacements for each mutation operator. In every case the operator is substituted by another operator from the group. This is done until all operators are covered.

Table 3.1. Mutation operators

Mutation operator	List of replacements
LCR (logical connector replacement)	AND, NAND, OR, NOR, XOR
AOR (arithmetic operator replacement)	ADDER, SUBTR, MULT, DIV, MOD
UOI (unary operator insertion)	NEG, INV
SOR (shift operator replacement)	SHIFT_LEFT, SHIFT_RIGHT, U_SHIFT_LEFT
ROR (relational operator replacement)	EQ, NEQ, GT, LT, GE, LE, U_GT, U_LT, U_GE, U_LE

Figure 3.2 *Algorithm 2* presents the Mutation Analysis (MA) algorithm on HLDD representations. The MA process starts with HLDD simulation in order to find the correct output responses to be saved at this point. A mutated operator is injected to the node m and simulated. As the final step the simulated output responses are compared to the correct ones to determine whether the mutant has been killed or not.

```
1:    HLDD_MA()
2:      SimulateHLDD() /* Figure 2.8 Algorithm 1*/
3:      Save output responses
4:      For each node m
5:        For each mutated operation p where x_m= Z(m) ≠ p
6:          Replace x_m by p
7:          SimulateHLDD() /* Figure 2.8 Algorithm 1 */
8:          If output responses differ from the saved ones then
9:            Report mutant killed
10:         End if
11:       End for
12:     End for
13:   End HLDD_MA
```

Figure 3.2 Algorithm 2. HLDD-based mutation analysis

3.1.3 Experimental results

In the following there are mutation analysis experiments with the (ITC99, 2009) circuits, which were introduced in order to measure the quality of test generation in hardware systems and with an industrial design implementing a cyclic redundancy check (CRC) from the FP6 VERTIGO project (Vertigo, 2009).

Basic quantitative VHDL characteristics of the ITC99 benchmarks and the CRC design are listed below in Table 3.2. In the Table, the number of VHDL code lines, primary input signals, primary output signals, variables/signals corresponding to registers and the number of VHDL processes are reported, respectively.

Table 3.2. VHDL code characteristics

Design	Code lines #	Inputs #	Outputs #	Registers #	Processes #
b01	96	4	2	3	1
b02	61	3	1	2	1
b04	76	6	1	9	1
b06	112	4	4	5	1
b09	81	3	1	5	1
b11	107	4	1	5	1
b13	273	5	7	24	5
CRC	371	10	3	11	9

Table 3.3 presents the mutation analysis experiments on the full-HLDD versions of the ITC99 benchmarks. The row '# Vectors' shows the number of stimuli in the test bench. All the test benches provide 100 % statement coverage, except for b11 (97 %) and b13 (96.1%), where creation of full tests was not achieved. All the test sets were generated manually.

Table 3.3. Mutation analysis experiments

	b01	b02	b04	b06	b09	b11	b13
# Vectors	14	10	8	11	23	88	11
# Mutants inserted	154	78	233	336	213	375	972
# Mutants killed	49	9	18	39	17	178	77
Mutation coverage	0.32	0.12	0.08	0.12	0.08	0.47	0.08
Time, s	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	0.22	0.21

The next row shows the number of mutants injected. The row "# Mutants killed" presents the total number of mutants killed. The row "Mutation coverage" shows the ratio of killed mutants to the number of mutants injected in the approach. One of the most interesting observations is the very low mutation coverage: only 8 per cent for b04, b09 and b13. The explanation lies in rather short test sets. Nevertheless, this gives an idea how small observation coverage is guaranteed by 100 % code coverage tests in the worst case.

The last row shows the execution times of the mutation analysis, which were in the range of tenths of seconds. All the experiments were run on a 1.7 GHz laptop PC.

Table 3.4 lists the results of mutation analysis experiments with the previously described ITC99 benchmarks using longer tests, covering also branches. In most cases mutation coverage has increased, but it still remains low, which clearly states the need for better test sets. The enormous rise of processing time with b13 can be explained by the fact that test length was increased 100 times.

Table 3.4. Mutation analysis experiments 2

	b01	**b02**	**b04**	**b06**	**b09**	**b11**	**b13**
# Vectors	23	14	11	52	33	132	1148
# Mutants inserted	154	78	233	336	213	375	972
# Mutants killed	57	9	32	50	35	198	281
Mutation coverage	0.37	0.12	0.14	0.15	0.16	0.53	0.29
Time, s	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	0.34	15.36

Results of the mutation analysis experiments on the CRC example are presented in Table 3.5. The rows in this table have similar semantics to the ones in Table 3.4. It can be seen that the HLDD-based mutation analysis time is in the range of seconds. Again, the mutation coverage is very low (only 21 per cent) compared to the code coverage. While partly explained by the short test set it confirms the weak observation coverage guaranteed by code coverage tests and motivates the use of mutation analysis.

Table 3.5. CRC example

	CRC
# Vectors	42
# Mutants inserted	1247
# Mutants killed	268
Mutation coverage	0.21
Time, s	3.73

3.2 ESL mutation analysis on System C TLM

Mutation analysis has been borrowed from the software-testing domain as a technique for evaluating the quality of testbenches in validating digital systems. This section 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 section, 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. The section presents a novel method for mutation analysis directly working on uncompiled SystemC TLM code.

This subsection is based on Paper II:

Guarnieri, Valerio; Di Guglielmo, Giuseppe; Bombieri, Nicola; Pravadelli, Graziano; Fummi, Franco; Hantson, Hanno; Raik, Jaan; Jenihhin, Maksim; Ubar, Raimund. “On the Reuse of TLM Mutation Analysis at RTL”. Journal of Electronic Testing-Theory and Applications, 28(4), 2012, pp. 435–448.

3.2.1 State-of-the-art

The initial concept of mutation analysis was first proposed by Richard Lipton (R. Lipton, 1971). However, major work was not published until the end of 1970s (Budd T.A., Sayward F.G., 1977), (DeMillo R. A., Lipton R. J., Sayward F. G., 1978), (Hamlet R. G., 1977).

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 (Choi B.J., et.al., 1989), (Offut, A. J., King, K. N., 1987), 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, (Agrawal H., et.al, 1989) 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 A. J., Rothermel G., Zapf C., 1993) 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 presented in this Thesis is based on these key operator categories.

Mutation analysis has been applied also to Java (Irvine S. A., et.al. 2007) and SQL (Ma Y. S., Offut A. J., Kwon Y. R., 2005), (Tuya J., Suarez-Cabal M. J., De La Riva C., 2006). Several approaches (Alexander R. T., et.al., 2002), (Belli F., Budnik C.-J., Wong W.-E., 2006), empirical studies (Lyu M.-R., et.al., 2003) and frameworks (Bradbury J. S., Cordy J. R., Dingel J., 2006) have been presented in the literature for mutation analysis of such languages.

(Hantson H., et al. 2010) 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 (Bombieri N., Fummi, F., Pravadelli G., 2008; Bombieri N., Fummi, F., Pravadelli G., 2009; Bombieri N., et.al., 2009; Lisherness P., Cheng K.-T. (Tim), 2010; Sen A., 2009), (Sen A., Abadir M. S., 2010). Mutation models for perturbing SystemC TLM descriptions are proposed in (Bombieri N., Fummi, F., Pravadelli G., 2008; Bombieri N., Fummi, F., Pravadelli G., 2009; Sen A., 2009). 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 A., 2009) 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., 2009) 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 A., Abadir M. S., 2010) 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 P., Cheng K.-T. (Tim), 2010) presents SCEMIT, a tool for the automated injection of errors into C/C++/ SystemC models. A selection of mutation style errors is 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 (Bombieri N., Fummi F., Pravadelli G., 2006). In this work it is shown 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 is done in this Thesis.

The novelty of the approach presented in this Thesis lies in the fact that it faces the reuse of mutation analysis through the different refinement steps of a TLM-based design flow as done in the following sections. This Thesis extends the work presented in (Guarnieri V., Hantson H., et. Al., 2011) 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.

3.2.2 Mutation analysis method

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 method was not to propose new operators or operator categories. Therefore, a slightly modified set of five key operator categories, proposed in (Hantson H., et al. 2010), was used. In the 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 (Offutt A. J., Rothermel G., Zapf C., 1993). The categories of operators used in the current method are the following: arithmetic operator replacement (AOR), logical connector replacement (LCR), shift operator replacement (SOR), relational operator replacement (ROR) and unary operator injection (UOI).

Table 3.6 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 3.6. Categories of mutation operators

Mutation operator	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)	Shift left (SL), Shift right (SR)
ROR (relational operator replacement)	Equal (EQ), not equal (NEQ), greater than (GT), less than (LT), greater than or equal (GE), less than or equal (LE)
UOI (unary operator 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.

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. Currently the testbench-based method is used 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, a system description is generated 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 3.3 illustrates the whole injection process.

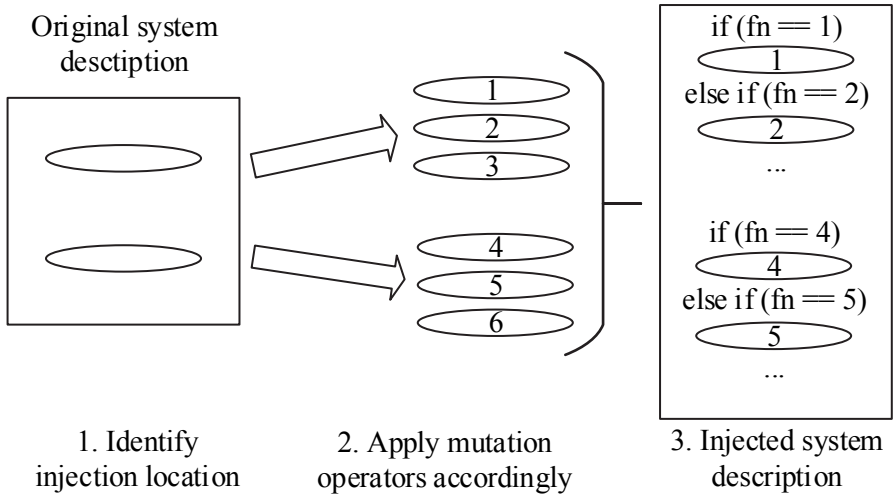


Figure 3.3. Mutant injection overview

3.2.3 Experimental results

In order to validate the efficiency, in terms of speed and coverage differences of the method at different abstraction levels, mutation analysis on a number of designs was performed, 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, 2010) (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 3.7.

Table 3.7. Experimental results

Method	**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%	98%	90%	35%	98%	90%	41%	100%	94%
# 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 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 3.4 shows. This again confirmed the 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 3.4 and Figure 3.6 represent simulation times and mutation coverage, respectively.

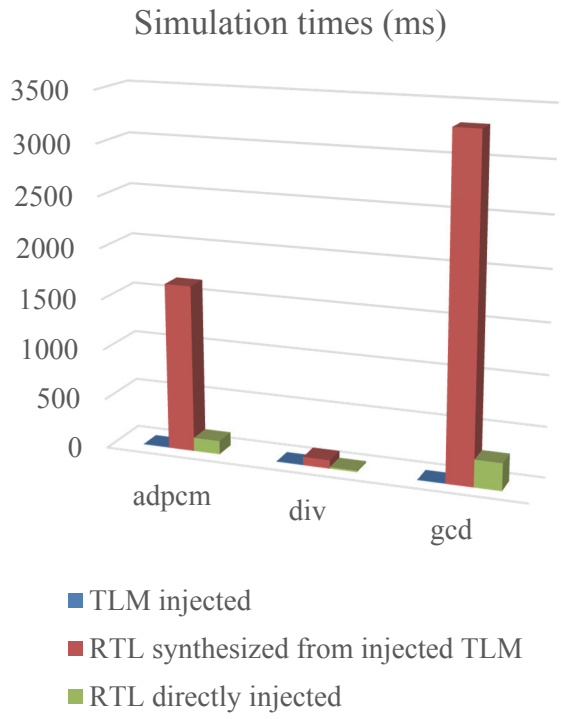


Figure 3.4. Simulation times

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 3.5 and Figure 3.7 respectively.

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

Figure 3.5. SystemC code example at TLM

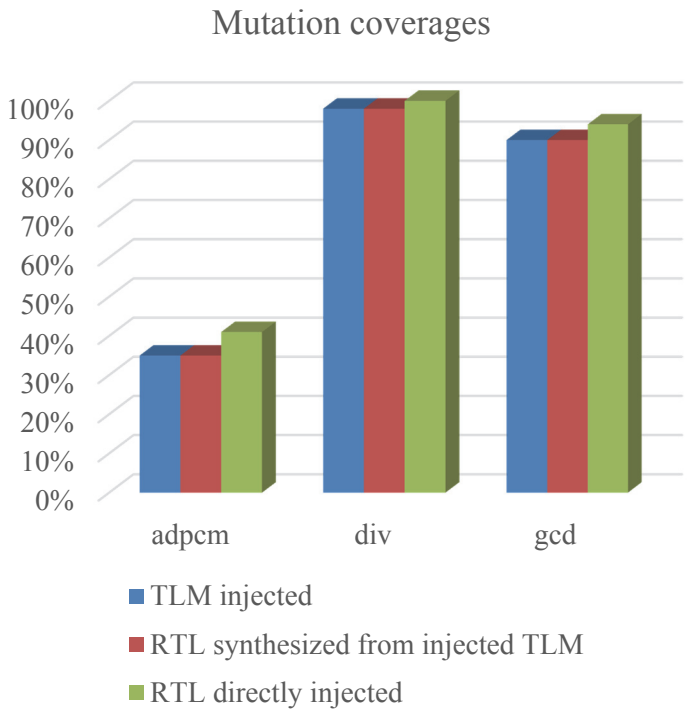


Figure 3.6. Mutation coverages

The difference between the two pieces of source code should be striking immediately. 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 or to be manually edited by human beings.

It was somewhat surprising 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 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 3.7. SystemC code example at generated RTL

It is worth noting that these experiments and the subsequent analysis led 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 a bug in the design description was also discovered when investigating the reasons for low mutation coverage. Thus, it can definitely be claimed that mutation analysis allowed to evaluate the quality of the verification environment and to verify the correctness of a design through simulation.

3.3 Conclusions

The section presented a new tool for mutation testing in hardware description languages using the system model of high-level decision diagrams (HLDD). The tool is integrated into the APRICOT verification environment. It is based on HLDD simulation and graph perturbation. A strategy that relies on a restricted set of five key mutation operators is developed in order to speed up the mutation analysis.

Experiments on several ITC99 benchmarks and an industrial example prove the feasibility of the approach. The tests showed that the mutation coverage was always very low compared to the code coverage. While partly explained by the short test sets applied it confirms the weak observation capabilities guaranteed by code coverage tests and motivates the use of mutation analysis.

A method to automatically inject faults into the functionality of system descriptions that works at different abstraction levels (TLM and behavioral RTL) was presented. The novelty of the method lies in 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.

4 RTL and ESL error correction methods

Verification is increasingly becoming the bottleneck in designing digital systems. In fact, most of the verification cycle is not spent on detecting the occurrences of errors but on debugging, consisting of locating and correcting the errors. However, automated design-error debug, especially at the system-level, has received far less attention than error detection.

This chapter presents design error localization and correction on High-Level Decision Diagram (HLDD) Register-Transfer Level (RTL) followed by a case study of existing ROBSY processor and finally design error correction for simple C at Electronic System Level (ESL).

Subsection 4.1 presents design error localization and correction on HLDDs at the RTL. Subsection 4.2 follows with a case study of and industrial microprocessor ROBSY. In subsection 4.3 a method for design error correction in C programs is presented.

4.1 Design error localization and correction on HLDDs at the RTL

The subsection proposes a method for locating design errors at the source-level of RTL hardware description language code using the design representation of HLDD models and correcting them by applying mutation operators. The error localization is based on backtracing the mismatched and matched outputs of the design under verification on HLDDs. As a result of the localization step, all the variables in the RTL description receive a suspiciousness score.

Subsequently, a mutation-based correction algorithm is applied providing automated correction for the design under verification. Experiments on a set of sequential RTL benchmarks show that the method is capable of locating the design errors injected with a high accuracy, and a short run time. In fact a majority of the errors injected in the experiments were identified as top suspects by the current diagnosis algorithm. Furthermore, it is shown that because of this localization accuracy the mutation-based correction requires very small number of iterations and thus a short run-time.

This subsection is based on Paper III:

Raik, Jaan; Repinski, Urmas; Tšepurov, Anton; Hantson, Hanno; Ubar, Raimund; Jenihhin, Maksim. “Automated design error debug using high-level decision diagrams and mutation operators”. Microprocessors and Microsystems: Embedded Hardware Design, 37(4), 2013, pp. 1–10.

4.1.1 State-of-the-art

Automated debug of design errors consists of two steps: error localization and error correction. Error localization identifies the portion of the design responsible for the erroneous behavior, while error correction is responsible for locally modifying the functionality of the identified portion.

For error localization, simulation-based (Ali M, et. Al., 2005), (Wahba A., Borrione D., 1995), (Smith A., Veneris A., Viglas A., 2004), (Fey G., et.al, 2008), (Chang K.-H., 2007), (Debroy V., Wong W. E., 2010) and formal approaches (Könighofer R., Bloem R., 2011) are known. It is widely accepted that simulation-based techniques scale well with the design size, but are not exhaustive while formal techniques provide a high grade of confidence in the results but are susceptible to the design complexity.

For error correction, error matching (Madre J. C., Coudert O., Billon J. P., 1989), (Abadir M. S., Ferguson J., Kirkland T. E., 1988) and re-synthesis (Ali M. F., et. Al., 2005) have been investigated in the literature. In particular, re-synthesis provides a correction that is represented as a partial truth table based on the stimuli under consideration. This kind of correction is not readable and cannot be easily understood and verified by the design engineer. Moreover, the resynthesized erroneous portion of the design is likely to fail when new stimuli will be added to the suite.

Previous works on error debug for high-level models, such as the Register-Transfer Level (RTL), are based on the work by (Smith A., Veneris A., Viglas A., 2004). There is a range of works extending this idea of the SAT-based debug e.g. (Fey G., et. Al., 2008, Chang K.-H., 2007). However, these methods reduce the debugging problem to SAT or SAT Modulo Theory (SMT) solvers, which is an NP-complete problem. Although SAT/SMT engines are being constantly developed and improved, there is a limit to the circuit size where the approach is applicable. The current Thesis considers a different approach relying on design error localization utilizing HLDD backtrace that executes in polynomial time. This means that much larger designs could be potentially handled by the method.

This Thesis utilizes HLDD backtrace and mutation as a source-level reasoning engine for automated debug. The engine operates directly on the register-transfer level. This results in a readable diagnostic feedback and is therefore better understandable to the engineer than logic-level debug information provided by previous methods.

Recently, a similar approach has been adopted in software testing. In (Debroy V., Wong W. E., 2010), Debroy and Wong propose a program slicing based diagnosis tool Tarantula to calculate the suspiciousness scores for operations and apply mutation to correct C and Java programs. The current approach for hardware

debug and the one proposed in (Debroy V., Wong W. E., 2010) for software debug were developed simultaneously and are independent of each other.

4.1.2 Backtrace

This section presents the algorithm for diagnostic tree generation using backtrace on HLDD models. Followed by two analysis steps to perform error localization on the set of diagnostic trees generated.

Firuge 4.1 *Algorithm 3* presents the recursive diagnostic tree generation on HLDDs. The process starts from the primary outputs (Line 2) and from each clock-cycle (Line 3). Subsequently, the diagnostic tree is recursively generated using the function *RecursiveTreeGeneration*.

```
1:      GenerateDiagnosticTree()
2:          For each primary output G_O in the model
3:              For each time-step t
4:                  δ(G_O, t) = ∅
5:                  RecursiveTreeGeneration(G_O, t, δ)
6:              End for
7:          End for
8:      End GenerateDiagnosticTree
9:
10:     RecursiveTreeGeneration(G_y, t, δ)
11:         SimulateHLDD(Gy) /* Figure 2.8 Algorithm 1 */
12:         For each v_i at the main activated path
13:             If variable x_k = x_vi at-time step t is not in δ then
14:                 Add x_k to δ
15:                 If x_k is not a primary input then
16:                     RecursiveTreeGeneration(Gx_k, t, δ)
17:                 End if
18:             End if
19:         End for
20:     End RecursiveTreeGeneration
```

Figure 4.1 Algorithm 3. HLDD-based diagnostic tree generation

Figure 4.1 *Algorithm 3* generates a separate diagnostic tree $\delta(G_O, t)$ for each output diagram G_O at each clock-cycle t. The resulting diagnostic tree δ is a set of pairs (x_i, t_j) that show at which time-steps t_j the variable x_i was backtraced.

4.1.3 Localization

In the following, two analysis steps that could be implemented for locating the design error are presented. In order to perform the analysis, let us partition the set of all diagnostic trees $\Delta = \delta_k(G_O, t)$ into failing diagnostic trees Δ_F and passing diagnostic trees Δ_P. A diagnostic tree is failing if $\delta_k(G_O, t)$ of the simulated value of output variable $o \in Y$ on the faulty design differs from the corresponding value of the golden device at time-step t. Otherwise, δ_k is called a passing diagnostic tree.

Diagnosis step 1:

For each variable x_i count the number c_{FAILED} of failing diagnostic-trees $\delta_k \in \Delta_F$, where x_i is present at least in one of the pairs (x, t) of δ_k. Select the variables x_i receiving a non-zero score c_{FAILED} as the set of suspected faults $X_{suspected}$ and sort the set $X_{suspected}$ according to the score c_{FAILED}. The variables with a higher score are more suspected of causing the error than the ones with a lower score (Raik, J., et. al., 2013).

Diagnosis step 2:

Perform step 1. For each variable $x_{step1} \in X_{suspected}$ count the number of passing diagnostic-trees $\delta_l \in \Delta_P$ c_{PASSED} , where x_{step1} is present at least in one of the pairs (x, t) of δ_l. Compute the score $c_{TOTAL}=c_{FAILED} / (c_{FAILED}+ c_{PASSED})$ for variables x_{step2}. Sort the set $X_{suspected}$ according to the score c_{TOTAL}.

Step 1 is more exact as it can be easily proven that at least one of the variables x_v that is labeling a vertex v along one of the main activated paths in simulated HLDDs must be also the cause of the error. However, step 2 may be unavoidable in order to guarantee a good diagnostic resolution, especially if the number of failing sequences is one or very small. In fact, the experiments presented in this subsection fully confirm this observation.

The straight-forward implementation of this backtracing algorithm could be time-consuming because of the square complexity introduced by the need to backtrace from each subsequent time step back to the initial time step. Therefore, in current implementation intermediate backtracing results were stored at each time step in order to gain speed.

4.1.4 Localization example

Consider the following example of design error localization on the basis of the erroneous GCD design description presented in Figure 2.10a. Let there be a given set of input stimuli (e.g. a functional test) and a set of correct output responses for the stimuli obtained on a golden model. Assume that there is a design error in it such that at state *s3* a faulty operation *a–b* is assigned to the variable *b* instead of the correct operation *b–a*. In Figure 4.2, two test sequences are presented as tables. Rows of the table show values of the variables at different time-steps. The first column *t* lists the time steps $t_0,...,t_6$, The next three columns present the values of input variables *res*, *in1* and *in2* in the test sequence. Final four columns show the values of the internal variables *state*, *a*, *b* and the primary output *out*. These values have been obtained by simulating the HLDDs in Figure 2.10b using Figure 2.8 *Algorithm 1*.

Figure 4.2a shows the test sequence for the design when primary inputs *in1* and *in2* hold values 4 and 2, respectively. This sequence passes the test, giving a correct response that the greatest common divisor of 4 and 2 is two. In Figure 4.2b, another sequence is presented, which produces an erroneous the test. Because of the design error, the primary output *out* receives an erroneous value.

t	res	in1	in2	state	a	b	out
t_0	1	4	2	-	-	-	-
t_1	0	-	-	s0	4	2	-
t_2	0	-	-	s1	4	2	-
t_3	0	-	-	s2	2	2	-
t_4	0	-	-	s1	2	2	-
t_5	0	-	-	s4	2	2	-
t_6	0	-	-	s4	2	2	2

a)

t	res	in1	in2	state	a	b	out
t_0	1	2	4	-	-	-	-
t_1	0	-	-	s0	2	4	-
t_2	0	-	-	s1	2	4	-
t_3	0	-	-	s3	2	-2	-
t_4	0	-	-	s1	2	-2	-
t_5	0	-	-	s4	2	-2	-
t_6	0	-	-	s4	2	-2	**2 ->** -

b)

Figure 4.2. Passing a) and failing b) test sequences for the GCD design

In order to locate the design error, a diagnostic tree is generated on the HLDD model of the GCD design presented in Figure 2.10b. Figure 4.3 presents the diagnostic tree for the passing test shown in Figure 4.2a while Figure 4.4 presents the diagnostic tree for the test shown in Figure 4.2b. As it can be seen from the Figures, the "tree" generated by Figure 3.2 *Algorithm 2* does not have a tree-like structure. It is rather a directed graph, where the vertices represent a subset of the

time-expansion model of the design. Directed edges show relations between the variables in the simulation process.

The algorithm starts at the time step when an output response is expected. For the test sequences in Figure 4.2 it is the time step t_6. Then, it continues towards the first time step and recursively generates the diagnostic tree $\delta(G_{out}, t_6)$. For the sake of compactness of presentation, the reset variable *res* was omitted from Figures 4.3 and 4.4. In addition, the operation $a=b$ (in Figure 4.3 is also given in a minimized form from $\neg(a>b)\wedge\neg(a<b)$ obtained by backtracing the HLDD for the *state* variable (see Figure 2.10b).

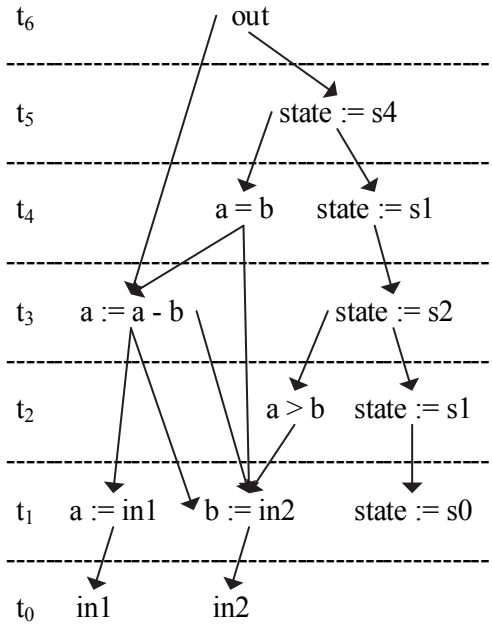


Figure 4.3. Diagnostic tree for the passing test in Figure 4.2a

The diagnostic trees presented in Figures 4.3 and 4.4 can be used for effect-cause diagnosis of design errors. Reasoning on the diagnostic trees takes place as follows. The diagnosis tree in Figure 4.3 of the passing test sequence in Figure 4.2a contains vertices that are unlikely to be related to the cause of the error because the sequence resulted in a matched output. However, the diagnostic tree in Figure 4.4 was backtraced from the mismatched output *out* at time-step t_6. These two backtraces should give us information about the location of the error.

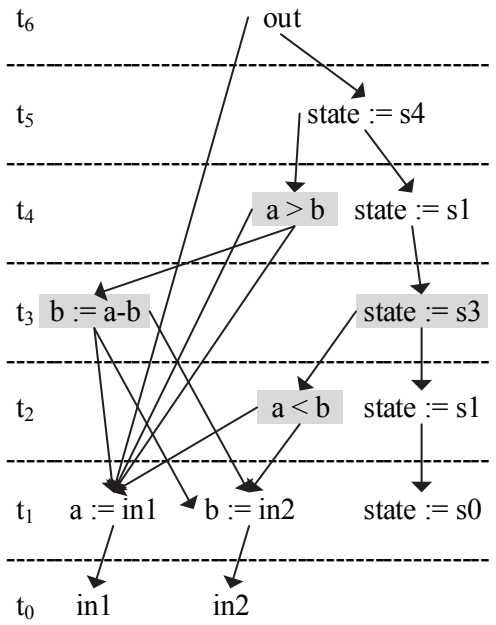


Figure 4.4. Diagnostic tree for the failing test in Figure 4.2b

Indeed, the vertex labeled by $b:=a-b$ (marked by grey background in Figure 4.4) is among the faults selected as suspects for causing the design error by the diagnosis step 2 presented in previous subsection. The four vertices with grey background are chosen as suspects because only these four vertices are present in the diagnostic tree of the failing sequence but are missing from the passing sequence. Thus, in this simple example they receive the highest score. In a real case there would be many failing and passing test sequences as well as there may be multiple faults. Furthermore, in most cases it is not possible to partition the test set into sequences. Figure 3.2 *Algorithm 2* takes the latter assumption. Therefore in experiments reported in current method, backtrace is started at each clock cycle for each output.

The HLDD-based diagnosis is related to known debugging techniques such as *program slicing* (Weiser M., 1981) and *critical path tracing* (Abramovici M., Menon P. R., Miller D. T., 1983). Modeling discrete systems by a system of HLDDs may be regarded as a form of program slicing, because a separate diagram is generated for each variable x in the program, reflecting the control flow branches where assignments are made to x and including the data assigned to x. Activating paths in HLDD diagrams using Figure 2.8 *Algorithm 1* is equivalent to critical path tracing. The technique of critical path tracing consists of simulating the fault-free system (true-value simulation) and using the computed signal values

for backtracing all sensitized paths from primary outputs towards primary inputs in order to determine the faults that would affect the primary output. In HLDDs the same task is solved in a single run as a byproduct of simulation.

4.1.5 Correction

Mutation analysis is a technique that was initially introduced to fulfill the task of evaluating the ability of testbenches to detect bugs in software programs. In this subsection applying mutation operators for correcting a faulty circuit is considered. Subsequent to the fault localization step described in Sections 4.1.3 and 4.1.4 mutation operators are applied to perturb the HLDD model of the RTL design in order to perform the correction. It is intuitively clear that this kind of correction may be extremely time-consuming in the worst case. The time required to correct the circuit is proportional to the product of the number of vertices, the number of mutants to be injected to each vertex and the number of test patterns in the test.

The design error localization technique presented in previous sections allows minimizing the number of vertices where the faults have to be injected. However, it is crucial to keep the number of mutants as small as possible. In this Thesis, the five *key* operators proposed in (Offutt A. J., Rothermel G., Zapf C., 1993) have been implemented. In experiments, those five operators 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 (Offutt A. J., Rothermel G., Zapf C., 1993). The 5 sufficient operators are ABS, which forces each arithmetic expression to take on the value 0, a positive value, and a negative value, AOR, which replaces each arithmetic operator with every syntactically legal operator, LCR, which replaces each logical connector with several kinds of logical connectors, ROR, which replaces relational operators with other relational operators, and UOI, which inserts unary operators in front of expressions.

The five operators have been implemented with the following constraints and specifics. UOI currently replaces only unary operators with other unary operators and ABS is applied to variables only, and not to expressions. Note also that in HLDD there are no signed/unsigned variables, but signed and unsigned relational operators exist. Therefore ROR replaces, both, signed and unsigned relational operators. In AOR mutation by division and mod operations is allowed and a check for the case of divide-by-zero is included. The reduced-5-key-operator strategy represents a do fewer strategy. The purpose would be to reduce the cost of the mutation analysis as much as possible.

4.1.6 Experimental results

Table 4.1 presents the main characteristics of the benchmarks used in the experiments and their respective test sets. The benchmarks include the Greatest Common Divisor (*gcd*) and the Differential Equation (*diffeq*) examples from the HLSynth92 and HLSynth95 academic benchmarks suite, respectively. The design *risc* is a processor example from a FUTEG research project. In addition, two real-world designs were included to the experiments. These were a commercial core for circular redundancy check (*crc*) from (Vertigo, 2009) and an open-source core *uart16750* from the OpenCores repository (OpenCores, 2010). The test stimuli for the academic benchmarks were generated by a hierarchical test pattern generator Decider (Raik J., Ubar R., 2000) while for *crc* the provided functional test bench was applied and *uart16750* was tested by 1000 randomly generated test vectors. The second column reports the system complexity in terms of the number of HLDD vertices. The third column represents the number of functions in the design. Finally, the fourth column shows the number of stimuli in the test suite.

Table 4.1. Benchmarks and their test sets

Design	# vertices	# functions	# gates	# FFs	# test stimuli
gcd	25	4	~500	48	4000
diffeq	39	9	~2500	80	16855
risc	61	16	~2000	96	4000
crc	232	74	~10000	171	193
uart16750	1747	401	~100000	1403	1000

In Table 4.2, the design error localization experiments are provided. Faults were injected into the design by randomly mutating a function one-by-one, so that during each diagnosis run only one function was mutated. The column 'success rate' shows the ratio of the times the actual location of the mutation achieved **the highest rank** in relation to all diagnosis runs. The column 'average resolution, # suspects' reports the average number of suspects that received the highest score. Here, the diagnostic resolution is very good for step 2 and two or more times worse for step 1. The same trend applies to the worst resolution, which reports the worst case suspected fault list size over all the faults injected. The final column reports the run times achieved on a PC, Dual-Core CPU, 2.6GHz, 3.25GB RAM, Windows XP operating system are provided. This time includes both performing step 1 and step 2 of the diagnosis algorithm. As it can be seen, the run times are very different. They do not only depend on the circuit size but also the number of vectors and the sequential depth of the designs. The run time for step 1 is actually very much shorter than the time for steps 1 and 2 combined, because in step 1, only mismatched outputs have to be backtraced. Table 4.2 excludes the error

localization details for the core *uart16750*. The time for localization for this core was in average 90.0 s on the 1000 vector test.

Table 4.2. Design error localization experiments

Design	Success rate, ratio of correct localizations		Average resolution, # suspects		Worst resolution, # suspects		Processing time, s
	step1	step2	step1	step2	step1	step2	
gcd	**4/4**	**4/4**	2.25	**1.00**	3	1	18.0
diffeq	**9/9**	**9/9**	3.33	**1.88**	6	3	700.0
risc	**16/16**	13/16	8.18	**1.93**	11	5	0.3
crc	**74/74**	69/74	31.83	**9.04**	50	20	0.5

As shown in the previous table, a majority of the errors injected in the experiments were identified as top suspects by the diagnosis algorithm. Because of this localization accuracy the mutation-based correction requires a very small number of iterations and thus a short run-time. See Table 4.3, which lists the average time to correct a design by applying mutation. The last column of Table 4.3 shows the average number of substitution functions (mutants) generated until the design was corrected.

Table 4.3. Mutation based correction experiments

Design	Average correction time. s	Average number of substitutions
gcd	0.0040	2.00
diffeq	0.0410	3.62
risc	0.0276	5.52
crc	0.0422	4.13
uart16750	0.5810	9.11

4.2 Localization case study

As a case study, the approach was evaluated by debugging an industrial processor developed as a part of the ROBSY (Reconfigurable On Board self test SYstem) project. This custom processor follows a new test approach (Meza-Escobar J.H., et.al., 2012), (Sachsse J., et.al., 2011) to improve the fault coverage and reduce the test time of Printed Circuit Boards (PCBs) during the manufacturing process, and it is developed in cooperation with a major vendor of PCB testing equipment.

The ROBSY processor is classified as a Single Instruction Single Data (SISD) processor with separated program and data buses (Harvard architecture). The processor has many of the properties of a Reduced Instruction Set Computer (RISC), and uses the Wishbone protocol (WB) for the I/O transactions. The current implementation of the processor core contains 17K lines of VHDL code. There are 481 direct signal assignment statements, 413 branches and 1573 conditions.

This subsection is based on Paper IV:

Jenihhin, Maksim; Tšepurov, Anton; Tihhomirov, Valentin; Hantson, Hanno; Raik, Jaan; Ubar, Raimund; Bartsch, Gu¨nter; Meza-Escobar, Jorge Hernan; Wuttke, Heinz-Dietrich. “Automated Design Error Localization in RTL Designs”. IEEE Design & Test of Computers, 1, 2014, pp.83–92.

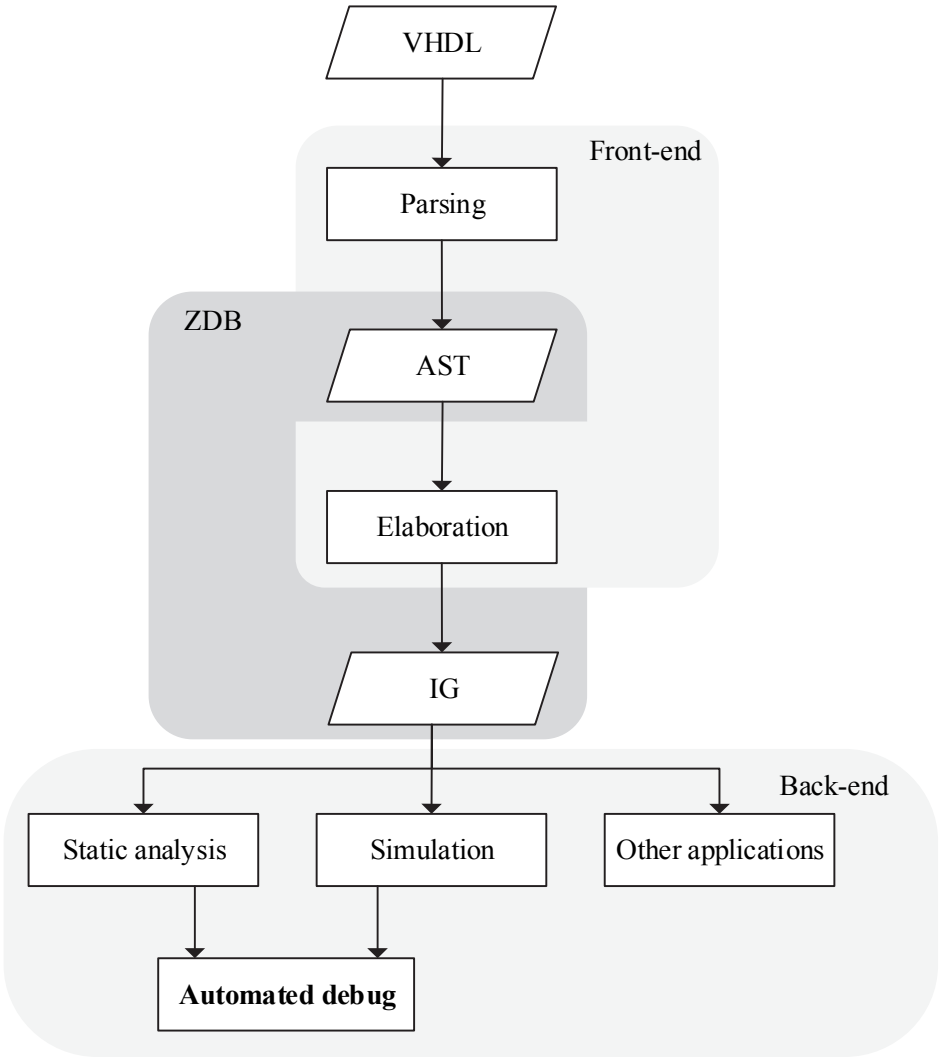


Figure 4.5. zamiaCAD framework

The bug localization method described here in Figure 4.5, has been implemented on top of an open source HDL-centric framework zamiaCAD (Tšepurov A., et. al., 2012), which puts emphasis on scalability and non-intrusiveness. The front-end of zamiaCAD includes a parser and an elaboration engine that both support full VHDL 2002 standard specification. On the back-end side the framework allows design simulation, static analysis and other applications such as synthesis and design structure visualization. zamiaCAD has an Eclipse IDE plug-in based agile graphical user interface for advanced design entry and navigation.

An object database ZDB (zamiaCAD Data Base), which has been custom-designed and highly optimized for scalability and performance is used for zamiaCAD applications. The database is HDL independent and able to accommodate extremely large designs. Full elaboration in zamiaCAD semantically resolves the Abstract Syntax Tree (AST) generated by the parser and results in a set of scalable Instantiation Graph (IG) data structures, stored in ZDB. *Instantiation Graph* is a data structure represented by a densely connected graph of semantically resolved objects representing elements of hardware design.

IG is the basis for zamiaCAD applications. In order to handle designs that do not fit into memory, ZDB containing the elaborated design is automatically and efficiently persisted to disk, thus saving processing time. As demonstrated in (Tšepurov A., et. al., 2012) the framework is capable of handling very large industrial multi-core designs (tens of millions of VHDL code lines, e.g. a SoC made of more than 3500 Leon3 processor cores).

4.2.1 Statistical bug localization

The statistical bug localization method assumes that design verification has been performed and an erroneous behavior at observable outputs of the processors has been detected. The method is based on four main phases: static slicing, dynamic slicing, statistical suspiciousness ranking of the HDL code items and an optional cone inspection phase. First, the design is simulated in order to obtain the list of executed statements and information about passed and failed test cases. A test case is considered to be passed if the simulated output responses match with expected ones and it is regarded as failed otherwise. Then, *static slicing* computation is performed based on generating reference graphs. Subsequently, *dynamic slicing* reduces the debugging analysis to all the code items that actually affect the design's faulty behavior for a given test case. Finally, the *statistical suspiciousness ranking* assigns a *suspiciousness score* to each code item based on its presence in the dynamic slices and on the information of passed/failed test cases. Intuitively, if a code statement occurs very frequently in executions revealing the error, it is very likely to contain a bug. The statistical ranking is performed for the statement items in the HDL code. In order to reveal the bug locations more accurately, the suspiciousness ranking is performed also hierarchically for the branches and conditions that the ranked statements may have. Figure 4.6 presents the statistical bug localization flow.

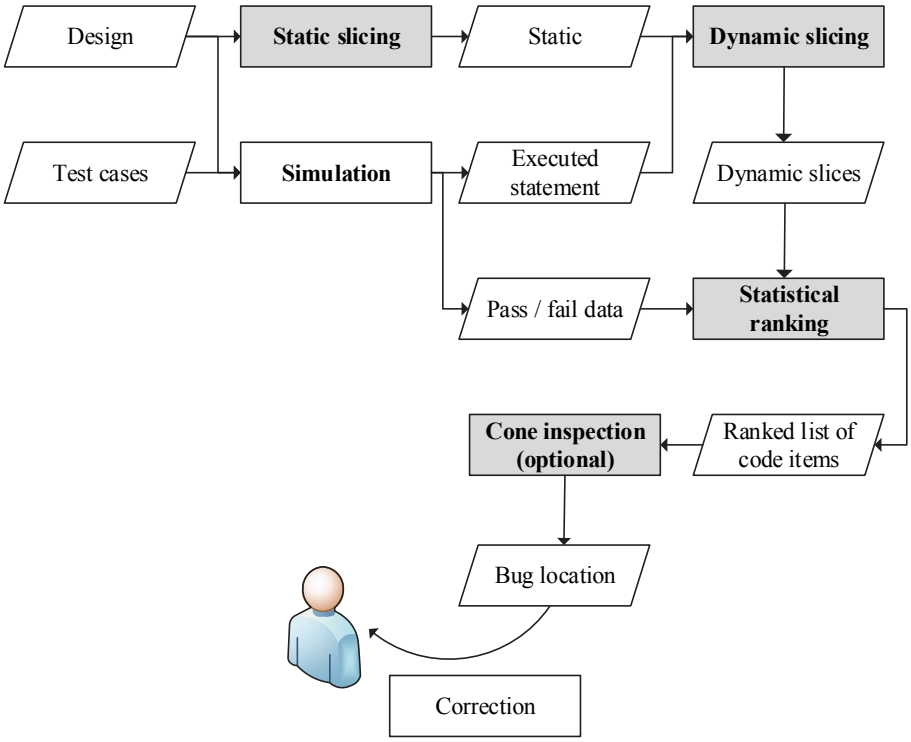


Figure 4.6. Statistical bug localization flow

Currently debugging is considered as a process of locating the failure, with the correction task being left to the designer. After the designer has received the ranked list of code items the following task is to localize the root cause of the erroneous behavior. Likely locations for bugs are in those code items having the highest suspiciousness scores in the list. In a simple case the designer has to inspect code items at the top of the ranked list, which score is higher than a preselected threshold value $S_{threshold}$. Ideally, when the automated localization method is accurate enough, then the artifact with the highest score leads us to the location of the bug, or alternatively the bug is localized among very few highly ranked artifacts. In the case study presented here it can be seen that in many cases the bug was attached to an artifact with the absolutely topmost rank. Thus, in the majority of situations inspecting the first, or few highest ranked, code artifacts reveals the bug location. However, there exist cases where the statistical ranking does not directly pin-point the root location of the error, and the actual location is not among the highest ranking code items, or too many items share the highest rank. In those cases the case study showed that it is easy to locate the bug by activating depth-limited forward and backward cones from the signals included to the highest ranked items. This type of cone activation is supported by the

zamiaCAD infrastructure of through-signal assignments search. The study showed that only low depth cones (up to 1 level) starting from the signals of the very highest ranked artifact need to be inspected in practice. Figure 4.7 illustrates the process of code inspection by the designer.

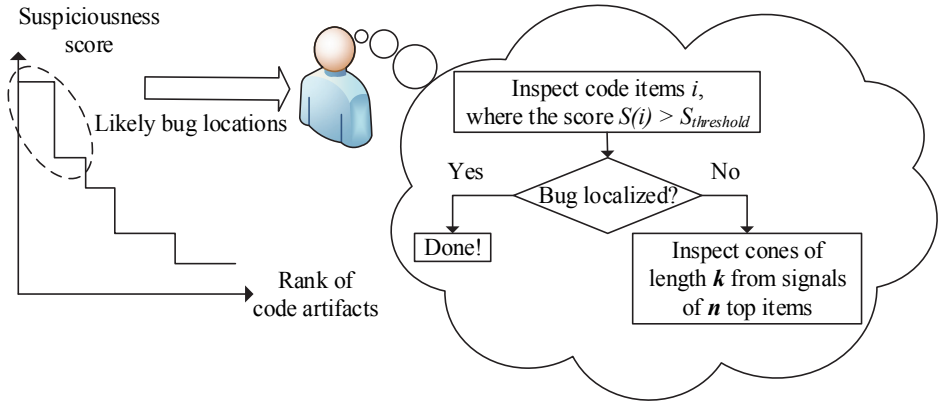


Figure 4.7. Inspection of likely bug locations

4.2.2 Motivational example

Consider the motivational design example shown in Figure 4.8 that presents a VHDL implementation of a signal chopper design named *chopper*. The *chopper* design has 3 processes calculating 4 outputs representing different chops for the input signal SRC based on the design configuration by inputs INV and DUP. It is assumed that the design has 5 individual tests T1-T5 of varied length each keeping the values of INV and DUV constant while flipping the value of the SRC input and having appropriate behavior of the clock and reset signals (CLK, CLKN, RST). The design has a bug on line 28 where instead of correct assignment F0 <= FF; the design has a buggy assignment F0 <= not FF;. Test cases T1, T3 and T4 are able to detect the bug and are referred to as *failing tests*, while test cases T2 and T5 pass despite the presence of the bug and are referred to as *passing tests*, respectively. The faulty behavior of the design caused by the failing tests is observed at output TAR_f (assigned at line 46).

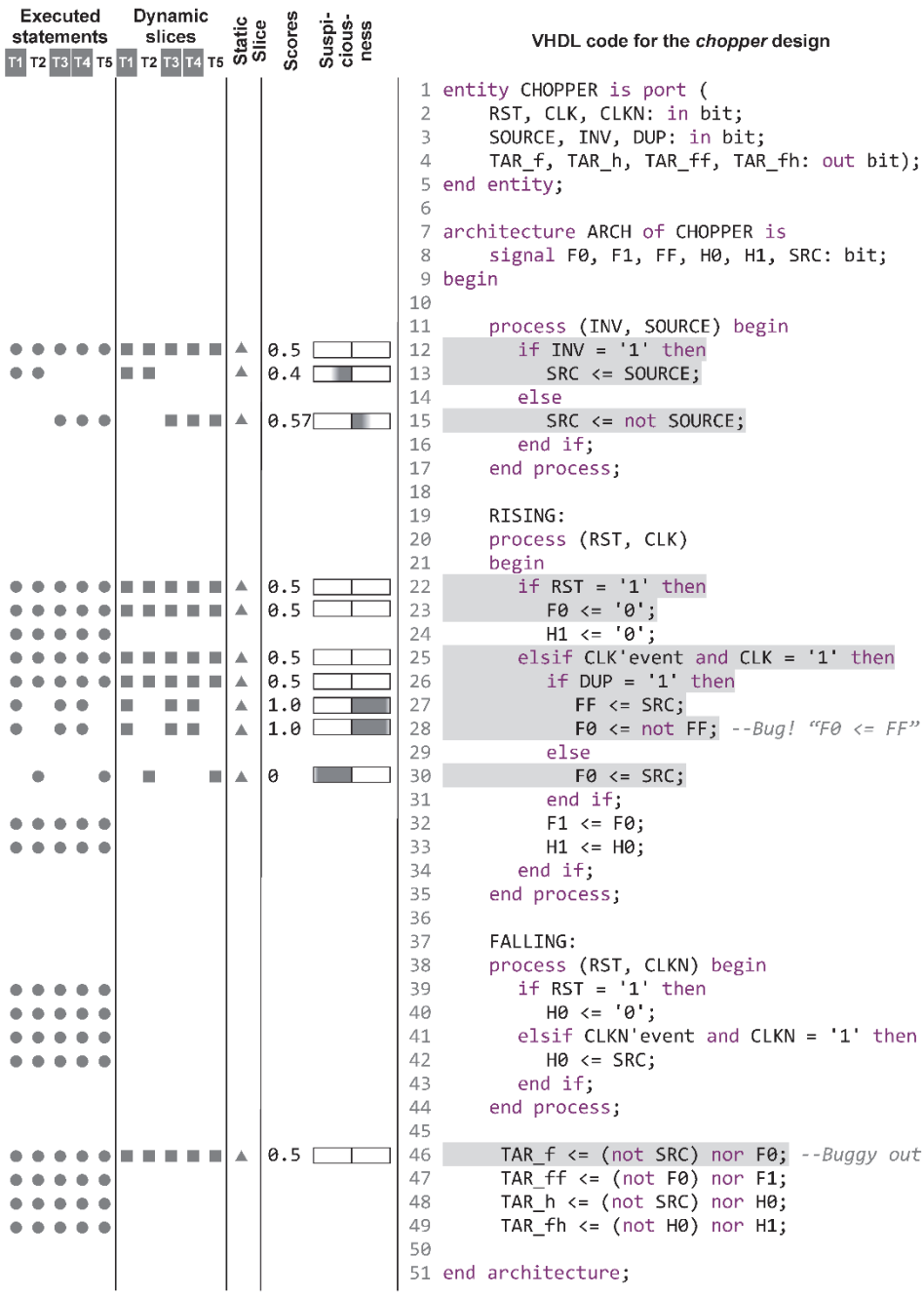


Figure 4.8. Bug localization on a motivational example

4.2.3 Static slicing

The presence of concurrent constructs, such as the ones found in HDLs versus sequential software languages, makes static slice computation considerably more complicated (Clarke E. M., et. al., 1999). zamiaCAD exploits its elaborated model referred to as Instantiation Graphs (IGs) (Tšepurov A., et. al., 2012) for this purpose. Given the IG model it is possible to perform a signal references search through its assignments, both backward to find the dependencies and forward to find other signals and variables influenced by the signal. The resulting *reference graph* has the signals and variables in its nodes and the dependencies are expressed by directed edges. It may contain cyclic dependencies and may be very large, especially if the search was initiated from primary inputs/outputs of the design. It is possible to limit such search by constraining the depth of the graph. An example dependency graph computed for the *chopper* design's output TAR_f is shown in Figure 4.9.

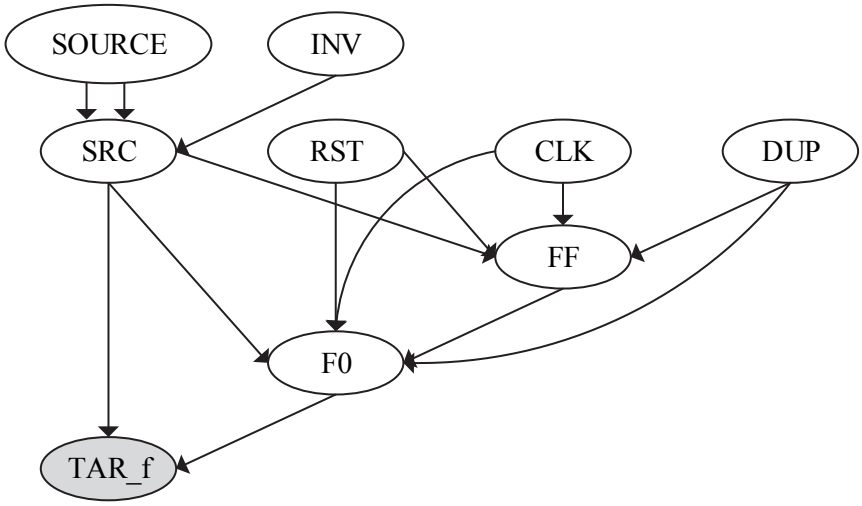


Figure 4.9. Through-signal-assignment search based backward reference graph on the signal TAR_f in the chopper design

Given the reference graph, the HDL statements representing the signal and variable dependencies in its edges are collected into a set. The resulting set represents a *static slice* on the signal of interest. However the approach for static slice computation does not consider the order of HDL assignment statements and can therefore be slightly too optimistic i.e. it can potentially include some statements that do not represent dependencies influencing the signal of interest into the static slice. It can be observed only for certain combinations of variable (versus signal) assignments which are a rare case in practical HDL descriptions.

The column Static Slice in Figure 4.8 marks VHDL statements of a static slice on the *TAR_f* output by *triangles*. Static slicing allows having a design ”filter” eliminating from the analysis space the design parts that do not influence the signal of interest. As a result in the *chopper* design example the entire process FALLING and a large part of other statements were excluded from the further analysis.

4.2.4 Suspiciousness ranking based on statement/branch coverage metrics

The statistical suspiciousness ranking procedure used in this Thesis is based on design simulation by a *diagnostic test*. A requirement for the diagnostic test is that it has to contain a set of independent test cases (e.g. separated by design reset) where both failing and passing test cases are represented. The quality of the statistical ranking is highly dependent on the quality of the diagnostic test. Functional tests for processors are suitable as diagnostic tests because they are divided into separate test cases for processor instructions, so that each such test case can be executed independently.

The column Executed Statements in Figure 4.8 marks the VHDL statements executed during design simulation with each of the 5 tests by *circles*. A fraction of the set of executed statements can be excluded from the further analysis by applying a static slice filter on an output signal where the faulty behavior was observed. This approach allows obtaining a *dynamic slice* of the design on this signal. The column Dynamic Slices in Figure 4.8 marks the VHDL statements taking part in the dynamic slices of the tests by rectangles. Thus the analysis space for the current example was reduced by 2.2 times (42 covered statements in dynamic slices versus 92 statement executions by the diagnostic test).

The statistical *suspiciousness score* for ranking of the HDL code item *i* is calculated as shown in Formula 1:

$$S(i) = \frac{\frac{Failed}{TotalFailed}}{\frac{Passed}{TotalPassed} + \frac{Failed}{TotalFailed}} \quad (1)$$

Where *S(i)* is the suspiciousness score value of the code item *i*, *Passed* and *Failed* are counts of passing and failing tests that covered the code item *i* in the dynamic slice, while *TotalPassed* and *TotalFailed* are the total numbers of the passing and failing tests in the complete diagnostic test, respectively.

Further zamiaCAD environment visualizes by colors the suspiciousness level of HDL code items (i.e. statements, branches, conditions) based on their scores *S(i)*. The score values are interpreted as follows:

- *S(i)=1* - the code item *i* is highly suspicious to contain or to lead to the bug
- *S(i)=0* - the code item *i* is above suspicion
- $S(i)=S_{threshold}$ - the code item *i* cannot be emphasized by the analysis

Here $0<S_{threshold}<1$ is the *suspiciousness threshold* specified by the designer and is by default equal to 0.5. The code items having score values in-between 0 and $S_{threshold}$ and in-between $S_{threshold}$ and 1 represent corresponding levels of suspiciousness. The *ranking* of code items is performed according to the score values starting from the highest. Code items without a score are either eliminated from the analysis by the static slice filter or not covered by the diagnostic test.

An example of applying the suspiciousness ranking to the *chopper* design is demonstrated in Figure 4.8. Here the assignment statements at lines 27 and 28 were calculated as the most suspicious (score S=1.0) and are assigned with the first rank. The statement at line 15 has score S=0.57 and therefore a lower rank. The assignment statements at lines 13 and 30 have scores 0.4 and 0.0 correspondingly and are therefore considered above suspicion and not assigned ranks.

4.2.5 Hierarchical analysis based on condition coverage

As it will be demonstrated further, the ROBSY processor case study emphasizes an important general category of design errors that are difficult to localize. They are bugs in complex condition expressions of conditional statements. E.g. Bug 1 in this case study is an erroneous comparison of one of the 35 conditions in a conditional assignment *when* of the ALU module. Localization of such bugs is assisted by suspiciousness ranking of conditions.

It is proposed to hierarchically rank conditions of the selected suspicious branches that belong to suspicious statements. Formula 1 is applied for this purpose considering for *i* branches and conditions instead of statements. A detailed example for hierarchical conditions ranking and its application for bug localization is demonstrated on example of a real bug (Bug 1) localization in the ROBSY processor further in the next section.

4.2.6 ROBSY processor: functional test

To verify the correct functionality of the Instruction Set Architecture (ISA), a functional test was developed. The functional test consists of a test program written in assembler, executed in a predefined order to test all the instructions supported by the processor. The test program is divided into sub-tests, where each sub-test is in charge of testing a specific instruction and setting register R1 to a specific value that acts as a sub-test label (error code). During the sub-test execution, it is evaluated if the values obtained in the registers, flags, etc., are as expected. Figure 4.10 has an example of a sub-test corresponding to the compare (CMP) instruction.

```
; check CMP with flags (register content unsigned)
MOV     R1, 01;     -- error code 01--
MOV     R2, A3;
MOV     R2, 05;
JZ      fail;       if R2 equals 05      (jump zero)
JC      fail;       if R2 < 05           (jump carry)
CMP     R2, A3;
JNZ     fail;       if R2 not equal 05   (jump not zero)
JC      fail;       if R2 < 05
MOV     R3, A4;
CMP     R2, R3;
JNC     fail;       if R2 > R3           (jump not carry)
```

Figure 4.10. ROBSY processor test program

In the case of an unexpected value, the processor goes to the code section labeled with “fail”. Here the execution is aborted and the error code of the failed sub-test is written to the WB register. If all sub-tests are successfully executed, a pass code is written to the WB register, interrupts are activated and the processor enters into an infinite loop. By looking at the value of this register at the end of the simulation, it is possible to distinguish if the test execution was successful or not.

4.2.7 Set of documented design errors

The ROBSY design team has documented a set of their VHDL coding bugs that have the following nature:

- Bug 1 A *wrong register* is used as one of the operands in a very long conditional expression (35 operators) inside a conditional signal assignment. Possibly, due to a copy-paste error.
- Bug 2 An entire *conditional sub-expression* (3 operators) resides in the wrong branch of a conditional signal assignment, which contains 9 branches in total.
- Bug 3 Both, a *missing branch* and a *missing driver* in a short conditional signal assignment.
- Bug 4 A *wrong enumeration constant* is used in a comparison operation inside a conditional signal assignment.
- Bug 5 A *wrong driver* is used in a conditional signal assignment. More specifically, register *R* is not updated with its newly computed value typically stored in *R_next* or *R_new* signal. Instead, the same register *R* is used as a driver, which indicates an obvious copy-paste error.
- Bug 6 A *missing conditional sub-expression* (3 operators out of 6 required ones) in one of the 4 branches of a conditional signal assignment.
- Bug 7 *One bit* of a register is always and unconditionally set to 0. The whole code line to blame is unnecessary and hence incorrect.

4.2.8 Experimental results

This section presents experimental results for the design errors localization approach evaluation on the industrial processor ROBSY. For the purpose of the current approach the original functional test (i.e. an Assembler program) was split into 31 independent sub-tests, each targeting a separate instruction. Each of the 7 buggy versions of the processor was simulated with the resulted diagnostic test.

#	Stm. score	Bran. score	Cond. score	Line	Source code lines
			alu.vhd		
6	0.51			88	
5	0.55			104	
2	0.67			108	
1	0.80			110	
5	0.55			116	
5	0.55			127	
				...	
3	0.64	0.64		260	svFlag_new(0) <= '1' when afClass=cfClass_1
			0.64^{1}_{T}	261	and ((svOp_mux(cnD_w)=REG_SOURCE_DEST_IN(cnD_w)--add case
				262	and svOp_mux(cnD_w)/=svRes(cnD_w)
				263	and ((aCmd=cvCmd_ADD_R_R and c_en_ADD_R_R)
				264	or (aCmd=cvCmd_ADD_R_IMM and c_en_ADD_R_IMM)))
			0.51^{1}_{T}	265	or (svOp_mux(cnD_w)/=REG_SOURCE_DEST_IN(cnD_w)--sub case
					-- Bug: correct compar. between REG_SOURCE_DEST_IN and svRes
				266	and svOp_mux(cnD_w)/=svRes(cnD_w)
			0.69^{1}_{T}	267	and ((aCmd=cvCmd_SUB_R_R and c_en_SUB_R_R)
			0.55^{2}_{T}		
			0.75^{2}_{T}	268	or (aCmd=cvCmd_SUB_R_IMM and c_en_SUB_R_IMM)
			0.57^{1}_{T}	269	or (aCmd=cvCmd_CMP_R_R and c_en_CMP_R_R)
			0.55^{2}_{T}		
			0.51^{1}_{T}	270	or (aCmd=cvCmd_CMP_R_IMM and c_en_CMP_R_IMM)))
				271	or (REG_SOURCE_DEST_IN(cnD_w)/=svRes(cnD_w)--shift cases
			0.55^{1}_{T}	272	and ((aCmd=cvCmd_SHL_R and c_en_SHL_R)
			0.55^{2}_{T}	273	or (aCmd=cvCmd_SHR_R and c_en_SHR_R))))
5	0.55	0.55		274	else '0' when afClass=cfClass_1 --overflow reset
				275	and ((aCmd=cvCmd_ADD_R_R and c_en_ADD_R_R)
				276	or (aCmd=cvCmd_ADD_R_IMM and c_en_ADD_R_IMM)
				277	or (aCmd=cvCmd_SUB_R_R and c_en_SUB_R_R)
				278	or (aCmd=cvCmd_SUB_R_IMM and c_en_SUB_R_IMM)
				279	or (aCmd=cvCmd_CMP_R_R and c_en_CMP_R_R)
				280	or (aCmd=cvCmd_CMP_R_IMM and c_en_CMP_R_IMM)
				281	or (aCmd=cvCmd_SHL_R and c_en_SHL_R)
				282	or (aCmd=cvCmd_SHR_R and c_en_SHR_R))
NA	0.50			283	else svFlag(0);
			data_interface_mod.vhd		
2	0.67			155	
2	0.67			158	
			gprs_mod.vhd		
4	0.60			97	
			state_machine.vhd		
4	0.60			100	
4	0.60			123	
4	0.60			168	

Figure 4.11. Details of automated localization

Figure 4.11 demonstrates the hierarchical localization of *Bug 1*. The grey areas denote that some detailed information was omitted from the figure. First the dynamic slices (intersection of executed statements with the static slice on an observable faulty output) were generated for all of the test cases and the statistical

suspiciousness ranking was performed. This analysis resulted in 14 statement items (out of the initial total 481 assignment statements) whose suspiciousness score S was above the default suspiciousness threshold $S_{threshold}$ =0.5. The figure shows in the second column *Stm. score* the scores for these 14 suspicious statements, and in the first column their rank based on the score (6 ranks in total). Most of the statements with high scores were found in the ALU processor module (file *alu.vhd*).

The figure demonstrates a part of the actual VHDL code for the conditional assignment of the overflow flag signal *svFlag_new(0)*. *Bug 1* is located in the condition expression at line 266 (correct comparison had to be made between signals *svRes(cnD_w)* and REG_SOURCE_DEST_IN instead of *svOp_mux(cnD_w)*). This complex conditional assignment (lines 260-283) contains 3 individual assignments at lines 260, 274 and 283. The first two assignments have 3rd and 5th ranks while the last one has the score $S = 0.5$ and is filtered out together with other statements with scores 0.5 and less.

The automated localization iteratively advises the designer to consider as bug location candidates the statements with the highest ranks starting with the one at line 110 in *alu.vhd*, followed by statements at line 108 in *alu.vhd* and lines 155 and 158 in *data_interface_mod.vhd* (complemented with hierarchical analysis of the corresponding branches and conditions). Further it will advise the designer the statement at line 260 in *alu.vhd* with the next rank 3 and score value 0.64. The hierarchical analysis will proceed with score computation of the branches of this statement (column *Bran.score*). The suspiciousness scores of separate condition evaluations to 'true' and 'false' related to this branch artifact are also calculated. The ones that have score $S > 0.5$ are specified in column *Cond. score*. One of the highest scores here has the logical *and* at line 267. One of its operands is actually the incorrect signal comparison documented as *Bug 1*.

Table 4.4 demonstrates the statistics of applying the bug localization approach to all of the 7 bugs. The second column depicts the ratio of failing versus passing test cases for the bugs. The third column in the table shows how many statements were proposed as bug location candidates by the statistical ranking step. The column also demonstrates these numbers in percentage of the total number of the statements which was 481. The fourth column shows the rank of the statement actually containing the bug. If ranking alone was not sufficient then the column shows the rank of the statement from which cone inspection was activated. Column six shows the direction (i.e backward/forward) and the depth of the cone if cone inspection was required while column seven shows the number statements added as bug candidates by this step.

Table 4.4. Statistics of the bug localization approach

Bug data		Automated localization					Manual debug
		Statistical ranking		Cone inspection		Time (min)	
Bug name	Failed / Passed Test cases	Statements cand. / %	Located stm. rank	Cone dir. / depth	Added stm. cand		Time (h)
Bug 1	4 / 24	14 / 2.9 %	3	-	-	2	4
Bug 2	2 / 26	7 / 1.4 %	1	-	-	2	2
Bug 3	2 / 26	20 / 4 %	3	-	-	2	4
Bug 4	1 / 27	6 / 1.2 %	(1)	fw / 1	21	2 +(5)	4
Bug 5	2 / 26	11 / 2.3 %	1	-	-	2	2
Bug 6	1 / 27	8 / 1.7 %	(1)	bw / 1	13	2 +(10)	5
Bug 7	1 / 27	21 / 4.3 %	(1)	fw / 1	10	2 +(1)	1

The diagnostic test was sufficient to automatically localize 4 of the 7 bugs by the ranking step only. Pessimistic estimation of the candidates' count with the shown rank or higher that was necessary to check before the bug discovery is 5, 1, 12, and 4 for Bugs 1, 2, 3 and 5, respectively. Localization of the remaining three bugs was required cone inspection as an addition step. The cones of a limited depth were generated by zamiaCAD by the through-signal-assignment reference search (also used for static slice computation) from the signals involved in the highly ranked assignment statements. In the current case study Bugs 4, 6 and 7 were present within the cones of depth 1 on the signals from the statements with the highest rank. These cones have added 21, 13, and 10 additional candidates as shown in column six.

The last two columns in Table 4.4 compare time required for bug localization by the automated localization approach and conventional manual debug process. The time values for the manual process are reported by the ROBSY processor designers based on their experience with locating these bugs using commercial

design environments. The time reported for the automated approach consists, first, of time spent for the statistical ranking step which is mainly spent for simulation of the 28 test cases and constantly equals to 2 minutes for the case study diagnostic test. Second, it is estimation of time spent for manual cone inspection (shown in brackets). The runtime required for the static slices and cones construction in zamiaCAD takes a fraction of second and can be neglected.

Previous state-of-the-art automated hardware design error localization approaches are not capable to handle industrial size RTL designs such as ROBSY. Therefore direct comparison to other than manual approaches was not possible for this empirical study.

4.3 Design error correction for C

Verification is increasingly becoming the bottleneck in designing digital systems. In fact, most of the verification cycle is not spent on detecting the occurrences of errors but on debugging, consisting of locating and correcting the errors. However, automated design-error debug, especially at the system-level, has received far less attention than error detection. The current section presents an automated approach to correcting system-level designs. Dynamic-slicing and location-ranking based method for accurately pinpointing the error locations combined with a dedicated set of mutation operators for automatically proposing corrections to the errors are presented. In order to validate the approach, experiments on the Siemens benchmark set have been carried out. The experiments show that the method is capable of correcting three times more errors compared to the state-of-the-art mutation-based correction methods while examining fewer mutants.

This subsection is based on Paper V:

Raik, Jaan; Repinski, Urmas; Hantson, Hanno; Jenihhin, Maksim; Di Guglielmo, Giuseppe; Pravadelli, Graziano; Fummi, Franco. “Combining Dynamic Slicing and Mutation Operators for ESL Correction”. Proceedings of the 17th IEEE European Test Symposium, IEEE Computer Society Press, 2012, pp. 1–6.

4.3.1 State-of-the-art

In debugging, the error localization is considered the most time expensive activity and its quality affects the following (manual or automatic) correction phase (Vessey I., 1985). In manual error localization, engineers run the design with some input stimuli till they observe a failure; then, they iteratively place breakpoints, analyze the system status, and backtrack to the error origin using a source-level debugger, e.g., GNU GDB (Stallman R. M., Pesch R. H., 1991).

On the other hand, automatic error localization is based on different methodologies. In particular, they may be simulation-based and use coverage information (Wong W. E., Debroy V., Choi B., 2010), (Wong W. E., Qi Y.,2009), (Jones J. A., Harrold M. J., 2005), binary search (Cleve H., Zeller A., 2005), and statistical analysis (Liblit B., et.al., 2005), (Liu G., et.al., 2006). As well, formal approaches for error localization exist that are very effective but may suffer the state-explosion of the underlying solver (Staber S., Jobstmann B., Bloem R., 2005), (Könighofer R., Bloem R., 2011). Of all these solutions, the Tarantula (Jones J. A., Harrold M. J., 2005) coverage-based approach has been proven suitable for real-world designs. Present Thesis provides an improvement for error localization, which significantly reduces the overhead of the error-correction phase based on ESL-code mutation.

After an error is detected and localized, it should be corrected. Design-error correction for combinational circuits has been thoroughly studied for decades. There exist, both, error-matching-based (Madre J. C., Coudert O., Billon J. P., 1989), (Könighofer R., Bloem R., 2011), (Abadir M. S., Ferguson J., Kirkland T. E., 1988) and resynthesis (Ali M. F., et.al, 2005) approaches. There have also been attempts to generalize the above mentioned methods for design-error correction of sequential circuits (Ali M. F., et.al, 2005), (Wahba A., Borrione D., 1995). In particular, the SAT-based correction and re-synthesis approach developed by (Smith A., Veneris A., Viglas A., 2004) has been extended to higher abstraction levels such as register-transfer level (Chang, K.-H., et al., 2007), (Chang K.-H., Markov I. L.; Bertacco V., 2008). The re-synthesis approach for high-level design-error correction has two main limitations. The correction is *not readable* and thus cannot be checked by the designer. Moreover, the *correction is limited* to the set of used stimuli: this is due to the logic optimization freedom created by the partial truth table of the portion to be corrected.

Finally, in (Könighofer R., Bloem R., 2011) a symbolic-simulation-based approach is proposed for both error correction and localization in ESL designs described as C programs. All the reasoning is done with a Satisfiability Modulo Theory (SMT) solver (De Moura L., Bjorner N., 2009), thus it can be classified as a formal method. In particular, the approach performs the error correction by using approximation heuristics and a template-based methodology, which gives

readable corrections. In the experimental-result section, comparisons of the approach presented in this Thesis and (Könighofer R., Bloem R., 2011) are provided, showing better correction capability and preserving correction readability.

4.3.2 Error correction method

At electronic-system level (ESL), designs are described in an algorithmic way with a high level of abstraction with respect to the final hardware implementation (Könighofer R., Bloem R., 2011). In order to formally represent the ESL algorithmic descriptions the flowgraph model has been chosen as an underlying model. In such flowgraph, there is a one-to-one correspondence between the program statements and nodes and edges represent the control flow of the program. More precisely, the model representation is a special case of flowgraph known as the *hammock graph* (Kašjanov V.N., 1975), which was proposed for program slicing in (Weiser M., 1984).

Definition 2: A *hammock graph* is a structure $H=<N, E, n_0, n_e>$, where N is a set of nodes, E is a set of edges in $N \times N$, n_0 is the *initial node* and n_e is the *end node*. If (n, m) is in E then n is an *immediate predecessor* of m and m is an *immediate successor* of n. A *path* from a node n_1 to a node n_2 is a list of nodes p_0, $p_1, ..., p_k$ such that $p_0 = n_1$, $p_k = n_2$, and for all i, $1 \leq i \leq k - 1$, (p_i, p_{i+1}) is in E. There is a path from n_0 to all other nodes in N. From all nodes of N, excluding n_e, there is a path to n_e.

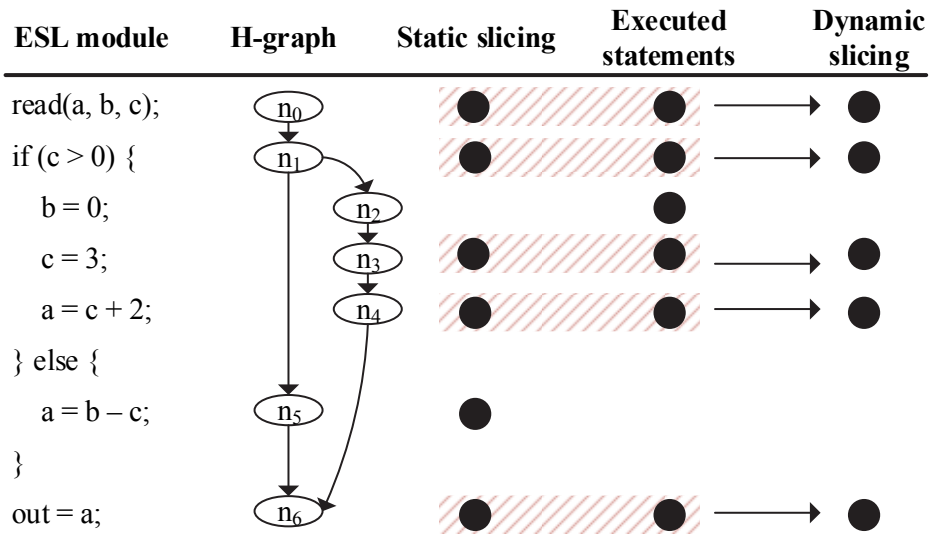


Figure 4.12. The ESL description is modeled as a flowgraph, i.e., hammock graph. Simulation and slicing are performed on the model representation

Figure 4.12 presents a simple ESL functionality in C language, i.e., column *ESL MODULE*, and the corresponding flowgraph *H*, i.e., column *H-GRAPH*. In the following, some definitions are introduced in order to explain the slicing process on flowgraph structures.

Program slicing (Weiser M., 1984) is a technique for extracting portions of a program affecting a selected set of variables of interest. By focusing on the computation of only few variables the slicing process can be used to discard portions of the program, which cannot influence these variables, thereby reducing the size of the program. The reduced program is called a slice. Slices reproduce a projection from the behavior of the initial program. This projection represents the values of certain variables as seen at certain statements.

Definition 3: A *slicing criterion* of a program P is a tuple (x,V), where x is a statement in P and V is a subset of the variables in P.

Informally, given a slicing criterion $C = (x, V)$, a static program slice S consists of all statements in program P that may affect the value of $v \in V$ for a set of all possible inputs at the point of interest, i.e., at the statement x. Static slices are computed by finding consecutive sets of indirectly relevant statements, according to data and control dependencies. Unfortunately, the size of the slices so defined may approach that of the original program. Indeed, static slicing preserves the behavior of the original program for *all the possible* input values. In this case, the

usefulness of the slices in debugging tends to diminish as the size of the slices increases.

In (Korel B., Laski J., 1988) a more accurate slicing technique, i.e., *dynamic slicing* was introduced. Dynamic slicing provides more narrow slices, preserving the behavior of the original program and consisting of only the statements that influence the value of a variable for *a given input*.

Figure 4.12 illustrates the concepts of static and dynamic slicing applied to the flowgraph representation of an ESL functionality. In particular, the Figure reports an intuitive correlation between static slicing, execution trace, and dynamic slicing. Let us consider, for example, the slicing criterion $C = (n_6, \{out\})$. In this case the n_6 is the end node n_e of the hammock graph. The black dots in the column *STATIC SLICING* indicate the statements included into the slice in case of static slicing. These mark the statements that are needed in order to calculate the value of the variable *a* at the node n_6. It can be seen that the node n_2 is excluded from the slice because the statement b=0 is not necessary for calculating the value of the variable *out* at the node n_6.

The column *DYNAMIC SLICING* refines that Alice according to the execution trace obtained with actual value assignments. Assuming that variables get assignments a=2, b=4 and c=7, the slice shown in the last column of Figure 4.12 is obtained. The *else* branch of the condition is not activated by these input values and therefore the respective statement are not included into the slice. The column *EXECUTED STATEMENTS* shows all the statements that were executed in current trace with the given input assignments. As one can see, the statements occurring in the dynamically-computed slice are a proper subset of the statements in the statically-computed Alice and execution trace. This narrows the search space of the following step for ranking the error locations.

In this subsection, a design-error localization approach is considered, where ESL implementations fail on some of the given test cases. The error localization relies on error detection results. The mechanisms of the latter are out of scope of this Thesis and may involve for instance the golden output responses specified by the test cases, assertions supplied with the test environment or results obtained from analyzing the specification (e.g. UML, SW program, etc.).

The error localization method is based on calculating the dynamic slices for all the observable outputs of the system with all the test cases. Depending on whether an output response obtained by a given slice is correct or not, the slice is marked as a passed or failed one, respectively. Then, a statistical and coverage-based approach is implemented assigning score to flowgraph nodes based on the number of times they were included into failed slices with respect to the number of times they occur in the previous executions. Finally, the flowgraph nodes are ranked according to this score, referred to as the *suspiciousness score*.

In details, the error ranking and localization takes place as follows. Let T be a test suite consisting of test cases t_i for verifying the functionality of the ESL description. Let H be the flowgraph associated with the description. Let y_j be the observable output variables of the design. Finally, let the nodes n_j of H be the respective nodes were value assignments to y_j are made. Over each test case t_i and, in turn, over each observable output variable y_j a dynamic slice d_{ij} is generated according to the values of current test case t_i and a slicing criterion $C = (x_j, \{y_j\})$, where x_j is the statement at the flowgraph node n_j.

If y_j resulted in a correct value at test case t_i, then the dynamic slice d_{ij} is included into the set of passed slices D_{PASSED}. Otherwise, it is included to the failed slices, i.e. $d_{ij} \in D_{\text{FAILED}}$. Each node n_k of flowgraph H gets a score according to the number of times c_{FAILED} it is included into the set of failed slices D_{FAILED} and the number of times c_{PASSED} it is included into the set of passed ones, i.e. D_{PASSED}. This score of suspiciousness is calculated as shown in Formula 2:

$$suspiciousness(n_k) = \frac{C_{FAILED}}{C_{FAILED} + C_{PASSED}} \quad (2)$$

The nodes n_k are ranked according to the suspiciousness score with more probable candidates for error correction having higher score values. This ranking is used for selecting statements to be corrected by the mutation-based methodology presented in the following sections.

4.3.3 Mutation-based error correction

Traditionally mutations are performed by perturbing the behavior of the program in order to see if the test suite is able to detect the difference between the original program and the mutated versions. The effectiveness of the test suite is then measured by computing the percentage of detected, or *killed*, mutations.

In this subsection, mutation operators are applied for correcting erroneous circuits. The goal is to develop an error-matching based correction approach, which would be capable of modeling realistic design errors. Moreover, it is crucial to select a limited number of mutation operators, because the perturbation and simulation of erroneous design implementations with a large number of error locations and mutant operators would become prohibitively time-consuming.

Table 4.5 presents the set of ESL-mutation operators that were implemented in the error-matching based correction method. Since ESL descriptions in C language are targeted, the focus is on algorithmic aspects of the description and software-specific constructs and related errors, such as dynamic-memory allocation, pointer arithmetic, and file I/O are not considered. This permits to reduce the overhead of the code mutation phase and address only system-level issues.

Table 4.5. List of mutation operators for correction

Mutation operator	C operators/examples
AOR (arithmetic operator replacement)	+, - *, /, %
ROR (relational operator replacement)	==, !=, >, <, >=, <=
LCR (logical connector replacement)	&&, ||
ASOR (assignment operator replacement)	+=, -=, *=, /=, %=, =
UOR (unary operator replacement)	+, -, ~, !
Bitwise operator replacement	<<, >>, &, |, ^
Bitwise assignment operator replacement	<<=, >>=, &=, |=, ^=
Increment/decrement operator replacement	x++, ++x, x--, --x
Number mutation (decimal digit replacement in integers, floats and array indexes)	0-9
Constant replacement unary minus / unary plus / zero	+C, 0, -C

In particular, the mutation operators include replacement of C language operators, which have been divided into several groups: arithmetic operators, relational operators, assignment operators, unary operators, etc. In addition, number mutations are performed by replacing each decimal digit in the numeric values one-by-one with other decimal values. This includes both, integer and floating point numbers and it covers also the array indexes. Also, constants are mutated by inserting unary operators + and – as well as replaced by zero.

Figure 4.13 explains the mutation-based correction process. Subsequent to the error localization step described in subsection 4.1.3, which ranks the statements of the program, the suspected error locations are iteratively tried according to their rank. The operators in the statements are, in turn, iteratively substituted by mutation operators, i.e., valid operators from the same category. In other words, replacing arithmetic operators by arithmetic operators, relational operators by relational ones etc. These iterations stop when the simulation result confirms that the mutated program provides output responses equal to the golden output responses, in other words, a correction has been found. Otherwise the process continues until there exist untried error locations and/or mutant operators, or when a user-specified time limit is reached.

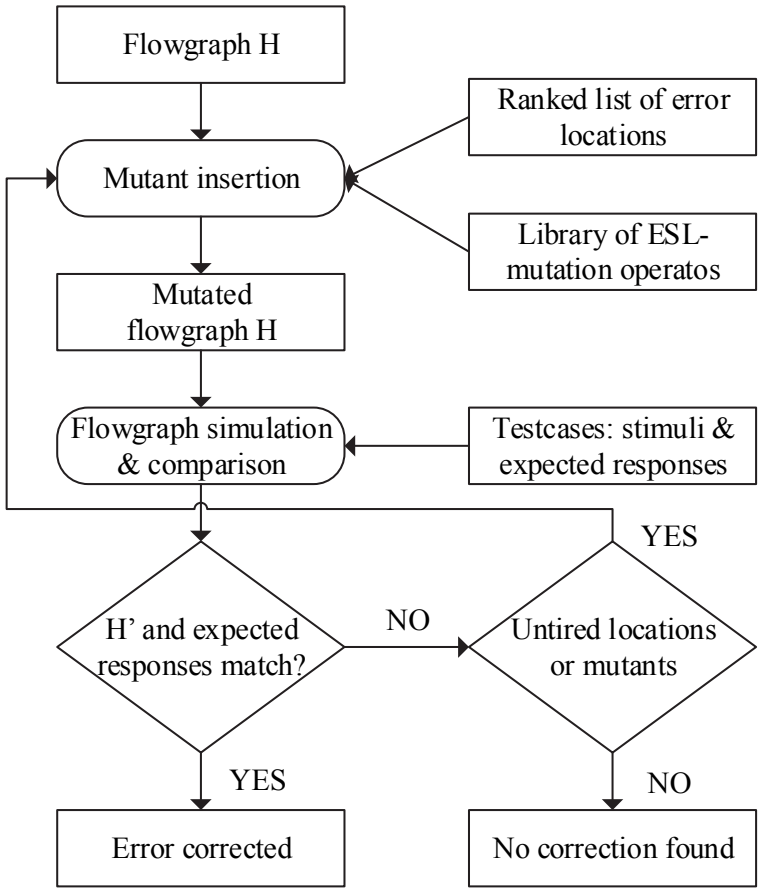


Figure 4.13. The mutation-based error correction flow

This mutation-based correction method is an error-matching approach. Error-matching is known to have the limitation that it is generally not capable of fixing errors that are not included to the model. On the other hand, the mutation-based error-matching provides easy-to-read corrections of system-level descriptions. Moreover, the experiments show that the mutation-based approach can fix some of the not modeled errors by proposing alternative but equivalent fixes.

4.3.4 Experimental results

Current debugging approach has been implemented as a module of a larger tool, i.e., FoREnSiC (DIAMOND, 2011), which also features formal and semi-formal approaches for debugging of ESL design (Könighofer R., Bloem R., 2011). This framework supports debugging of algorithmic descriptions of hardware in C

language. In order to evaluate the method, experiments on Siemens benchmark suite (Siemens, 2010) comparing it to a recently published formal (Könighofer R., Bloem R., 2011) and dynamic (Debroy V, Wong W. E., 2010) technique were carried out. The front-end of FoREnSiC was applied for generating the flowgraph models for the C language designs (Raik J., et.al., 2012).

In Table 4.6, the main characteristics of the benchmark circuits are presented. Column LoC shows the number of lines of code for the corresponding C designs; column TEST-CASE # shows the number of test cases for the design, which include both failing test stimuli and passing stimuli; finally, column FAULTY-VERSION # shows the number of faulty versions of the benchmark programs. One faulty version from benchmark schedule2 was exploded because the design error did not result in any test case failure making the correction process meaningless.

Table 4.6. Characteristics of Siemens benchmarks

Design	LoC	Test-case #	Faulty version #
replace	507	5542	32
schedule	397	2650	9
schedule2	299	2710	9
tcas	174	1608	41
tot_info	398	1052	23
print_tokens	539	4130	7
print_tokens2	489	4115	10

In Table 4.7, the results of the design error correction experiments are presented. Current method is compared to two recently published methods: a symbolic-simulation-based method (Könighofer R., Bloem R., 2011) and a mutation-based method (Debroy V, Wong W. E., 2010). For each methodology, columns # FIXED show the number of corrected faulty model versions and Columns % FIXED show the percentage of corrected models from the total number of faulty model versions.

Table 4.7. Design error repair experiments

Design	(Könighofer R., Bloem R., 2011)		(Debroy V, Wong W. E., 2010)		Current method		
	# fixed	**% fixed**	**# fixed**	**% fixed**	**# fixed**	**# fixed**	**Mutants examined**
replace	-	-	3	9.4	**12**	**37.5**	855.2
schedule	-	-	0	0.0	**2**	**22.2**	188.0
schedule2	-	-	1	11.1	**3**	**33.3**	460.7
tcas	7	17.1	9	22.0	**26**	**63.4**	131.1
tot_info	-	-	8	34.8	**15**	**65.2**	781.3
print_tokens	-	-	0	0.0	**1**	**14.3**	825.0
print_tokens2	-	-	0	0.0	**7**	**70.0**	952.3
Total:		*N/A*		*16.0*		***50.4***	*599.1*

As it can be seen from the table, current approach clearly outperforms (Könighofer R., Bloem R., 2011), where only 8 faulty versions (out of 41) of *tcas* design are analyzed. The approach in (Könighofer R., Bloem R., 2011) is able to correct 7 out of these 8 faulty versions, whereas the current approach corrects all 8. Furthermore, due to the underlying solver, the formal approach (Könighofer R., Bloem R., 2011) is only able to model the designs which bit-width is reduced from 32 to 8 bits.

With respect to (Debroy V, Wong W. E., 2010), the current method increases the percentage of successful corrections from 16.0% to 50.3%. Thus, the rate of corrections is increased by the factor of three.

It is important to stress that the increase in successful fixes does not come at the expense of more mutants to be considered. The last column of Table 4.7 shows the localization accuracy in terms of the average number of examined mutants per design error. In fact, this number is 599.1, which is even slightly fewer than 642 mutants in average obtained in (Debroy V, Wong W. E., 2010).

The significant increase in successful corrections with respect to (Debroy V, Wong W. E., 2010) is due to the selection of mutation operators, which are not limited to control flow errors. The run-time advantages in terms of the number of mutants examined comes partly from the more accurate diagnosis method based on dynamic slicing and location ranking.

4.4 Conclusions

The chapter presented a method for automated debug of multiple simultaneous design errors for RTL circuits. A critical path tracing based error localization method was implemented, which performs statistical analysis in order to rank suspected error locations. Then, an error matching approach to correction was applied implementing mutation operations. Localization of multiple erroneous data operations and their mutation-based correction was analyzed in the experiments. The two metrics of statistical analysis were compared and their capabilities in localizing multiple errors were shown.

As a result of the experiments it was discovered that the localization of two simultaneous errors by one of the metrics (metric B) is accurate and comparable to that of a single error localization. In some cases, the multiple error localization was even more accurate than in the case of single errors, which can be explained by the fact that secondary ranking criterion was used to refine the localization. Average correction times using mutation was just in fractions of seconds. Therefore statistical error diagnosis combined with mutation based error correction appears to be a feasible approach to automated debug of multiple design errors.

The Thesis presents a method for correcting design errors in algorithmic descriptions of system-level hardware. The method applies dynamic slicing and location ranking to accurately pinpoint the error locations and combines it with a dedicated set of ESL-mutation operators for automatically proposing fixes to the errors. In order to validate the approach, experiments on the Siemens benchmarks were carried out. The experiments show that the method is able to correct three times more errors than previously achievable by mutation-based error correction while examining fewer mutants. In addition, the method clearly outperforms a recent formal correction approach.

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). It was shown 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, it was shown 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.

Finally, the capability of TLM-derived mutation analysis of preserving the mapping between TLM and RTL mutants was elaborated. Thus, allowing to identify possible problems in the synthesis process more easily. 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 the original TLM functionality.

The chapter presents an approach to automatic localization of design errors (bugs) in processor designs. The approach is based on two main iterative phases: dynamic slicing and statistical suspiciousness ranking of the HDL statements in the design. The dynamic slicing reduces the debugging analysis to all the statements that actually affect the design's faulty behavior for a given stimuli. Then, the suspiciousness ranking assigns a suspiciousness score to each statement present in the dynamic slice.

The novelty of the approach is that it successfully in a scalable manner applies static slicing for analysis space reduction to realistic-size industrial designs and considers different coverage metrics for refining the bug localization. The approach is fault-model free and supports localization of multiple bugs. The original functional tests of processor designs can be used as a diagnostic test and is sufficient for the approach. However, quality diagnostic test can further increase the localization accuracy.

Last but not least, in this Thesis, a debug method for locating and correcting design errors at the source-level of hardware description language code using the design representation of high-level decision diagrams is presented. Experiments on a set of sequential register-transfer level benchmarks and one real-world design from the OpenCores repository show that the method is capable of locating the design errors injected with a high accuracy. Because of this localization accuracy the mutation-based correction requires a very small number of iterations and thus short run-times.

Conclusions

Conclusions

The main contribution of the Thesis is to propose new tools, case studies and methods to enable the designer automatically locate hard-to-detect bugs and offer solutions to save time and effort.

The specific contributions of the Thesis are divided into four main topics:

- RTL mutation analysis
- RTL and ESL mutation analysis comparison
- RTL localization and correction
- ESL localization and correction

RTL mutation analysis

The Thesis presented a new tool for mutation testing in hardware description languages using the system model of high-level decision diagrams (HLDD). The tool is integrated into the APRICOT verification environment. It is based on HLDD simulation and graph perturbation. A strategy that relies on a restricted set of five key mutation operators is developed in order to speed up the mutation analysis.

RTL and ESL mutation analysis comparison

The Thesis presented a method to automatically inject faults into the functionality of system descriptions that works at different abstraction levels (TLM and behavioral RTL). This is the first method for mutation analysis directly working on uncompiled SystemC TLM code.

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 chapter 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, it was shown 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.

Finally, the capability of TLM-derived mutation analysis of preserving the mapping between TLM and RTL mutants was elaborated. Thus, allowing to identify possible problems in the synthesis process more easily. 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 the original TLM functionality.

RTL localization and correction

In this Thesis, a debug method for locating and correcting design errors at the source-level of hardware description language code using the design representation of high-level decision diagrams is presented. Experiments on a set of sequential register-transfer level benchmarks and one real-world design from the OpenCores repository show that the method is capable of locating the design errors injected with a high accuracy. Because of this localization accuracy the mutation-based correction requires a very small number of iterations and thus short run-times.

The Thesis presents a case study of automatic localization of design errors (bugs) in processor designs. The approach is based on two main iterative phases: dynamic slicing and statistical suspiciousness ranking of the HDL statements in the design. The dynamic slicing reduces the debugging analysis to all the statements that actually affect the design's faulty behavior for a given stimuli. Then, the suspiciousness ranking assigns a suspiciousness score to each statement present in the dynamic slice.

The novelty of the approach is that it successfully in a scalable manner applies static slicing for analysis space reduction to realistic-size industrial designs and considers different coverage metrics for refining the bug localization. The approach is fault-model free and supports localization of multiple bugs. The original functional tests of processor designs can be used as a diagnostic test and it is sufficient for the approach. However, quality diagnostic test can further increase the localization accuracy.

ESL localization and correction

The Thesis presents a method for correcting design errors in algorithmic descriptions of system-level hardware. The method applies dynamic slicing and location ranking to accurately pinpoint the error locations and combines it with a dedicated set of ESL-mutation operators for automatically proposing fixes to the errors. In order to validate the approach, experiments on the Siemens benchmarks have been carried out. The experiments show that the method is able to repair three times more errors than previously achievable by mutation-based repair while examining fewer mutants. In addition, the method clearly outperforms a recent formal correction approach.

Future work

Future work includes an experimental study of real defects, a comparison with HDL mutation analysis and identification of equivalent mutants.

Additional plans include 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.

References

Abadir M. S., Ferguson J., Kirkland T. E. "Logic design verification via test generation". IEEE Transactions on Computer-Aided Design, Vol. 7, No. 1, 1988.

Abramovici M., Menon P. R., Miller D. T. “Critical path tracing - an alternative to fault simulation”. Proceedings of the 20th Design Automation Conference, 1983, pp. 214–220.

Agrawal H., DeMillo R. A., Hathaway B., Hsu W., Hsu W., Krauser E. W., Martin R. J., Mathur A. P., Spafford E. “Design of mutant operators for the C programming language”. Purdue University, West Lafayette, Indiana, Technical report SERC-TR-41-P, 1989.

Alexander R.T., Bieman J. M., Ghosh S., Bixia J. “Mutation of Java objects”. Proc. of IEEE ISSRE, 2002, pp. 341–351.

Ali M. F., Safarpour, S., Veneris, A., Abadir, M. S., Drechsler, R. “Post-verification debugging of hierarchical designs”. Proceedings of ICCAD Conference, 2005, pp. 871–876.

Belli F., Budnik C.-J., Wong W.-E. “Basic operations for generating behavioral mutants”. Proc. of IEEE ISSRE, 2006, pp. 10–18.

Bolchini C., Baresi L. "Software Methodologies in VHDL Code Analysis". Journal of Systems Architecture: the EUROMICRO Journal, Elsevier, Volume 44 , Issue 1, October 1997,pp. 3–21.

Bombieri N., Fummi F., Pravadelli G. “On the evaluation of transactor-based verification for reusing TLM assertions and testbenches at RTL”. Proc. of ACM/IEEE conference on design, automation and test in Europe (DATE), 2006, pp. 1007–1012.

Bombieri N., Fummi F., Pravadelli G . “A mutation model for the SystemC TLM 2.0 communication interfaces”. Proc. of ACM/IEEE conference on design, automation and test in Europe (DATE), 2008, pp. 396–401.

Bombieri N., Fummi F., Pravadelli G. “On the mutation analysis of SystemC TLM-2.0 standard”. Proceedings of IEEE international workshop on microprocessor test and verification (MTV), 2009, pp. 32–37.

Bombieri N., Fummi F., Pravadelli G., Hampton M., Letombe F. “Functional qualification of TLM verification”. Proc. of theACM/IEEE conference on design, automation and test in Europe (DATE), 2009, pp. 190–195.

Bradbury J. S., Cordy J. R., Dingel J. “Mutation operators for concurrent Java (J2SE 5.0)”. Proc. of IEEE ISSRE workshops, 2006, pp. 11–20.

Bryant R. E. “Graph-Based Algorithms for Boolean Function Manipulation”. IEEE Trans. on Computers, Vol. C-35, No. 8, 1986, pp. 677–691.

Budd T.A., Sayward F.G. “Users guide to the Pilot mutation system”. Yale University, New Haven, Connecticut, Technical report 114, 1977.

Certitude. [Online]
http://www.springsoft.com/products/functionalqualification/certitude, 2009.

Chang K.-H., Wagner I., Bertacco V., Markov I. "Automatic Error Diagnosis and Correction for RTL Designs". Proceedings of the High-Level Design and Validation Workshop (HLDVT), 2007.

Chang K.-H., Markov I. L.; Bertacco V. "Fixing Design Errors With Counterexamples and Resynthesis". IEEE Trans. on CAD of ICs and Systems, vol.27, no.1, January 2008, pp.184–188.

Chayakul V., Gajski D. D., Ramachandran L. “High-Level Transformations for Minimizing Syntactic Variances”. Proc. of ACM/IEEE DAC, June 1993, pp. 413–418.

Choi B.J., DeMillo R.A., Krauser E.W., Martin R.J., Mathur A.P., Offutt A.J., Pan H., Spafford E.H. “The Mothra tool set”. Proceedings of the 22nd annual Hawaii international conference on system sciences (HICSS), 1989, pp 275–284.

Clarke E., Fujita M., McGeer P., McMillan K.L., Yang J., Zhao X. „Multi terminal BDDs: an efficient data structure for matrix representation“. Proc. of Int’l Workshop on Logic Synth., 1993, pp. P6a: 1–15.

Clarke E. M., Fujita M., Rajan S. P., Reps T., Shankar S., Teitelbaum T. “Program slicing for VHDL”. In Charme99, Bad Herrenalb, Germany, September 1999.

Cleve H., Zeller A. “Locating causes of program failures”. Proc. of Int. Conf. on Software Engineering, 2005, pp. 342–351.

CNN Money. [Online] http://money.cnn.com/2014/09/30/technology/security/internet-bug/index.html, 2014.

Computer History Museum. [Online] http://www.computerhistory.org/revolution/digital-logic/12/285, 2015.

Debroy V, Wong W. E. "Using Mutation to Automatically Suggest Fixes for Faulty Programs". Proceedings of the Third International Conference on Software Testing, Verification and Validation, 2010, pp. 65–74.

DeMillo R. A., Guindi D. S., McCracken W. M., Offutt A. J., King K. N. "An extended overview of the Mothra software testing environment". Second Workshop on Software Testing, Verification, and Analysis, July 1988, pp. 142–151.

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

De Moura L., Bjorner N. “Satisfiability modulo theories: An appetizer”. Formal Methods: Foundations and Applications, 2009, pp. 23–36.

DIAMOND project website. [Online] http://www.fp7-diamond.eu/, 2011.

Dowson M. "The Ariane 5 Software Failure". Software Engineering Notes 22 (2): 84, March 1997.

Drechsler R., Becker B., Ruppertz S. „K*BMDs: a new data structure for verification“. Proc. of European Design & Test Conf., 1996, pp. 2–8.

Fey G., Staber S., Bloem R., Drechsler R. “Automatic Fault Localization for Property Checking”. IEEE Transactions on CAD of Integrated Circuits and Systems, 27(6), 2008, pp. 1138–1149.

FP6 PROSYD (Property-Based System Design), FP6 funded STREP. [Online] http://www.prosyd.org/, 2004.

Grötker T., Martin G., Liao S., Swan S. “System Design with SystemC”. Kluwer Academic Publishers, 2002.

Guarnieri V., Bombieri N., Pravadelli G., Fummi F., Hantson H., Raik J., Jenihhin M., Ubar R. “Mutation analysis for SystemC designs at TLM”. Proc. of IEEE Latin-American Test Workshop (LATW), 2011, pp. 27–30.

Hantson H., Raik J., Jenihhin M., Chepurov A., Ubar R., di Guglielmo G., Fummi F. “Mutation analysis with high-level decision diagrams”. IEEE Latin-American Test workshop (LATW), 2010, pp. 1–6.

Hamlet R. G. “Testing programs with the aid of a compiler”. IEEE Treans Softw Eng 3(4), 1977, pp.279–290.

Harris I. G. "A Coverage Metric for the Validation of Interacting Processes". Proceedings of the conference on Design, Automation and Test in Europe (DATE), 2006, pp. 1019–1024.

Hayek, G. Robach C. "From Specification Validation to Hardware Testing: A Unified Method". Proceedings of the IEEE International Test Conference, 1996, pp. 885–893.

Irvine SA et al. “Jumble Java byte code to measure the effectiveness of unit tests”. Mutation testing workshop, 2007, pp. 169–175.

International Test Conference 1999 (ITC99) benchmarks. [Online] http://www.cerc.utexas.edu/itc99-benchmarks/bench.html, 2009.

International Technology Roadmap for Semiconductors (ITRS). [Online] http://www.itrs.net/reports.html, 2013.

Jones J. A., Harrold M. J. “Empirical evaluation of the Tarantula automatic fault-localization technique”. Proc. of Int. Conf. on Automated Software Engineering, 2005, pp. 273–283.

Juniper Research. “‘Internet of Things’ Connected Devices to Almost Triple to Over 38 Billion Units by 2020”. [Online] http://www.juniperresearch.com/press/press-releases/iot-connected-devices-to-triple-to-38-bn-by-2020, 2015.

Jutman A. “Design error diagnosis in digital circuits”. Master thesis, Tallinn University of Technology, 1999.

Kašjanov V. N. “Distinguishing Hammocks in a Directed Graph”. Soviet Math. Doklady, vol. 16, no. 5, 1975, pp. 448–450.

Keutzer K., Newton R. Sematech. [Online]
http://public.sematech.org/Pages/home.aspx, 2015.

Kim C.-R. “Toyota to recall 1.9 million Prius cars for software defect in hybrid system”. Reuters. [Online] http://www.reuters.com/article/2014/02/12/us-toyota-recall-idUSBREA1B1B920140212, 2014.

Korel B., Laski J. “Dynamic program slicing”. Information Processing Letters, vol. 29, no. 3, 1988, pp. 155–163.

Könighofer R., Bloem R. "Automated Error Localization and Correction for Imperative Programs". Proceedings of 11th International Conference of Formal Methods in Computer Aided Design (FMCAD), 2011, pp. 91–100.

Lam W. K. “Hardware Design Verification: Simulation and Formal Method-Based Approaches”. Prentice Hall PTR, 2005.

Leveson, N. “Therac-25 Accidents: An Updated Version of the Original Accident Investigation Paper”. Software, System Safety, and Computers, Addison Wesley. [Online]
http://www.cs.washington.edu/research/projects/safety/www/therac-25.html, 1995

Liblit B., Naik M., Zheng A. X., Aiken A., Jordan M. I. “Scalable statistical bug isolation”. ACM SIGPLAN Notices, vol. 40, no. 6, 2005, pp. 15–26.

Lions J. L. “Ariane 5, Flight 501 Failure”. Inquiry Board report. [Online]
https://www.ima.umn.edu/~arnold/disasters/ariane5rep.html, 1996.

Lipton R. “Fault diagnosis of computer programs”. Carnegie Mellon University, Student report, 1971.

Lisherness P., Cheng K.-T. (Tim). “SCEMIT: a SystemC error and mutation injection tool”. Proc. of ACM/IEEE design automation conference (DAC), 2010, pp. 228–233.

Liu G., Fei L., Yan X., Han J., Midkiff S. P. “Statistical debugging: A hypothesis testing-based approach”. IEEE Trans. on Software Engineering, vol. 32, no. 10, 2006, pp. 831–848.

Lyu M.-R., Zubin H., Sze S. K. S., Xia C. “An empirical study on testing and fault tolerance for software reliability engineering”. Proc. of IEEE ISSRE, 2003, pp. 119–130.

Ma Y.-S,, Offutt A. J., Kwon Y. R. “MuJava: an automated class mutation system: research articles”. Software Test Verif Reliab 15, 2005, pp. 97–133.

Madre J. C., Coudert O., Billon J. P. "Automating the Diagnosis and the Rectification of Design Errors with PRIAM". Proceedings of ICCAD Conference, 1989, pp. 30–33.

Manning S., Krisher T. “Toyota woes mount as gov't examines Prius brakes”. Associated Press Writers. [Online] http://web.archive.org/web/20100206043650/http://news.yahoo.com/s/ap/20100204/ap_on_bi_ge/toyota_recall, 2010.

Mentor Graphics. “Catapult C Synthesis - Full-Chip High-Level Synthesis”. [Online] http://www.mentor.com/esl/catapult/, 2010.

Mettler F.A. Jr., Ortiz López P., et al. “Investigation of an accidental exposure of radiotherapy patients in Panama”. Report of a Team of Experts, International Atomic Energy Agency. [Online] http://www-pub.iaea.org/MTCD/publications/PDF/Pub1114_scr.pdf, 2001.

Meza-Escobar J.H., Sachsse J., Ostendorff S., Wuttke H. D. “Automatic generation of an FPGA based embedded test system for printed circuit board testing” Proc. LATW2012, 2012, pp. 75–80.

Nicely T. “Pentium FDIV flaw FAQ”. [Online] http://www.trnicely.net/pentbug/pentbug.html, 2011

Offut A. J., Untch R. H. “Mutation 2000: Uniting the Orthogonal”. „Mutation testing for the new century”. Kluwer Academic Publishers Norwell, 2001, pp. 34–44.

Offutt A. J., Rothermel G., Zapf C. "An experimental evaluation of selective mutation". Proceedings of the IEEE Fifteenth International Conference on Software Engineering, May 1993, pp. 100–107.

Offutt A. J., King K. N. “A Fortran 77 interpreter for mutation analysis”. Papers of the symposium on interpreters and interpretive techniques. SIGPLAN, 1987, pp. 177–188.

OpenCores design repository. [Online] http://www.opencores.org/, 2012.

Open SystemC Initiative (OSCI). [Online] http://www.systemc.org, 2009.

Open SystemC Initiative Transaction-Level Modeling (OSCI TLM-2.0) Language Reference Manual. [Online] http://www.systemc.org, 2009.

Raik J., Repinski U., Hantson H., Jenihhin M., Di Guglielmo G., Pravadelli G., Fummi F. "Combining Dynamic Slicing and Mutation Operators for ESL Correction". 17th IEEE European Test Symposium, IEEE Computer Society Press, 2012, pp. 1–6.

Raik, J.; Repinski, U.; Tšepurov, A.; Hantson, H.; Ubar, R.; Jenihhin, M. "Automated design error debug using high-level decision diagrams and mutation operators". Microprocessors and Microsystems: Embedded Hardware Design, 37(4), 2013, pp. 1–10.

Raik J., Ubar R. "Fast Test Pattern Generation for Sequential Circuits Using Decision Diagram Representations". JETTA, Kluwer Academic Publishers. Vol. 16, No. 3, June 2000, pp. 213–226.

Sachsse J., Ostendorff S., Wuttke H. D., Meza-Escobar J. H. "Architecture of an adaptive Test System built on FPGA" Proc. Architecture of Computing Systems (ARCS), vol. LNCS 6566, 2011, pp. 86–97.

Schaller, R. R. "Moore's law: past, present and future" IEEE Spectrum, Volume 34, Issue 6, 1997, pp. 52–59.

Sen A. "Mutation operators for concurrent SystemC designs". Proc. of IEEE international workshop on microprocessor test and verification (MTV), 2009, pp. 27–31.

Sen A., Abadir M. S. "Coverage metrics for verification of concurrent SystemC designs using mutation testing". Proc. of IEEE international high-level design, validation and test workshop, 2010, pp. 75–81.

Siemens benchmark suite. [Online] http://pleuma.cc.gatech.edu/aristotle/Tools/subjects/, 2010.

Smith A., Veneris A., Viglas A. "Design Diagnosis Using Boolean Satisfiability". Proc. Asia and South Pacific Design Automation Conference (ASPDAC), 2004, pp. 218–223.

Staber S., Jobstmann B., Bloem R. “Finding and fixing faults”. Proc. of Conference on Correct Hardware Design and Verification Methods, 2005, pp. 35–49.

Stallman R. M., Pesch R. H. “Using GDB: A guide to the GNU source-level debugger”. Free Software Foundation, 1991.

Tasiran S., Keutzer K. "Coverage Metrics for Functional Validation of Hardware Designs". IEEE Design & Test, Volume 18, Issue 4, July 2001, pp. 36–45.

Tao L., Jian-Ping F., Xiao-Wei L., Ling-Yi L. "Observability Statement Coverage Based on Dynamic Factored Use-Definition Chains for Functional Verification". Journal of Electronic Testing: Theory and Applications, Springer, Volume 22, Issue 3, 2006, pp. 273–285.

Tšepurov A., Bartsch G., Dorsch R., Jenihhin M., Raik J., Tihhomirov V. “A Scalable Model Based RTL Framework zamiaCAD for Static Analysis”. IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC), October 2012.

Tuya J., Suarez-Cabal M. J., De La Riva C. “SQLMutation: a tool to generate mutants of SQL database queries”. Mutation testing workshop, 2006, pp. 39–43.

Ubar R., Morawiec A., Raik J. “Back-Tracing and Event-Driven Techniques in High-Level Simulation with Decision Diagrams”. Proc. of the IEEE ISCAS'2000 Conference, Vol. 1, 2000, pp. 208–211.

Ubar R, Raik J, Vierhaus H. T. “Design and Test Technology for Dependable Systems-on-Chip”. Hershey - New York, Information Science Reference, IGI Global, 2011.

Vessey I. “Expertise in debugging computer programs”. International Journal of Man-Machine Studies: A Process Analysis, vol. 23, no. 5, 1985, pp. 459–494.

Vertigo project. [Online] http://www.vertigo-project.eu, 2009.

Wahba A., Borrione D. “Design error diagnosis in sequential circuits”. Lecture Notes In Computer Science, Springer, Vol. 987, 1995, pp. 171–188.

Warnock, J., et al. “4.1 22nm Next-generation IBM System z microprocessor”. IEEE International Solid- State Circuits Conference (ISSCC), 2015.

Weiser M. "Program slicing". Proceedings of the 5th International Conference on Software Engineering, IEEE Computer Society Press, 1981, pp. 439–449.

Weiser M. “Program slicing” IEEE Transactions on Software Engineering, vol. 10, no. 4, 1984, pp. 352–357.

Wong W. E., Debroy V., Choi B. “A family of code coverage-based heuristics for effective fault localization”. Journal of Systems and Software, vol.83, no.2, 2010, pp. 188–208.

Wong W. E., Qi Y. “BP neural network-based effective fault localization”. International Journal of Software Engineering and Knowledge Engineering, vol. 19, no. 4, 2009, pp. 573–597.

Abstract

Modern society is very dependent on electronics. At the same time, devices have become more and more complex. Point, where verifying device correctness has become more complicated than designing them, is reached. In verification locating and fixing the bugs requires more effort than identifying them. Automating bug localization and correction is the topic that the current Thesis focuses on. Solutions are divided into two major fields.

First, verification of the designs by applying mutation analysis is addressed at two abstraction levels: Register-Transfer Level (RTL) and Electronic System Level (ESL).

On High-Level Decision Diagrams (HLDD) model at the RTL mutation analysis is applied and a new tool is implemented. The tool is integrated into the APRICOT verification environment. It is based on HLDD simulation and graph perturbation. A strategy that relies on a restricted set of five key mutation operators is developed in order to speed up the mutation analysis.

This is followed by a new method to automatically inject faults into the functionality of system descriptions that works at different abstraction levels (TLM and behavioral RTL). The results of injecting mutants directly on the RTL models (native RTL mutation analysis) and injecting mutants on the TLM descriptions and then synthesizing the corresponding RTL mutated models (TLM-derived mutation analysis) are compared.

Second, the focus is on design error localization and correction, which is presented in the two above mentioned abstraction levels.

At RTL a debug method for locating and correcting design errors at the source-level of hardware description language code using the design representation of HLDDs is presented and implemented. Additionally a case study of automatic localization of design errors (bugs) in industrial processor ROBSY is presented. The approach is based on two main iterative phases: dynamic slicing and statistical suspiciousness ranking of the HDL statements in the design.

Finally, a method for correcting design errors in algorithmic descriptions of system-level hardware is presented. In the experiments simple C programs are used as benchmarks. The method applies dynamic slicing and location ranking to accurately pinpoint the error locations and combines it with a dedicated set of ESL-mutation operators for automatically proposing fixes to the errors. In order

to validate the approach, experiments on the Siemens benchmarks have been carried out.

The main contribution of the Thesis is to propose new tools, case studies and methods enabling the integrated circuit designer automatically locate hard-to-detect bugs and offer solutions to save time and effort.

Annotatsioon

Tohutult kiire tehnoloogia areng on viinud meid ajastusse, kus arvutid ja elektroonika on osa peaaegu kõigest. Samal ajal on seadmete tehnoloogiline keerukus järjest kasvanud. Oleme saavutanud olukorra, kus seadmete õigsuse kontroll ehk verifitseerimine nõuab rohkem aega ja vaeva kui nende väljatöötamine. Verifitseerimisel on vigade leidmine ning parandamine aeganõudvamad kui nende olemasolu tuvastamine. Automatiseeritult vigade leidmise ja parandamise teemale käesolev doktoritöö keskendubki. Pakutavad lahendused jagunevad kaheks.

Esmalt analüüsitakse verifitseerimist kahel abstraktsiooni tasemel: registersiirde- (RTL) ja süsteemitasemel (ESL).

Kõrgtaseme otsustusdiagrammide (HLDD) mudelil RTL tasemel kasutatakse mutatsioonianalüüsi ja luuakse uus tarkvaraline tööriist. Tööriist on integreeritud APRICOT verifitseerimise raamistikku. See põhineb HLDD simulatsioonil ja muudatustel graafide struktuuris. Lähenemise võtmeks on viis mutatsioonioperaatorit, mis aitavad mutatsioonianalüüsi kiirendada.

Järgneb uue meetodi kirjeldus vigade automaatseks sisestamiseks süsteemi funktsionaalsusesse, mis töötab erinevatel abstraktsioonitasemetel (TLM ja käitumuslik RTL). Võrreldakse mutatsioonide kasutamise tulemusi otse RTL mudelil ja TLM kirjeldustel, mis sünteesitakse vastavate RTL mudelite põhjal.

Seejärel keskendutakse vigade lokaliseerimisele ja kõrvaldamisele ning seda tehakse kahel eelpool nimetatud abstraktsiooni tasemel.

RTL tasemel esitatakse ja realiseeritakse vigade lokaliseerimine ja parandamine riistvara kirjelduskeele lähtekoodi tasemel kasutades mudelina HLDDsid. Lisaks viiakse läbi juhtumiuuring tööstuslikus mikroprotsessoris ROBSY kasutades automatiseeritud vigade lokaliseerimist. Lähenemine põhineb kahel iteratiivsel etapil: dünaamiline viilutamine ja statistilisel analüüsil pingerea koostamine võimalikest vigade asukohtadest disainis.

Viimasena esitatakse mutatsioonidel põhinev disainivigade parendamise meetod süsteemitaseme riistvara algoritmilistele kirjeldustele. Eksperimentide hindamisel kasutatakse C keeles kirjutatud programme. Meetod kasutab dünaamilist viilutamist ja veakandidaatide pingerida leidmaks tõenäolisi vigade asukohti. Võimalike lahenduste pakkumisel kasutatakse kindlat kogumit

kõrgtaseme mutatsiooni operaatoreid. Lahenduse tulemuste hindamiseks kasutatakse eksperimentide läbiviimisel Siemens võrdlusprogramme.

Dissertatsiooni peamiseks panuseks on uute, automatiseeritud töövahendite loomine, juhtumiuuring ja meetodid võimaldamaks riistvara projekteerijal säästa aega raskesti tuvastatavate vigade leidmisel.

Acknowledgements

I would like to express my extreme gratitude to my supervisor, Professor Jaan Raik, who has been with me all those years regardless of the situation.

Additionally I would like to acknowledge everyone from Tallinn University of Technology and from the University of Verona who have participated in our joint projects. In particular FP6 Vertigo, FP7 REGPOT Credes, FP7 Diamond and Estonian ICT Doctoral School.

Finally I would like to appreciate my family and friends who have supported and encouraged me throughout my studies.

Thank you all!

Hanno Hantson

Tallinn, September 2015

Curriculum Vitae

Personal data

Name: Hanno Hantson

Date and place of birth: 13.04.1982, Estonia

Nationality: Estonian

Contact information

Address: Raja 15, 12618, Tallinn

Telephone: +372 620 2257

E-mail address: hannoh@gmail.com

Education

2006 –	Ph.D. student in Department of Computer Engineering, Tallinn University of Technology (TUT)
2004 – 2006	M.Sc. in Computer Engineering, TUT
2000 – 2004	Diploma in Computer Engineering, TUT

Career

2010 –	Estonian Academy of Security Sciences, ICT manager
2009 – 2010	IT and Development Centre of the Ministry of the Interior, system administrator
2007 – 2009	Estonian Academy of Security Sciences, system administrator
2005 – 2007	ELIKO Ltd, engineer
2001 – 2005	Girf Ltd, programmer

Scientific activities

2010 – 2012	Member of IEEE Computer Society

Defended theses

Hanno Hantson (2006). Comparison of DFT tools. Master of Science degree. Tallinn University of Technology, Department of Computer Engineering. Supervisor: Jaan Raik.

Main areas of scientific work/current research topics

Mutation analysis, design error localization and correction.

Elulookirjeldus

Isikuandmed

Ees- ja perekonnanimi: Hanno Hantson

Sünniaeg ja -koht: 13.04.1982, Eesti

Kodakondsus: eestlane

Kontaktandmed

Aadress: Raja 15, 12618, Tallinn

Telefon: +372 620 2257

E-posti aadress: hannoh@gmail.com

Hariduskäik

2006 –	doktorant, Arvutitehnika instituut, Tallinna Tehnikaülikool (TTÜ)
2004 – 2006	tehnikateaduste magister, arvuti- ja süsteemitehnika eriala, TTÜ
2000 – 2004	diplom, arvutisüsteemide eriala, TTÜ

Teenistuskäik

2010 –	Sisekaitseakadeemia, IKT juht
2009 – 2010	Siseministeeriumi infotehnoloogia- ja arenduskeskus, süsteemiadministraator
2007 – 2009	Sisekaitseakadeemia, süsteemiadministraator
2005 – 2007	ELIKO OÜ, arendusinsener
2001 – 2005	Girf OÜ, programmeerija

Teadustegevus

2010 – 2012	IEEE Computer Society. Liige

Kaitstud lõputööd

Hanno Hantson (2006). Testitava projekteerimise vahendite võrdlev analüüs. Teadusmagistri kraad. Tallinna Tehnikaülikool, Arvutitehnika instituut. Juhendaja: Jaan Raik.

Teadustöö põhisuunad

Mutatsioonianalüüs, projekteerimisvigade lokaliseerimine ja parandamine.

Appendix I

Research paper I

Hantson, Hanno; Raik, Jaan; di Guglielmo, Giuseppe; Jenihhin, Maksim; Chepurov, Anton; Fummi, Franco; Ubar, Raimund. "Mutation Analysis with High-Level Decision Diagrams". Proceedings of the 11th Latin-American Test Workshop, IEEE Computer Society Press, 2010, pp. 1–6.

Contributes to Section 3.1 of this Thesis. The author's contributions are: participating in development of the HLDD-based mutation analysis method, implementing mutation analysis tool to the Apricot framework and presenting the paper at 11th Latin-American Test Workshop.

Mutation Analysis with High-Level Decision Diagrams

Hanno Hantson, Jaan Raik, Maksim Jenihhin, Anton Chepurov, Raimund Ubar
Department of Computer Engineering
Tallinn University of Technology
Tallinn, Estonia
{hanno|jaan|maksim}@ati.ttu.ee

Giuseppe di Guglielmo, Franco Fummi
Department of Computer Science
University of Verona
Verona, Italy
{giuseppe.diguglielmo|franco.fummi}@univr.it

Abstract – **The paper presents a new tool for mutation analysis using the system model of high-level decision diagrams (HLDD). The tool is integrated into the APRICOT verification environment. It is based on HLDD simulation and graph perturbation. A strategy that relies on a restricted set of five key mutation operators is developed in order to speed up the mutation analysis. Experiments on several ITC99 benchmarks and an industrial example show the feasibility of the mutation analysis approach.**

Keywords – mutation analysis; decision diagrams;

I. INTRODUCTION

Mutation analysis and mutation testing have gained importance during the last decades as being important techniques for software testing. Such testing approaches rely on the creation of several *mutated* versions of the program to be tested. Mutation is carried out by introducing syntactically correct functional 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. The effectiveness of the test suite is then measured by computing the percentage of detected, or *killed*, mutations.

In order for a functional verification or testing method to detect bugs in the program, 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.

Traditional verification methods suffer from the observability problem as they focus on the first point only. Techniques such as code coverage and functional coverage cannot guarantee that design bugs will be propagated. On the contrary, mutation testing guarantees observation and is thus more powerful in terms of bug detection capabilities.

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

Mutation analysis was first proposed in 1971, when Richard Lipton introduced the initial concepts of mutation in [1]. However, major work was not published until the end of 1970s [2],[3],[4]. PIMS [4], an early mutation testing tool, pioneered the general process typically used in mutation testing of creating mutants, accepting test cases from the users, and then executing the test cases on the mutants to decide how many mutants were killed. Afterwards, several tools for mutation testing have been developed, most widely known of them being probably the Mothra system [5] running on Fortran programs.

The observability problem of traditional coverage methods is widely analyzed in [10]. In particular the authors present an observability model and an algorithm to evaluate observability-based statement coverage for hardware designs. As in [9], it is clearly stated that hardware designs are highly concurrent, while code software coverage metrics do not address this essential characteristic. Hence it is far from sufficient to achieve complete code coverage during verification [11].

Despite of being originally a software testing technique, obvious similarities with procedural programming languages suggested tailoring some software analysis techniques to hardware description language (HDL) behavioral description analysis [6]. In particular, an adaptation of the mutation analysis to test VHDL functional descriptions is proposed in [7]. A VHDL language functional description can be assimilated to a software program, so it can be validated against (software) design faults using the mutation testing techniques. The methodology covers VHDL concurrent statements as block statement, process statement, and concurrent signal assignment statement. The VHDL code is translated into Fortran, and Mothra [5] is applied to generate test sequences. In the proposed approach, however, concurrent constructs are merely translated to a sequential language and not targeted explicitly. In addition to academic attempts to bring mutation testing into hardware domain, a commercial functional qualification tool Certitude [8] based on mutation analysis is available from Springsoft.

978-1-4244-7785-2/10/$26.00©2010

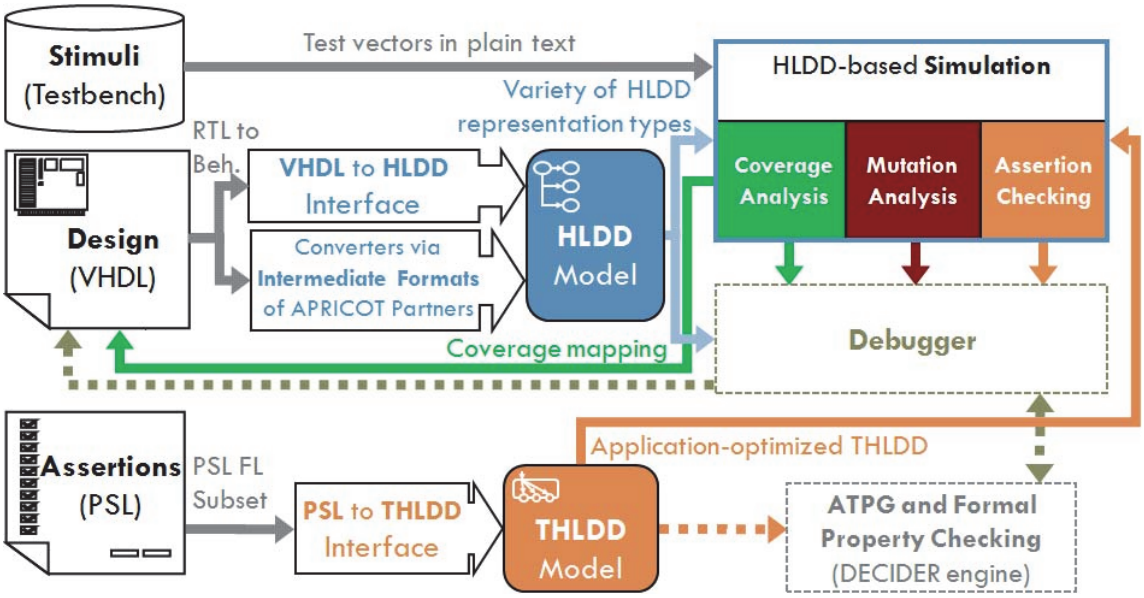


Figure 1. Hardware verification framework APRICOT

The adopted HLDD model provides fast simulation. Very efficient HLDD based simulation algorithms have been proposed which outperform commercial event-driven HDL simulators in 12 - 30 times and cycle-based simulators in 4 to 6 times [12]. This is due to the fact that HLDD simulation essentially combines event-driven (path activation in the HLDD graphs) and cycle-based (HLDDs are synthesized into cycle-accurate models) paradigms.

To the best of the authors' knowledge this is the first attempt to solve mutation analysis on the high-level decision diagram model. We show on an industrial example that high-quality tests receiving near-hundred-percent code coverage result only in 21 % mutation coverage. This indicates a clear advantage of the mutation testing over the coverage approach, due to considering fault observation.

The paper is organized as follows. Section 2 describes the integrated environment for verification and mutation analysis. Section 3 defines the High-Level Decision Diagram (HLDD) model. Section 4 explains simulation on HLDDs. In Section 5, the HLDD-based mutation testing environment is presented. Section 6 provides experimental results. Finally, conclusions and future work are given.

II. Integrated Environment for Verification and Mutation Analysis

APRICOT is a functional verification framework developed at Tallinn University of Technology [18]. APRICOT is an acronym for *A*ssertions checking (monitoring), formal *PR*operty check*I*ng, verification *CO*verage analysis and *T*est pattern generation. As it follows from its name, the framework supports a wide range of verification tasks.

The novelty of APRICOT lies in the usage of high-level decision diagrams for design representation. Development of the APRICOT framework has been started in order to target aspects of *speed*, *accuracy*, *complexity* and *diagnosability* of hardware functional verification. The novelty of the framework lies in taking advantages of design under verification representation by *High-Level Decision Diagrams* (HLDD) model [19]. Our previous works [20], [21] have shown that HLDDs are an efficient model for simulation and convenient for diagnosis and debug. In this paper we integrate mutation analysis functionality to the system. The structure of the framework is shown in Fig. 1.

Data flow of the HLDD-based mutation analysis environment is presented in Fig. 2. The analysis starts with test stimuli and the HLDD model generated automatically from the VHDL language description of the design. (Note, that automatic HLDD generation from VHDL is a very fast process, as shown in Section 5). Subsequently, HLDD simulation is performed according to Algorithm 1. This is followed by mutation analysis on HLDD models (Algorithm 2). As a result of the analysis there may remain live mutants. Tests will be generated for them, either manually or by an automated tool (the striped boxes in Fig. 2). However, the mutation test generation is out of the scope of this paper. An automated test pattern generator working on HLDD models has been presented in [15]. It is planned to extend it to mutation testing as one of the next developments in the system.

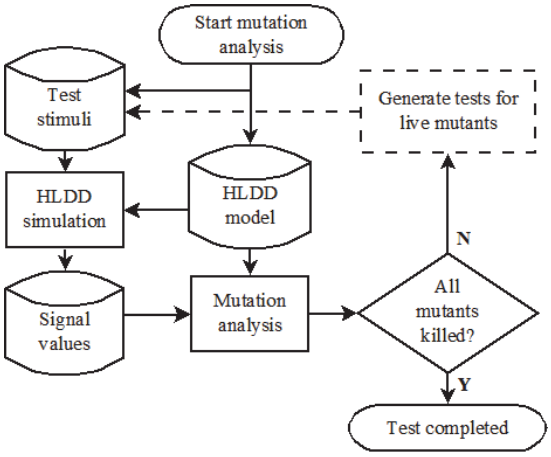


Figure 2. Data flow of the HLDD-based mutation analysis environment

III. HIGH-LEVEL DECISION DIAGRAMS

In the following we give a formal definition of High-Level Decision Diagrams (HLDD). Let us denote a discrete function $y = f(x)$, where $y = (y_1, \ldots, y_n)$ and $x = (x_1, \ldots, x_m)$ are vectors defined on $X = X_1 \times \ldots \times X_m$ with values $y \in Y = Y_1 \times \ldots \times Y_n$, and both, the domain X and the range Y are finite sets of values. The values of variables may be Boolean, Boolean vectors or integers.

Definition 1. A HLDD representing a discrete function $y=f(x)$ is a directed acyclic labeled graph that can be defined as a quadruple $G_y=(M,E,Z,\Gamma)$, where M is a finite set of vertices (referred to as *nodes*), E is a finite set of *edges*, Z is a function, which defines the *variables labeling the nodes*, and Γ is a function on E. The function $Z(m_i)$ returns the variable x_k, which is labeling node m_i. Each node of a HLDD is labeled by a variable. In special cases, nodes can be labeled by constants or algebraic expressions. An edge $e \in E$ of a HLDD is an ordered pair $e=(m_1,m_2) \in E^2$, where E^2 is the set of all the possible ordered pairs in set E. Γ is a function on E representing the activating conditions of the edges for the simulating procedures. The value of $\Gamma(e)$ is a subset of the domain X_k of the variable x_k, where $e=(m_i,m_j)$ and $Z(m_i)=x_k$. It is required that $Pm_i=\{ \Gamma(e) \mid e = (m_i,m_j) \in E\}$ is a partition of the set X_k. Fig. 3 presents a HLDD for a discrete function $y_{inst}=f(x_1,x_2,x_3,x_4)$. HLDD has only one starting node (root node) m_0, for which there are no preceding nodes. The nodes that have no successor nodes are referred to as terminal nodes $M^{term} \in M$.

HLDD models can be used for representing digital systems. In such models, the non-terminal nodes correspond to conditions or to control signals, and the terminal nodes represent data operations, variables or constants. When representing systems by decision diagram models, in general case, a network of HLDDs rather than a single HLDD is required. During the simulation in HLDD systems, the values of some variables labeling the nodes of a HLDD are calculated by other HLDDs of the system.

Different from the well-known Reduced Ordered BDD models which have worst-case exponential space requirements, HLDD size scales well with respect to the size of the RTL code. The main difference is that traditionally decision diagram is generated for a primary output of the system while nodes represent primary inputs. In HLDDs we generate a separate diagram for each variable (signal) v of the VHDL description and nodes represent variables (signals) assigned to v. Note, that the complexity of HLDDs is just $O(n)$ with respect to the number of processes in the code. Thus, very large realistic hardware systems can be represented in practice. An example of HLDD representation of VHDL is presented in Fig. 4 [13].

IV. SIMULATION USING HLDDS

Algorithm 1 presents the simulation method for HLDDs.

Algorithm 1. HLDD-based simulation

```
SimulateHLDD()
  For each diagram G in the model
    m_Current = m_0
    Let x_Current be the variable labeling m_Current
    While m_Current is not a terminal node
      If x_Current is clocked or its DD is ranked after G then
        Value = previous time-step value of x_Current
      Else
        Value = present time-step value of x_Current
      End if
      For {Γ | Value ∈ Γ(e_active), e_active =( m_Current, m_Next)}
        m_Current = m_Next
      End for
    End while
    Assign x_G = x_Current
  End for
End SimulateHLDD
```

In the RTL style, the algorithm takes the previous time step value of variable x_j labeling a node m_i if x_j represents a clocked variable in the corresponding HDL. In the behavioral style, the present value of x_j will be used. In the case of behavioral HDL coding style HLDDs are generated and ranked in a specific order to ensure causality. For variables x_j labeling HLDD nodes the previous time step value is used if the HLDD calculating x_j is ranked after current decision diagram. Otherwise, the present time step value will be used.

Let us explain the HLDD simulation process on the decision diagram example presented in Fig. 3. Assuming that variable x_4 is equal to 3, a path is activated from node m_0 (the root node) to a terminal node m_4 labeled by x_3. Let the value of variable x_3 be 4, thus, $y_{inst}=x_3=4$. Note, that this type of simulation is event-driven since we have to simulate only those nodes that are traversed by the main activated path.

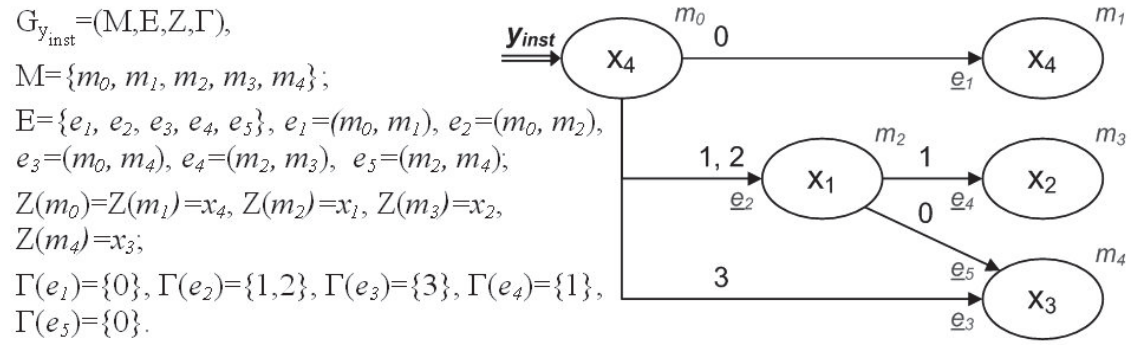


Figure 3. A high-level decision diagram representing a function $y_{inst}=f(x_1,x_2,x_3,x_4)$

V. MUTATION ANALYSIS AS HLDD PERTURBATIONS

The method presented in this paper is based on strong mutation. Being the first work on HLDD-based mutation testing the goal of this paper was not to propose new mutation operators, nor to find the most optimal set of mutation operators. Instead the five *key* operators proposed in [14] have been implemented according to the *do fewer* strategy. In experiments, those five operators 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 [14]. The 5 sufficient operators are ABS, which forces each arithmetic expression to take on the value 0, a positive value, and a negative value, AOR, which replaces each arithmetic operator with every syntactically legal operator, LCR, which replaces each logical connector with several kinds of logical connectors, ROR, which replaces relational operators with other relational operators, and UOI, which inserts unary operators in front of expressions.

We have implemented the five operators with the following constraints and specifics. UOI currently replaces only unary operators with other unary operators and ABS is applied to variables only, and not to expressions. Note also that in HLDD there are no signed/unsigned variables, but signed and unsigned relational operators exist. Therefore ROR replaces, both, signed and unsigned relational operators. In AOR we also allow mutation by division and mod operations and we have included a check for the case of divide-by-zero. In the future, our goal is to gradually extend the set of mutation operators and select the most optimal set for hardware programs. The reduced-5-key-operator strategy represents a do fewer strategy. The purpose would be to reduce the mutation analysis cost as much as possible.

Fig. 5 illustrates the HLDD graph perturbations for implementing the five key mutation operators on a sample diagram G_{y_out}. In HLDD models, the perturbation means simply replacement of an operator, variable or constant labeling the HLDD node by another operator, variable or constant.

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

TABLE I. MUTATION OPERATORS

Mutation operator	List of replacements
LCR (logical connector replacement)	AND, NAND, OR, NOR, XOR
AOR (arithmetic operator replacement)	ADDER, SUBTR, MULT, DIV, MOD
UOI (unary operation insertion)	NEG, INV
SOR (shift operator replacement)	SHIFT_LEFT, SHIFT_RIGHT, U_SHIFT_RIGHT
ROR (relational operation replacement)	EQ, NEQ, GT, LT, GE, LE, U_GT, U_LT, U_GE, U_LE

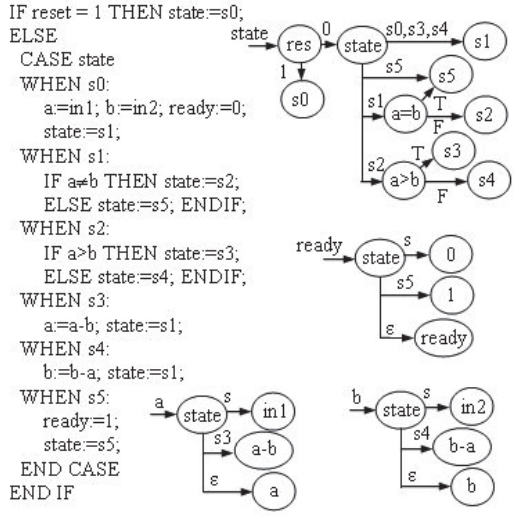


Figure 4. a) RTL VHDL and b) the corresponding HLDD

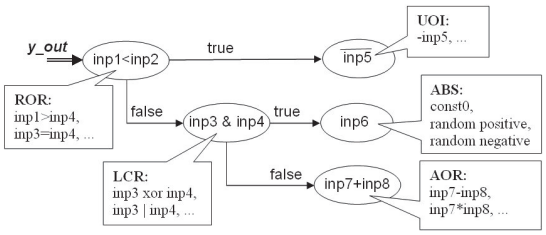


Figure 5. "Key" mutation operators as HLDD perturbations

TABLE II. VHDL CODE CHARACTERISTICS OF THE ITC99 BENCHMARKS AND THE CRC DESIGN

Design	Code lines #	Inputs #	Outputs #	Registers #	Processes #
b01	96	4	2	3	1
b02	61	3	1	2	1
b04	76	6	1	9	1
b06	112	4	4	5	1
b09	81	3	1	5	1
b11	107	4	1	5	1
b13	273	5	7	24	5
CRC	371	10	3	11	9

Algorithm 2 presents the Mutation Analysis (MA) algorithm on HLDD representations. The MA process starts with HLDD simulation in order to find the correct output responses to be saved at this point. We inject a mutated operator to the node *m* and simulate. As the final step the simulated output responses are compared to the correct ones to determine whether the mutant has been killed or not.

Algorithm 2 HLDD-based mutation analysis

```
HLDD_MA()
  SimulateHLDD() /* Algorithm 1*/
  Save output responses
  For each node m
    For each mutated operation p where x_m=Z(m) ≠ p
      Replace x_m by p
      SimulateHLDD() /* Algorithm 1*/
      If output responses differ from the saved ones then
        Report mutant killed
      End if
    End for
  End for
End HLDD_MA
```

TABLE III. MUTATION ANALYSIS EXPERIMENTS ON THE HLDDS OF ITC99 BENCHMARKS

	b01	b02	b04	b06	b09	b11	b13
# Vectors	14	10	8	11	23	88	11
# Mutants inserted	154	78	233	336	213	375	972
# Mutants killed	49	9	18	39	17	178	77
Mutation coverage	0.32	0.12	0.08	0.12	0.08	0.47	0.08
Time, s	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	0.22	0.21

VI. EXPERIMENTAL RESULTS

In the first part there are mutation analysis experiments with the ITC99 circuits [16], which were introduced in order to measure the quality of test generation in hardware systems. The second part presents experiments on an industrial design implementing a cyclic redundancy check (CRC) from the FP6 VERTIGO project [17].

Basic quantitative VHDL characteristics of the ITC99 benchmarks and the CRC design are listed below in Table 2. In the Table, the number of VHDL code lines, primary input signals, primary output signals, variables/signals corresponding to registers and the number of VHDL processes is reported, respectively.

Table 3 presents the mutation analysis experiments on the full-HLDD versions of the ITC99 benchmarks. The row '# Vectors' shows the number of stimuli in the test bench. All the test benches provide 100 % statement coverage, except for b11 (97 %) and b13 (96.1%), where we were unable to create full tests. All the test sets were generated manually.

The next row shows the number of mutants injected by the proposed approach. The row "# Mutants killed" presents the total number of mutants killed. The row "Mutation coverage" shows the ratio of killed mutants to the number of mutants injected in the approach. One of the most interesting observations is the very low mutation coverage: only 8 per cent for b04, b09 and b13. The explanation lies in rather short test sets. Nevertheless, this gives an idea how small observation coverage is guaranteed by 100 % code coverage tests in the worst case.

The last row shows the execution times of the mutation analysis, which were in the range of tenths of seconds. All the experiments were run on a 1.7 GHz laptop PC.

Table 4 lists the results of mutation analysis experiments with previously described ITC99 benchmarks using longer tests, covering also branches. Mutation coverage has increased in most cases but remains still low, which clearly states the need for better test sets. The enormous rise of processing time with b13 can be explained by the fact that test length was increased 100 times.

TABLE IV. MUTATION ANALYSIS EXPERIMENTS ON THE HLDDS OF ITC99 BENCHMARKS

	b01	b02	b04	b06	b09	b11	b13
# Vectors	23	14	11	52	33	132	1148
# Mutants inserted	154	78	233	336	213	375	972
# Mutants killed	57	9	32	50	35	198	281
Mutation coverage	0.37	0.12	0.14	0.15	0.16	0.53	0.29
Time, s	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	0.34	15.36

Results of the mutation analysis experiments on the CRC example are presented in Table 5. The rows in this table have similar semantics to the ones in Table 3. It can be seen that the HLDD-based mutation analysis time is in the range of seconds. Again, the mutation coverage is very low (only 21 per cent) compared to the code coverage. While partly explained by the

short test set it confirms the weak observation coverage guaranteed by code coverage tests and motivates the use of mutation analysis.

TABLE V. HLDD-BASED MUTATION ANALYSIS EXPERIMENT ON THE CRC EXAMPLE

	CRC
# Vectors	42
# Mutants inserted	1247
# Mutants killed	268
Mutation coverage	0.21
Time, s	3.73

VII. CONCLUSIONS

The paper presented a new tool for mutation testing in hardware description languages using the system model of high-level decision diagrams (HLDD). The tool is integrated into the APRICOT verification environment. It is based on HLDD simulation and graph perturbation. A strategy that relies on a restricted set of five key mutation operators is developed in order to speed up the mutation analysis.

Experiments on several ITC99 benchmarks and an industrial example prove the feasibility of the approach. The tests showed that the mutation coverage was always very low compared to the code coverage. While partly explained by the short test sets applied it confirms the weak observation capabilities guaranteed by code coverage tests and motivates the use of mutation analysis.

Future work includes experimental study of real defects, comparison with HDL mutation analysis and identification of equivalent mutants.

ACKNOWLEDGMENT

The work has been supported by European Commission Framework Program 7 projects FP7-2007-IST-1-217069 COCONUT, FP7-ICT-2009-4-248613 DIAMOND and FP7-REGPOT-2008-1 CREDES, by European Union through the European Regional Development Fund, by Estonian Science Foundation grants 7068 and 7483.

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] R. A. DeMillo, D. S. Guindi, W. M. McCracken, A. J. Offutt, K. N. King, " An extended overview of the Mothra software testing environment", Second Workshop on Software Testing, Verification, and Analysis, pp. 142-151, July 1988.

[6] Cristiana Bolchini and Luciano Baresi, "Software Methodologies in VHDL Code Analysis", Journal of Systems Architecture: the EUROMICRO Journal, Elsevier, Volume 44 , Issue 1 (October 1997), Pages: 3 – 21, 1997.

[7] G. Hayek, C. Robach, "From Specification Validation to Hardware Testing: A Unified Method", Proceedings of the IEEE International Test Conference, pp. 885–893, 1996.

[8] Certitude. URL: http://www.springsoft.com/products/functional-qualification/certitude, 2009

[9] Ian G. Harris, "A Coverage Metric for the Validation of Interacting Processes", Proceedings of the conference on Design, Automation and Test in Europe (DATE’06), pp. 1019 – 1024, 2006

[10] Tao Lv, Jian-Ping Fan, Xiao-Wei Li, and Ling-Yi Liu, "Observability Statement Coverage Based on Dynamic Factored Use-Definition Chains for Functional Verification", Journal of Electronic Testing: Theory and Applications, Springer, Volume 22 , Issue 3 (June 2006), Pages: 273 – 285, 2006.

[11] Serdar Tasiran and Kurt Keutzer, "Coverage Metrics for Functional Validation of Hardware Designs", IEEE Design & Test, Volume 18 , Issue 4 (July 2001), Pages: 36 – 45, 2001.

[12] Raimund Ubar, Adam Morawiec, Jaan Raik. Back-Tracing and Event-Driven Techniques in High-Level Simulation with Decision Diagrams, Proc. of the IEEE ISCAS'2000 Conference, Vol. 1, pp. 208-211, Geneva, Switzerland, May 28-31, 2000.

[13] M.Jenihhin, J.Raik, A.Chepurov, U.Reinsalu, R.Ubar, “High-Level Decision Diagrams based Coverage Metrics for Verification and Test”, Proc. of 10th IEEE Latin American Test Workshop (LATW’09), March 1-5, 2009, pp. 1-6

[14] 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.

[15] Jaan Raik, Raimund Ubar, Taavi Viilukas, Maksim Jenihhin. Mixed Hierarchical-Functional Fault Models for Targeting Sequential Cores. Elsevier Journal of Systems Architecture, Vol. 54, Issue 3-4, pp. 465-477, Elsevier, March-April 2008.

[16] URL: http://www.cerc.utexas.edu/itc99-benchmarks/ bench.html, 2009

[17] URL: http://www.vertigo-project.eu, 2009

[18] 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

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

[20] R.Ubar, J.Raik, A.Morawiec, “Back-tracing and Event-driven Techniques in High-level Simulation with Decision Diagrams”,ISCAS 2000, Vol. 1, pp. 208-211.

[21] J. Raik, and R. Ubar. “Sequential Circuit Test Generation Using Decision Diagram Models,” Proc. of the DATEConference, Munich, Germany, March 9-12, 1999, pp. 736-740

Appendix II

Research paper II

Guarnieri, Valerio; Di Guglielmo, Giuseppe; Bombieri, Nicola; Pravadelli, Graziano; Fummi, Franco; Hantson, Hanno; Raik, Jaan; Jenihhin, Maksim; Ubar, Raimund. "On the Reuse of TLM Mutation Analysis at RTL". Journal of Electronic Testing-Theory and Applications, 28(4), 2012, pp. 435–448.

Contributes to Section 3.2 of this Thesis. The author's contributions are: participating in development of the RTL-TLM-based mutation analysis method, performing experiments using Mentor Graphics CatapultC software and presenting the paper at 12th Latin-American Test Workshop. Paper II was an extended version of the latter.

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.

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

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

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.

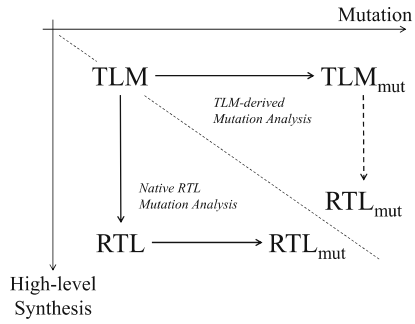


Fig. 1 High-level synthesis and mutation analysis may be combined in different ways

- 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

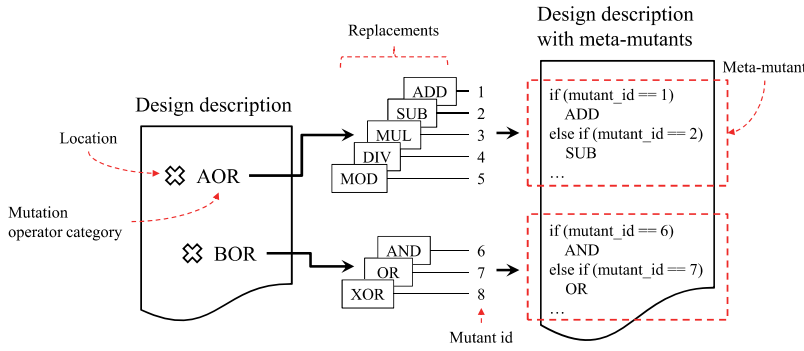


Fig. 2 The mutant-injection process. For each candidate location in the original design description, replacements are identified according to Table 1. The replacements generate mutants of the code and the resulting unique meta-mutated description is parameterized over the mutant id

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.

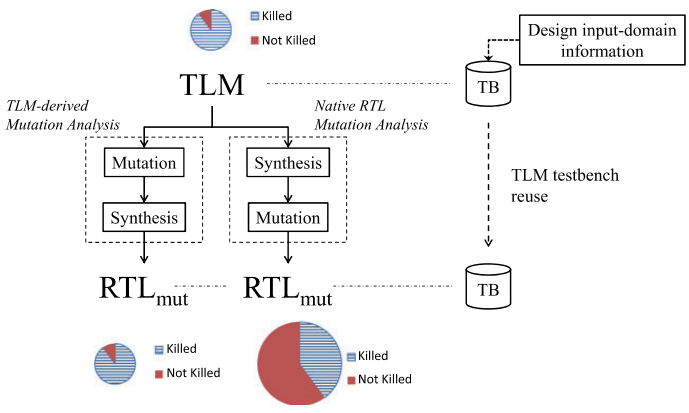


Fig. 3 TLM testbench reuse scenarios and corresponding mutation coverage. A higher number of mutants occurs along the native RTL mutation analysis, i.e., synthesis followed by mutation. Testbenches are generated at transaction level starting from the design specifications and test plan, which typically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

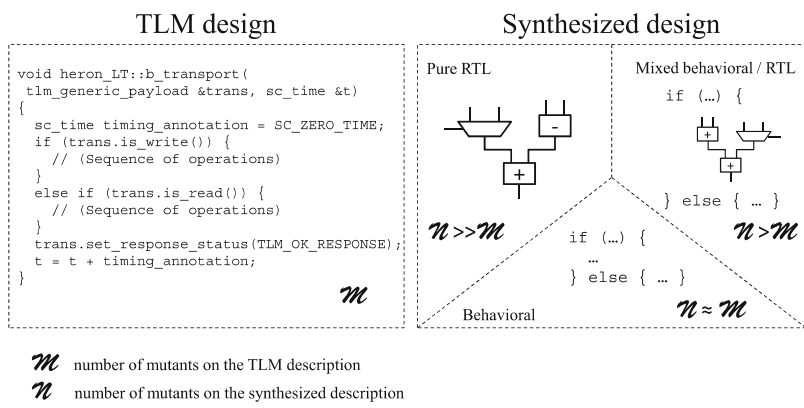


Fig. 4 Relationship between number of mutants on the TLM and the synthesized descriptions. According to the complexity of the TLM statements, the synthesis process may produce RTL code at different levels of abstraction. Typically, a pure RTL, i.e., low level, description provides an significantly higher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

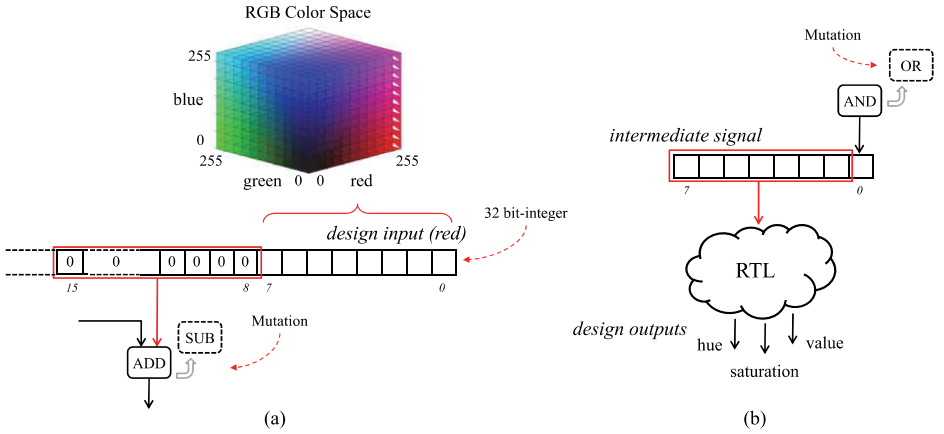


Fig. 5 Examples of live-mutants causes for native RTL mutation analysis. **a** The mutant depends on a bit range of the integer input port which is never assigned by the testbench: by default integers in C/C++ are initialized to 0. **b** The mutant operates on a bit which does not propagate to the outputs

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	–	–	–

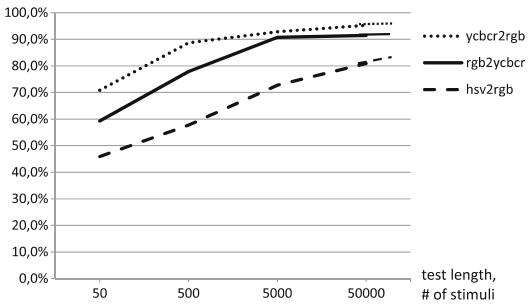


Fig. 6 RTL-native mutation coverage depending on the test length

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.

Appendix III

Research paper III

Raik, Jaan; Repinski, Urmas; Tšepurov, Anton; Hantson, Hanno; Ubar, Raimund; Jenihhin, Maksim. "Automated design error debug using high-level decision diagrams and mutation operators". Microprocessors and Microsystems: Embedded Hardware Design, 37(4), 2013, pp. 1–10.

Contributes to Section 4.1 of this Thesis. This paper was based on author's work on mutation analysis developed in Paper I.

Microprocessors and Microsystems 37 (2013) 505–513

Contents lists available at SciVerse ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

Automated design error debug using high-level decision diagrams and mutation operators

Jaan Raik *, Urmas Repinski, Anton Chepurov, Hanno Hantson, Raimund Ubar, Maksim Jenihhin

Tallinn University of Technology, Department of Computer Engineering, Akadeemia tee 15a, 12618 Tallinn, Estonia

ARTICLE INFO

Article history:
Available online 7 December 2012

Keywords:
Design errors
Debug
Mutation operators
High-level decision diagrams

ABSTRACT

The paper proposes a method for locating design errors at the source-level of Register-Transfer Level (RTL) hardware description language code using the design representation of High-Level Decision Diagram (HLDD) models and correcting them by applying mutation operators. The error localization is based on backtracing the mismatched and matched outputs of the design under verification on HLDDs. As a result of the localization step, all the variables in the RTL description receive a suspiciousness score. Subsequently, a mutation-based correction algorithm is applied providing automated correction for the design under verification. Experiments on a set of sequential RTL benchmarks show that the method is capable of locating the design errors injected with a high accuracy and a short run time. In fact a majority of the errors injected in the experiments were identified as top suspects by the proposed diagnosis algorithm. Furthermore, we show that because of this localization accuracy the mutation-based correction requires very small number of iterations and thus a short run-time.

1. Introduction

Designing a microelectronic chip is an expensive task and excessive design costs are the greatest threat to continuation of the semiconductor industry's growth [1]. It is a well acknowledged fact that verification is forming a major part in the total product design cycle [2] and this trend is continuing. At the same time when there have been numerous research works on verification methods identifying the occurrences of errors, the problem of diagnosing the causes of errors and correcting them has received less attention. Yet a large part of the verification cycle is consumed inside the design loops between debugging and correction. It is estimated that fault localization and correction constitute roughly half of the total time spent on verification [3]. Verification and debug (i.e. assuring the correctness of the design), in turn, represent the main reason of the excessive design costs, accounting for about 70% of the total expenses [2].

Automated debug of design errors consists of two steps: error localization and error correction. Error localization identifies the portion of the design responsible for the erroneous behavior, while error correction is responsible for locally modifying the functionality of the identified portion.

For error localization, simulation-based [6–11] and formal approaches [22] are known. It is widely accepted that simulation-based techniques scale well with the design size, but are not exhaustive while formal techniques provide a high grade of confidence in the results but are susceptible to the design complexity.

For error correction, error matching [4,5] and re-synthesis [6] have been investigated in the literature. In particular, re-synthesis provides a correction which is represented as a partial truth table based on the stimuli under consideration. This kind of correction is not readable and cannot be easily understood and verified by the design engineer. Moreover, as we will show in Section 9, the resynthesized erroneous portion of the design is likely to fail when new stimuli will be added to the suite.

Previous works on error debug for high-level models such as the Register-Transfer Level (RTL) are based on the work by Smith et al. [8]. There is a range of works extending this idea of the SAT-based debug (e.g. [9,10]). However, these methods reduce the debugging problem to SAT or SAT Modulo theory (SMT) solvers, which is an NP-complete problem. Although SAT/SMT engines are being constantly developed and improved, there is a limit to the circuit size where the approach is applicable. The current paper considers a different approach relying on design error localization utilizing HLDD backtrace that executes in polynomial time. This means that much larger designs could be potentially handled by the proposed method.

This paper utilizes HLDD backtrace and mutation as a source-level reasoning engine for automated debug. In our case the engine operates directly on the register-transfer level. This results in a readable diagnostic feedback and is therefore better understandable to the engineer than logic-level debug information provided by previous methods.

* Corresponding author.
E-mail address: jaan@pld.ttu.ee (J. Raik).

0141-9331/$ - see front matter © 2012 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.micpro.2012.11.004

Recently, a similar approach has been adopted in software testing. In [11], Debroy and Wong propose a program slicing based diagnosis tool Tarantula to calculate the suspiciousness scores for operations and apply mutation to correct C and Java programs. The current approach for hardware debug and the one proposed in [11] for software debug were developed simultaneously and independent of each other.

The rest of the paper is organized as follows. Sections 2–4 provide the preliminaries of HLDD modeling and simulation. Section 2 defines the HLDD data structure. Section 3 presents simulation on HLDDs. In Section 4, we explain how digital circuits can be modeled by HLDDs. In Section 5, we present the method for design-error diagnosis based on HLDD backtrace. Section 6 provides a motivational example for the proposed error localization method. Section 7 introduces HLDD-based mutation for correcting design errors. Section 8 presents the experimental results. Section 9 discusses the threats to validity and limitations of the approach and Section 10 concludes the paper.

2. High-level decision diagrams

Different kinds of Decision Diagrams (DD) have been applied to design verification for about two decades. Reduced Ordered Binary Decision Diagrams (ROBDD) [12], as canonical forms of Boolean functions, have their application in equivalence checking and in symbolic model checking. In this paper, we use a decision diagram representation called High-Level Decision Diagrams (HLDDs) that are word-level decision diagrams which can be considered as a generalization of BDD, where instead of single bits, computer words are considered. There exist a number of other word-level decision diagrams such as Multi-Terminal DDs (MTDDs) [13], Kronecker Multiplicative Binary Moment Diagrams (K*BMDs) [14] and Assignment Decision Diagrams (ADDs) [15]. However, in MTDDs the non-terminal vertices hold Boolean variables only, whereas in HLDDs the terminal vertices may be labeled by word-level variables. In K*BMDs, additive and multiplicative weights label the edges. Such representations are useful for compact canonical representation of functions on integers (especially wide integers). However, the main goal of HLDD representations described in this paper is not canonicity but ease of simulation and diagnosis. The principal difference between HLDDs and ADDs lies in the fact that ADDs edges are not labeled by activating values. In HLDDs the selection of a vertex activates a path through the diagram, which derives the needed value assignments for variables.

In this Section we first define the HLDD representation, then we introduce HLDD based simulation and representation for behavioral register-transfer level VHDL descriptions.

Consider a digital system (Z,F) as a network of subsystems or components, where Z is the set of variables (Boolean, Boolean vectors or integers), which represent connections between components, primary inputs and primary outputs of the network. Let $Z = X \cup Y$, where X is the set of function arguments and Y is the set of function values where $Q = X \cap Y$ is the set of state variables. $D(z)$ denotes the finite set of all possible values for $z \in Z$ and $D(Z')$ is the set of all possible vectors in some variable set $Z' \subseteq Z$. Obviously, if $Z' = \{z_1,\ldots,z_n\}$ then $D(Z') = D(z_1) \times \cdots \times D(z_n)$. Let F be the set of discrete functions: $y_k = f_k(X_k)$, where $y_k \in Y$, $f_k \in F$, and $X_k \subseteq X$(k iterates over all elements in F).

Definition 1. High-level decision diagram representing a function f_k: $D(X_k) \to D(y_k)$ is a directed acyclic multigraph $G = (V,E)$ with a single root vertex and a set of terminal vertices where:

- V is the set of vertices and E is the set of edges.
- Each edge $e \in E$ is an ordered pair $e = (v_1, v_2) \in V^2$, where V^2 is the set of all the possible ordered pairs in the set V.
- Each non-terminal vertex is labeled by some input or control variable $x \in X$. We shall denote the variable of vertex v by x_v.
- Each terminal vertex w is labeled by some function g_w: $D(X_w) \to D(y_k)$, where $X_w \subseteq X_k$.
- Each edge $e = (v,u)$, where v and u are vertices, is labeled by some constant $c_e \in D(x_v)$.
- Each two edges $e_1 = (v,u_1)$ and $e_2 = (v,u_2)$ starting from the same source vertex are labeled by different constants $c_{e1} \neq c_{e2}$.
- If the vertex v is labeled by x_v then the number of edges starting from this vertex is $|D(x_v)|$.

Remark 1. Each BDD is HLDD as well, with two terminal vertices labeled by constant functions 0 and 1, and $D(x) = \{0,1\}$ for every variable x.

In other words, HLDD is a data structure similar to BDD, but with many edges originating from a particular vertex and a number of functions at the end, instead of constants 0 and 1. We shall denote the set of terminal vertices by V^T and the set of non-terminal vertices by V^N and the set of all successors of the vertex v by $\Gamma(v)$. For non-terminal vertices $v \in V^N$ an onto function exists between the values $c \in D(x_v)$ of labels x_v and the successors $v^f - \in \Gamma(v)$ of v. By v^f we denote the successor of v for the value $x_v = c$.

The edge (v,v^f), which connects vertices v and v^f, is called *activated* iff there exists an assignment $x_v = c$. Activated edges, which connect v_i and v_j, form an *activated path* $l(v_i,v_j) \subseteq V$. An activated path $l(v_0,v^T)$ from the root vertex v_0 to a terminal vertex v^T is called the *full activated path* and v^T itself is referred to as the activated terminal vertex.

Without loss of generality we assume further that each variable has at least two values, i.e. $\forall z \in Z$, $|D(z)| > 1$. Let D_i designate a subset of $D(x_v)$ labeling vertex v, such that assignments from it will activate its successor vertex v_i. $D(x_v)$ is partitioned into non-intersecting sets $D_1, \ldots, D_m$, where $m = |\Gamma(v)|$. More formally,

$$\bigcup_{i=1}^{m} D_i = D(x_v) \wedge \forall i,j, \quad i \neq j \to D_i \cap D_j = \emptyset.$$

In other words, with every value assignment to variable x_v one and only one successor vertex will be activated. Further, let D_k designate a subset of $D(X)$, such that assignments from it will activate the terminal vertex v_k^T. With every value assignment to variables X, one and only one terminal vertex will be activated. Thus, $D(X)$ is partitioned into non-intersecting sets $D_1, \ldots, D_t$, where $t = |V^T|$:

$$\bigcup_{k=1}^{t} D_k = D(X) \wedge \forall k,l, \quad k \neq l \to D_k \cap D_l = \emptyset.$$

Fig. 1 presents a HLDD G_y representing a discrete function $y = f(x_1,x_2,x_3,x_4)$. The diagram contains five vertices $v_0, \ldots, v_4$. The root vertex v_0 is labeled by variable x_2 which is an integer with a range from 0 to 7. The vertex has three outgoing edges entering the vertices v_1, v_3 and v_4. The vertex v_1 is labeled by x_3 with a range from 0 to 3. It has two outgoing edges e_4 and e_5 entering terminal vertices v_2 and v_3, respectively. The edge e_4 is activated by $x_3 = 2$, while the edge e_5 is activated by x_3 having a value 0, 1 or 3. The ranges of variables x_1 and x_4 labeling terminal vertices v_3 and v_2, respectively, are not evident from the figure.

3. Simulation on HLDD models

HLDD models can be used for representing digital systems. In such models, the non-terminal vertices correspond to conditions

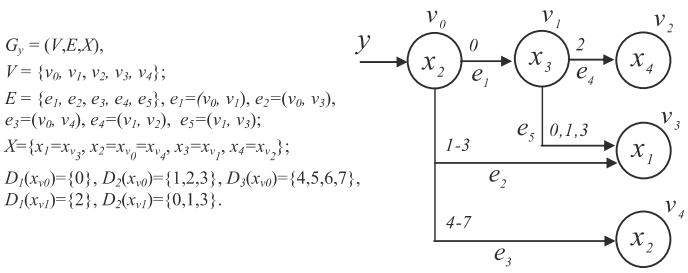


Fig. 1. Graphical representation of a HLDD for a function $y = f(x_1,x_2,x_3,x_4)$.

or to control signals, and the terminal vertices represent arithmetic operations, variables or constants. When representing systems by decision diagram models, in general case, a network of HLDDs rather than a single HLDD is required. During the simulation in HLDD systems, the values of some variables labeling the vertices of an HLDD are calculated by other HLDDs of the system.

Simulation on high-level decision diagrams takes place as follows. Consider a situation, where all the vertex variables are fixed to some value. According to these values, for each non-terminal vertex a certain output edge will be chosen which enters into its corresponding successor vertex. As mentioned above, such connections between vertices are referred to as the activated edges under the given values. Succeeding each other, activated edges form in turn activated paths. For each combination of values of all the vertex variables there exists always a corresponding activated path from the root vertex to some terminal vertex. Let us call this path the main activated path. The simulated value of the variable represented by the HLDD will be the value of the variable labeling the terminal vertex of the main activated path.

In Fig. 2 simulation on the decision diagram presented in Fig. 1 is shown. Assuming that variable x_2 is equal to 2, a path (marked by bold arrows) is activated from vertex v_0 (the root vertex) to a terminal vertex v_3 labeled by x_1. The value of variable x_1 is 4, thus, $y = x_1 = 4$. Note that this type of simulation is inherently event-driven since we have to simulate those vertices only (marked by gray color in Fig. 2) that are traversed by the activated path.

Algorithm 1 presents simulation on HLDD models. The simulation process starts in the root vertex v_0 (line 2 of the algorithm). The vertex $v_{Current}$ is iteratively replaced by its successor vertices selected according to the value of $x_{vCurrent}$ (line 4). In order to represent feedback loops in the RTL design, the algorithm takes the previous time-step value of variable x_k labeling a vertex v_i iff x_k represents a clocked variable in the corresponding HDL (lines 5 and 6). Otherwise, the present time step value will be used (line 8). Simulation ends when a terminal vertex is reached and the variable y corresponding to the simulated HLDD G_y is assigned the value $x_{vCurrent}$ (line 12).

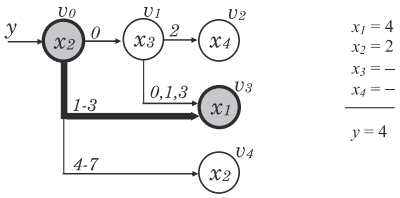


Fig. 2. Simulation on a decision diagram.

Algorithm 1. HLDD simulation

```
1:   SimulateHLDD (G_y)
2:     v_Current = v_0
3:     While v_Current ∉ V^T
4:       x_k = x_vCurrent
5:       If x_k is clocked then
6:         Value = previous time-step value of x_k
7:       Else
8:         Value = present time-step value of x_k
9:       End if
10:      v_Current = v_Current^Value
11:    End while
12:    Assign y = x_vCurrent
13:  End SimulateHLDD
```

4. Representing RTL designs by HLDDs

Consider the datapath depicted in Fig. 3a and its corresponding HLDD representation shown in Fig. 3b. Here, R_1 and R_2 are registers (R_2 is also a primary output), MUX_1, MUX_2 and MUX_3 are multiplexers, + and $*$ denote addition and multiplication operations, IN is an input bus, SEL_1, SEL_2, SEL_3 and EN_2 serve as control signals (multiplexer selects and register enables), and a, b, c, d and$e$ denote internal buses, respectively. In the HLDD, the control variables SEL_1, SEL_2, SEL_3 and EN_2 are labeling the internal decision vertices of the HLDD. The terminal vertices are labeled by word-level variables R_1 and R_2 (data transfers to R_2), and by expressions related to the data manipulation operations of the network.

Consider, simulating HLDD with some values assigned to the variables. Let the value of SEL_2 be 0, the value of SEL_3 be 3 and the value of EN_2 be 1 in the current simulation run. A full activated path in the HLDD from EN_2 to $R_1 * R_2$ is shown by bold lines and gray vertices, which corresponds to the pattern $EN_2 = 1$, $SEL_3 = 3$, and $SEL_2 = 0$. The activated part of the network at this pattern is denoted by gray boxes.

The main advantage and motivation of using HLDDs compared to the netlists of primitive functions is the increased efficiency of simulation and diagnostic modeling because of the direct and compact representation of cause-effect relationships. For example, instead of simulating the control word $SEL_1 = 0$, $SEL_2 = 0$, $SEL_3 = 3$, $EN_2 = 1$ by computing the functions $a = R_1$, $b = R_1$, $c = a + R_2$, $d = b * R_2$, $e = d$, and $R_2 = e$, we only need to trace the vertices EN_2, SEL_3 and SEL_2 on the HLDD and compute a single operation $R_2 = R_1 * R_2$. In case of detecting an error in $R_2$the possible causes can be defined immediately along the simulated path through EN_2, SEL_3 and SEL_2 without complex diagnostic analysis inside the

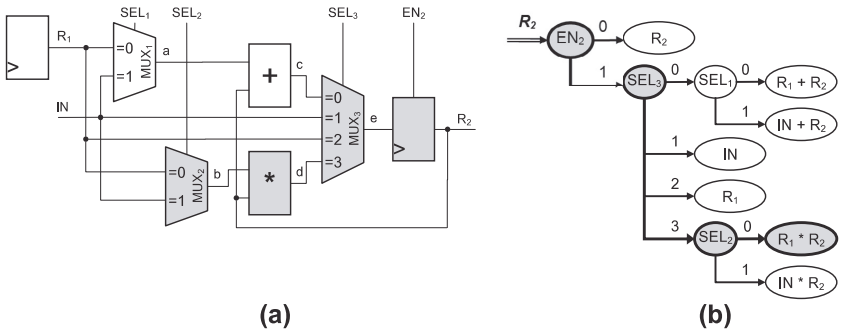


Fig. 3. A datapath of a DUV (a) schematic and (b) HLDD-based representations.

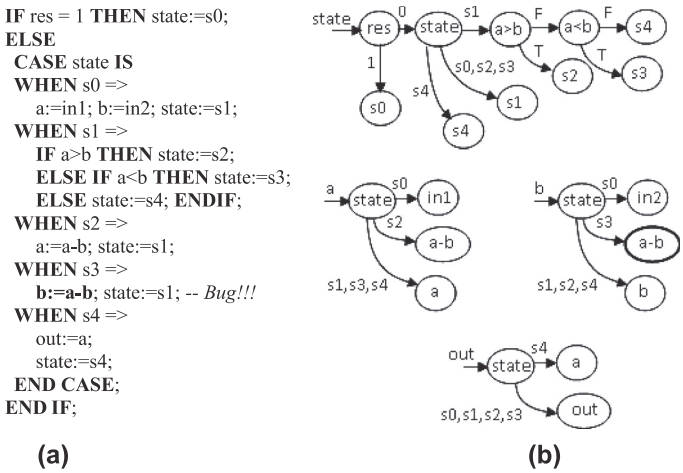


Fig. 4. (a) RTL VHDL and (b) its corresponding HLDD.

corresponding RTL netlist. The activated path provides the *fault candidates*, i.e. variables that are suspected to contain faults causing the error at R_2 during current simulation run. Further reasoning should be based on analyzing sources of these signals.

An example of HLDD representation (Fig. 4b) of a VHDL code fragment of the Euclidean algorithm for calculating the Greatest Common Divisor of two unsigned variables *in1* and *in2* is presented in Fig. 4a. The VHDL fragment contains seven variables: inputs *in1*, *in2* and *res*(the reset signal), internal variables (registers) *a*, *b* and *state* (for control state), and output *out*. The variable *state* is of enumeration type, variables *in1*, *in2*, *a*, *b* and *out* are integers and variable *res* is of bit type.

The algorithm proceeds as follows. When the reset input *res* becomes one, the Finite State Machine (FSM) of the control part is initialized to the state *s*0. In that state, input *in1* is assigned to variable *a*and input *in2* is assigned to variable *b*. The next FSM state is *s*1, where if $a > b$ we move to state *s*2, if $a < b$ we move to state *s*3, and otherwise if $a = b$ we move to state *s*4, respectively. In state *s*2, $a - b$ is assigned to *a*, and in state *s*3, $b - a$ is assigned to *b*. This guarantees that a smaller number is always subtracted from the larger one until *a* and *b* become equal and the FSM ends up in state *s*4, where the result is written to the output variable *out*.

Note that there is a bug in the VHDL description in Fig. 4a. In the FSM state *s*3, $a - b$ and not $b - a$ is assigned to variable *b*. Fig. 4b presents the HLDD models of four variables *state*, *a*, *b* and *out*, i.e. the internal state and output variables of the design.

HLDDs for design variables are generated by traversing the control flow branches of the VHDL code. Conditional statements (IF, CASE) transform into non-terminal vertices of the HLDD, control branches map to the HLDD edges and terminal vertices are created out of the right-hand side parts of value assignments to variables in corresponding control branches. In the figure, the symbols *T* and *F* labeling the HLDD edges stand for *true* and *false*, respectively.

5. HLDD backtrace for design error localization

In this section, we first present the algorithm for diagnostic tree generation using backtrace on HLDD models. Then, two analysis steps are introduced to perform error localization on the set of diagnostic trees generated.

Algorithm 2 presents the recursive diagnostic tree generation on HLDDs. The process starts from the primary outputs (Line 2) and from each clock-cycle (Line 3). Subsequently, the diagnostic tree is recursively generated using the function *RecursiveTreeGeneration*.

Algorithm 2. HLDD-based diagnostic tree generation

```
1:   GenerateDiagnosticTree ()
2:     For each primary output G_O in the model
3:       For each time-step t
4:         δ(G_O, t) = ∅
5:       RecursiveTreeGeneration (G_O, t, δ)
6:     End for
7:   End for
     End GenerateDiagnosticTree
     RecursiveTreeGeneration (G_y, t, δ)
8:   SimulateHLDD (Gy)//See algorithm 1!
9:   For each v_i at the main activated path
10:    If variable x_k = x_vi at-time step t is not in δ then
11:      Add x_k to δ
12:      If x_k is not a primary input then
13:        RecursiveTreeGeneration (Gx_k, t, δ)
14:      End if
15:    End if
16:  End for
     End RecursiveTreeGeneration
```

Algorithm 2 generates a separate diagnostic tree $\delta(G_O, t)$ for each output diagram G_O at each clock-cycle t. The resulting diagnostic tree δ is a set of pairs (x_i, t_j) that show at which time-steps t_j the variable x_i was backtraced.

In the following, two analysis steps that could be implemented for locating the design error are presented. In order to perform the analysis, let us partition the set of all diagnostic trees $\Delta = \delta_k(G_O, t)$ into failing diagnostic trees Δ_F and passing diagnostic trees Δ_P. A diagnostic tree is failing if $\delta_k(G_O, t)$ of the simulated value of output variable $o \in Y$ on the faulty design differs from the corresponding value of the golden device at time-step t. Otherwise, δ_k is called a passing diagnostic tree.

5.1. Diagnosis step 1

For each variable x_i count the number c_{FAILED} of failing diagnostic-trees $\delta_k \in \Delta_F$, where x_i is present at least in one of the pairs (x, t) of δ_k. Select the variables x_i receiving a non-zero score c_{FAILED} as the set of suspected faults $X_{suspected}$ and sort the set $X_{suspected}$ according to the score c_{FAILED}. The variables with a higher score are more suspected of causing the error than the ones with a lower score.

5.2. Diagnosis step 2

Perform step 1. For each variable $x_{step1} \in X_{suspected}$ count the number of passing diagnostic-trees $\delta_l \in \Delta_P$ c_{PASSED}, where x_{step1} is present at least in one of the pairs (x, t) of δ_l. Compute the score $c_{TOTAL} = c_{FAILED}/(c_{FAILED} + c_{PASSED})$ for variables x_{step2}. Sort the set $X_{suspected}$ according to the score c_{TOTAL}.

Step 1 is more exact as it can be easily proven that at least one of the variables x_v that is labeling a vertex v along one of the main activated paths in simulated HLDDs must be also the cause of the error. However, step 2 may be unavoidable in order to guarantee a good diagnostic resolution, especially if the number of failing sequences is one or very small. In fact, the experiments presented in this paper fully confirm this observation.

The straight-forward implementation of this backtracing algorithm could be time-consuming because of the square complexity introduced by the need to backtrace from each subsequent time step back to the initial time step. Therefore, in current implementation we stored intermediate backtracing results at each time step in order to gain speed.

6. Error localization example

Consider the following example of design error localization on the basis of the erroneous GCD design description presented in Fig. 4a. Let there be a given set of input stimuli (e.g. a functional test) and a set of correct output responses for the stimuli obtained on a golden model. Assume that there is a design error in it such that at state $s3$ a faulty operation $a - b$ is assigned to the variable b instead of the correct operation $b - a$. In Fig. 5, two test sequences are presented as tables. Rows of the table show values of the variables at different time-steps. The first column t lists the time steps $t_0, \ldots, t_6$, The next three columns present the values of input variables *res*, *in1* and *in2* in the test sequence. Final four columns show the values of the internal variables *state*, *a*, *b* and the primary output *out*. These values have been obtained by simulating the HLDDs in Fig. 4b using Algorithm 1.

Fig. 5a shows the test sequence for the design when primary inputs *in1* and *in2* hold values 4 and 2, respectively. This sequence passes the test, giving a correct response that the greatest common divisor of 4 and 2 is two. In Fig. 5b, another sequence is presented, which produces an erroneous the test. Because of the design error, the primary output *out* receives an erroneous value.

In order to locate the design error, a diagnostic tree is generated on the HLDD model of the GCD design presented in Fig. 4b. Fig. 6 presents the diagnostic tree for the passing test shown in Fig. 5a while Fig. 7 presents the diagnostic tree for the test shown in Fig. 5b. As it can be seen from the Figures, the "tree" generated by Algorithm 2 has not a tree-like structure. It is rather a directed graph, where the vertices represent a subset of the time-expansion model of the design. Directed edges show relations between the variables in the simulation process.

The algorithm starts at the time step when an output response is expected. For the test sequences in Fig. 5 it is the time step t_6. Then, it continues towards the first time step and recursively generates the diagnostic tree $\delta(G_{out}, t_6)$. For the sake of compactness of presentation, we have omitted the reset variable *res* from Figs. 6 and 7. In addition, the operation $a = b$ (in Fig. 6) is also given in a

t	res	in1	in2	state	a	b	out
t_0	1	4	2	-	-	-	-
t_1	0	-	-	s0	4	2	-
t_2	0	-	-	s1	4	2	-
t_3	0	-	-	s2	2	2	-
t_4	0	-	-	s1	2	2	-
t_5	0	-	-	s4	2	2	-
t_6	0	-	-	s4	2	2	2

(a)

t	res	in1	in2	state	a	b	out
t_0	1	2	4	-	-	-	-
t_1	0	-	-	s0	2	4	-
t_2	0	-	-	s1	2	4	-
t_3	0	-	-	s3	2	-2	-
t_4	0	-	-	s1	2	-2	-
t_5	0	-	-	s4	2	-2	-
t_6	0	-	-	s4	2	-2	2→-

(b)

Fig. 5. Passing (a) and failing (b) test sequences for the GCD design.

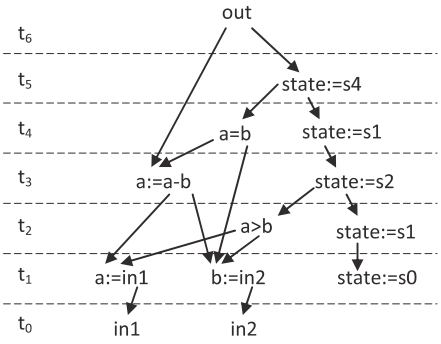


Fig. 6. Diagnostic tree for the passing test in Fig. 5a.

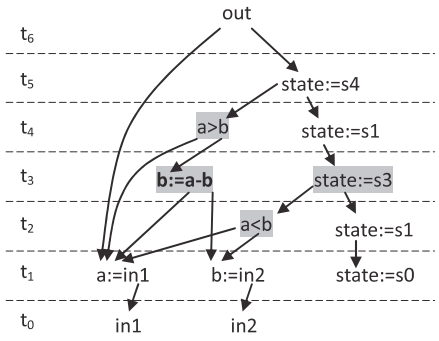


Fig. 7. Diagnostic tree for the failing test in Fig. 5b.

minimized form from $\neg(a > b)\bullet\neg(a < b)$ obtained by backtracing the HLDD for the *state* variable (see Fig. 4b).

The diagnostic trees presented in Figs. 6 and 7 can be used for effect-cause diagnosis of design errors. Reasoning on the diagnostic trees takes place as follows. The diagnosis tree in Fig. 6 of the passing test sequence in Fig 5a contains vertices which are unlikely to be related to the cause of the error because the sequence resulted in a matched output. However, the diagnostic tree in Fig. 7 was backtraced from the mismatched output*out* at time-step t_6. These two backtraces should give us information about the location of the error.

Indeed, the vertex labeled by $b := a - b$ (marked by gray background in Fig. 7) is among the faults selected as suspects for causing the design error by the diagnosis step 2 presented in previous subsection. The four vertices with gray background are chosen as suspects because only these four vertices are present in the diagnostic tree of the failing sequence but are missing from the passing sequence. Thus, in this simple example they receive the highest score. In the real case there would be many failing and passing test sequences as well as there may be multiple faults. Furthermore, in most cases it is not possible to partition the test set into sequences. Algorithm 2 takes the latter assumption. Therefore in experiments reported in current method we start backtrace at each clock cycle for each output.

The HLDD-based diagnosis is related to known debugging techniques such as *program slicing* [16] and *critical path tracing* [17]. Modeling discrete systems by a system of HLDDs may be regarded as a form of program slicing, because a separate diagram is generated for each variable x in the program, reflecting the control flow branches where assignments are made to x and including the data assigned to x. Activating paths in HLDD diagrams using Algorithm 1 is equivalent to critical path tracing. The technique of critical path tracing consists of simulating the fault-free system (true-value simulation) and using the computed signal values for backtracking all sensitized paths from primary outputs towards primary inputs in order to determine the faults that would affect the primary output. In HLDDs the same task is solved in a single run as a byproduct of simulation.

7. Mutation-based design error correction

Mutation analysis is a technique that was initially introduced to fulfill the task of evaluating the ability of testbenches to detect bugs in software programs. In this paper we consider applying mutation operators for correcting a faulty circuit. Subsequent to the fault localization step described in Sections 5 and 6 mutation operators are applied to perturb the HLDD model of the RTL design in order to perform the correction. It is intuitively clear that this kind of correction may be extremely time-consuming in the worst case. The time required to correct the circuit is proportional to the product of the number of vertices, the number of mutants to be injected to each vertex and the number of test patterns in the test.

The design error localization technique presented in previous sections allows to minimize the number of vertices where the faults have to be injected. However, it is crucial to keep the number of mutants as small as possible. In this work, the five *key* operators proposed in [18] have been implemented. In experiments, those five operators 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 [18]. The 5 sufficient operators are ABS, which forces each arithmetic expression to take on the value 0, a positive value, and a negative value, AOR, which replaces each arithmetic operator with every syntactically legal operator, LCR, which replaces each logical connector with several kinds of logical connectors, ROR, which replaces relational operators with other relational operators, and UOI, which inserts unary operators in front of expressions.

We have implemented the five operators with the following constraints and specifics. UOI currently replaces only unary operators with other unary operators and ABS is applied to variables only, and not to expressions. Note also that in HLDD there are no signed/unsigned variables, but signed and unsigned relational operators exist. Therefore ROR replaces, both, signed and unsigned relational operators. In AOR we also allow mutation by division and mod operations and we have included a check for the case of divide-by-zero. The reduced-5-key-operator strategy represents a do fewer strategy. The purpose would be to reduce the mutation analysis cost as much as possible.

Fig. 8 illustrates the HLDD graph perturbations for implementing the five key mutation operators on a sample diagram G_{y_out}. In HLDD models, the perturbation means simply replacement of an operator, variable or constant labeling the HLDD vertex by another operator, variable or constant.

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

Algorithm 3 presents the mutation-based correction algorithm on HLDD representations. In this algorithm, first Algorithm 2 is executed in order to rank the circuit vertices according to the suspiciousness score. Then, the vertices are substituted iteratively by mutants from the same group of functions as the function labeling the vertex (see the groups of functions in Table 1). This iteration stops when the simulation result confirms that the correction provides output responses equal to the golden output responses.

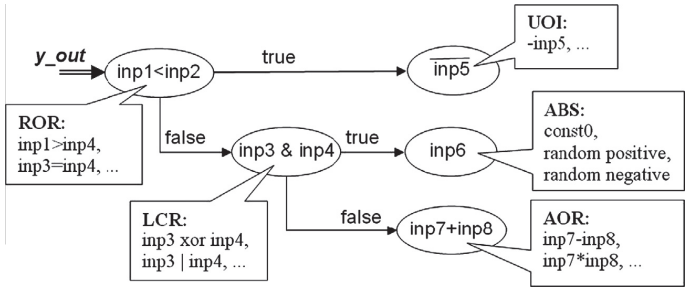


Fig. 8. "Key" mutation operators as HLDD perturbations.

Table 1
The list of key mutation operators.

Mutation operator	List of replacements
LCR (logical connector replacement)	&, NAND, |, NOR, XOR
AOR (arithmetic operator replacement)	+, −, ·, /, MOD
UOI (unary operation insertion)	NEGATE, INVERT
SOR (shift operator replacement)	SHIFT LEFT, SHIFT RIGHT, UNSIGNED SHIFT RIGHT
ROR (relational operation replacement)	=, ≠, <, >, ⩽, ⩾

Algorithm 3. Mutation-based correction as an HLDD perturbation

```
MutationBasedCorrection ()
 1:  Rank the vertices v ∈ V by executing
     GenerateDiagnosticTree ()/* Algorithm 2*/
 2:  For each vertex v
 3:    For each substitution p where x_v ≠ p in the mutation
     operator
 4:      Substitute x_v by p
 5:      SimulateHLDD ()/* Algorithm 1*/
 6:      If output responses match the golden ones then
 7:        Return correction "x_v to be substituted by p"
 8:      End if
 9:    End for
10:  End for
11:  Return "The design cannot be corrected by mutation"
12:  End MutationBasedCorrection
```

8. Experimental results

Table 2 presents the main characteristics of the benchmarks used in the experiments and their respective test sets. The benchmarks include the Greatest Common Divisor (*gcd*) and the Differential Equation (*diffeq*) examples from the HLSynth92 and HLSynth95 academic benchmarks suite, respectively. The design *risc* is a processor example from a FUTEG research project. In addition, two real-world designs were included to the experiments. These were a commercial core for circular redundancy check (*crc*) from [19] and an opensource core *uart16750* from the OpenCores repository [21]. The test stimuli for the academic benchmarks were generated by a hierarchical test pattern generator Decider [20] while for *crc* the provided functional test bench was applied and *uart16750* was tested by 1000 randomly generated test vectors. The second column reports the system complexity in terms of the number of HLDD vertices. The third column represents the number of functions in the design. Finally, the fourth column shows the number of stimuli in the test suite.

In Table 3, the design error localization experiments are provided. We injected faults into the design by randomly mutating a function one-by-one, so that during each diagnosis run only one function was mutated. The column 'success rate' shows the ratio of the times the actual location of the mutation achieved **the highest rank** in relation to all diagnosis runs. The column 'average resolution, # suspects' reports the average number of suspects that received the highest score. Here, the diagnostic resolution is very good for step 2 and two or more times worse for step 1. The same trend applies to the worst resolution, which reports the worst case suspected fault list size over all the faults injected. The final column reports the run times achieved on a PC, Dual-Core CPU, 2.6 GHz, 3.25 GB RAM, Windows XP operating system are provided. This time includes both performing step 1 and step 2 of the diagnosis algorithm. As it can be seen, the run times are very different. They do not only depend on the circuit size but also the number of vectors and the sequential depth of the designs. The run time for step 1 is actually very much shorter than the time for steps 1 and 2 combined, because in step 1, only mismatched outputs have to be backtraced. Table 3 excludes the error localization details for the core *uart16750*. The time for localization for this core was in average 90.0 s on the 1000 vector test.

As shown in the previous table, a majority of the errors injected in the experiments were identified as top suspects by the proposed diagnosis algorithm. Because of this localization accuracy the mutation-based correction requires very small number of iterations and thus a short run-time. See Table 4 which lists the average time to correct a design by applying mutation. The last column of Table 4 shows the average number of substitution functions (mutants) generated until the design was corrected.

9. Threats to validity and limitations

Similarly to a wide range of design error correction methods (e.g. [8,10,11]), one of the main limitations of the work proposed in this paper is the fact that the correction is calculated with

Table 2
Benchmarks and their test sets.

Design	# Vertices	# Fun.	# gates	#FFs	# Test stimuli
gcd	25	4	∼500	48	4000
diffeq	39	9	∼2500	80	16,855
risc	61	16	∼2000	96	4000
crc	232	74	∼10,000	171	193
uart16750	1747	401	∼100,000	1403	1000

Table 3
Design error localization experiments.

Design	Success rate, ratio of correct localizations		Average resolution, # suspects		Worst resolution, # suspects		Processing time, s
	Step1	Step2	Step1	Step2	Step1	Step2	
gcd	4/4	4/4	2.25	1.00	3	1	18.0
diffeq	9/9	9/9	3.33	1.88	6	3	700.0
risc	16/16	13/16	8.18	1.93	11	5	0.3
crc	74/74	69/74	31.83	9.04	50	20	0.5

Table 4
Mutation based correction experiments.

Design	Average correction time, s	Average number of substitutions
gcd	0.0040	2.0
diffeq	0.0410	3.62
risc	0.0276	2.25
crc	0.0422	4.13
uart16750	0.581	9.11

respect to a given set of input stimuli. Therefore, it may happen that if the test stimuli set is not comprehensive, a correction is calculated that holds for this test set only and will not hold for every possible input stimulus. There exist also design error correction methods that are, in theory, able to fix the circuits regardless of input stimuli. However, such approaches have serious size limitations. One of the most recent examples of this category of correction methods has been presented in [22]. Experiments on the Siemens benchmark set [23] showed that due to scalability limitations only very few buggy versions of the benchmarks could be handled by the method. Furthermore, in order to be feasible at all the bit-width of the designs had to be reduced from 32 down to 8 bits.

Another major limitation of the correction method is that it is an error-matching approach. In other words, a design error can be fixed if there is a fault described in a library that when inserted converts the design equivalent to the correct design. It is obvious that real design bugs may often not be represented by simple mutations. However, recently several studies have been published that show the applicability of mutation-based correction for realistic debugging cases. Debroy and Wong propose mutation-based correction for the control flow of software programs and show that 16% of all 132 buggy versions of Siemens benchmarks can be corrected by the approach. In the work by Repinski et al. [24], this number is increased to more than 50% mainly by additionally extending the mutation operators to dataflow.

Despite of the fact that mutation is not capable of fixing arbitrary design errors, it provides a feedback to the design engineer that is provided at the source level (i.e. RTL) and is more readable than the partial truth-tables or synthesized logic-level blocks obtained by resynthesis-based correction (e.g. [10]). This readability is crucial to keep the designer in the debugging loop. Methods that provide corrections in an automated way without a clear feedback to the designer are of limited practical application.

The complexity of the design error localization is polynomial versus the exponential complexity of the SAT-based methods [8,10]. Design error correction is polynomial as well, requiring $\sum_{i=0}^{m} k_i$ simulation runs, where m is the rank of the actual fault location in the list of circuit nodes ranked according to their localization score and k_i is the number of substitution mutants in the ith node within the ranked list, starting from the first node and ending with the mth node. Experiments show that m is usually a very small value, which means that the correction process is fast and scalable.

Finally, the effectiveness and efficiency of the presented error diagnosis and correction method strongly depends on the quality of input stimuli. There exist a wide range of Diagnostic Test Pattern Generation (DTPG) methods [25–29] that produce tests, which are designed to locate design errors. Most of such methods are targeting manufacturing faults. Test stimuli generated by DTPG would further improve the quality of locating and correcting design errors.

10. Conclusions

In this paper, a debug method for locating and correcting design errors at the source-level of hardware description language code using the design representation of high-level decision diagrams is proposed. Experiments on a set of sequential register-transfer level benchmarks and one real-world design from the OpenCores repository show that the method is capable of locating the design errors injected with a high accuracy. Because of this localization accuracy the mutation-based correction requires very small number of iterations and thus short run-times.

References

[1] International Technology Roadmap for Semiconductors, Design, 2011. <http://www.itrs.net/Links/2011ITRS/Home2011.htm>.
[2] Hans van der Schoot, Taking verification productivity to the next level, TechOnline India, Mentor Graphics, September 27, 2011.
[3] J. Raik, U. Repinski, M. Jenihhin, A. Chepurov, High-Level Decision Diagram Simulation for Diagnosis and Soft-Error Analysis, Design and Test Technology for Dependable Systems-on-Chip, IGI Publishing, 2010. pp. 294–309.
[4] J.C. Madre, O. Coudert, J.P. Billon, Automating the diagnosis and the rectification of design errors with PRIAM, in: Proc. of ICCAD, November 1989, pp. 30–33.
[5] M.S. Abadir, J. Ferguson, T.E. Kirkland, Logic design verification via test generation, IEEE Transactions on Computer-Aided Design 7 (1) (1988).
[6] M.F. Ali, S. Safarpour, A. Veneris, M.S. Abadir, R. Drechsler, Post-verification debugging of hierarchical designs, in: Proceedings of ICCAD, 2005, pp. 871–876.
[7] A. Wahba, D. Borrione, Design error diagnosis in sequential circuits, Lecture Notes In Computer Science, vol. 987, Springer, 1995, pp. 171–188.
[8] A. Smith, A. Veneris, A. Viglas, Design diagnosis using boolean satisfiability, in: Proc. Asia and South Pacific Design Automation Conference (ASPDAC), 2004, pp. 218–223.
[9] Görschwin Fey, Stefan Staber, Roderick Bloem, Rolf Drechsler, Automatic fault localization for property checking, IEEE Transactions on CAD of Integrated Circuits and Systems 27 (6) (2008) 1138–1149.
[10] Kai-hui Chang et al., Automatic error diagnosis and correction for RTL designs, in: Proc. High-Level Design and Validation Workshop (HLDVT), Irvine, CA, November 2007.
[11] Vidroha Debroy, W. Eric Wong, Using mutation to automatically suggest fixes for faulty programs, in: Proceedings of the Third International Conference on Software Testing, Verification and Validation, Paris, France, 2010, pp. 65–74.
[12] R.E. Bryant, Graph-based algorithms for Boolean function manipulation, IEEE Transactions on Computers C-35 (8) (1986) 677–691.
[13] E. Clarke, M. Fujita, P. McGeer, K.L. McMillan, J. Yang, X. Zhao, Multi terminal BDDs: an efficient data structure for matrix representation, in: Proc. of Int'l Workshop on Logic Synth., 1993, pp. P6a:1–15.
[14] R. Drechsler, B. Becker, S. Ruppertz, K*BMDs: a new data structure for verification, in: Proc. of European Design & Test Conf., 1996, pp. 2–8.
[15] V. Chayakul, D.D. Gajski, L. Ramachandran, High-level transformations for minimizing syntactic variances, in: Proc. of ACM/IEEE DAC, 1993, pp. 413–418.
[16] Mark Weiser, Program slicing, in: Proceedings of the 5th International Conference on Software Engineering, IEEE Computer Society Press, 1981, pp. 439–449.
[17] M. bramovici, P.R. Menon, D.T. Miller, Critical path tracing - an alternative to fault simulation, in: Proceedings of the 20th Design Automation Conference (Miami Beach), Florida, United States, June 27–29, 1983, pp. 214–220.

[18] A.J. Offutt, G. Rothermel, C. Zapf, An experimental evaluation of selective mutation, in: Proceedings of the Fifteenth International Conference on Software Engineering, IEEE, Baltimore, MD, 1993, pp. 100–107.
[19] EU FP6 IST STREP VERTIGO project benchmarks. <http://www.vertigo-project.eu/>.
[20] Jaan Raik, Raimund Ubar, Fast test pattern generation for sequential circuits using decision diagram representations, JETTA, vol. 16, Kluwer Academic Publishers, 2000, pp. 213–226. No. 3.
[21] OpenCores design repository. <http://www.opencores.org/>.
[22] Robert Könighofer, Roderick Bloem, Automated error localization and correction for imperative programs, in: Proceedings of 11th International Conference 2011 Formal Methods in Computer Aided Design (FMCAD 2011), 2011, pp. 91–100.
[23] Siemens benchmark suite. <http://pleuma.cc.gatech.edu/aristotle/Tools/subjects/>.
[24] U. Repinski et al., Combining dynamic slicing and mutation operators for ESL correction, in: Proceedings of the IEEE European Test Symposium (ETS 2012), IEEE Computer Society, 2012.
[25] A. Veneris, R. Chang, M.S. Abadir, M. Amir, Fault equivalence and diagnostic test generation using ATPG, in: Proc. International Symposium on Circuits and Systems, 2004, pp. 221–224.
[26] V.D. Agrawal, D.H. Baik, Y.C. Kim, K.K. Saluja, Exclusive test and its application to fault diagnosis, in: Proc. International Conference on VLSI Design, 2003, pp. 143–148.
[27] I. Pomeranz, S.M. Reddy, A diagnostic test generation procedure for synchronous sequential circuits based on test elimination, in: Proc. International Test Conference, 1998, pp. 1074–1083.
[28] T. Bartenstein, Fault distinguishing pattern generation, in; Proc. International Test Conference, 2000, pp. 820–828.
[29] Naresh K. Bhatti, R.D. Blanton, Diagnostic test generation for arbitrary faults, in: Proc. International Test Conference, 2006, pp. 1–9.

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 postdoc researcher. 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. His main research interests include high-level test generation and verification.

Urmas Repinski is a Ph.D. student at Tallinn University of Technology. His research interest is design error diagnosis and correction techniques.

Anton Chepurov is a PhD student at Tallinn University of Technology. His research interest is modeling hardware designs by high-level decision diagrams. He has co-authored 10 papers.

Hanno Hantson is a Ph.D. student at Tallinn University of Technology and a member of IEEE. His research interest is mutation analysis methods. He has co-authored 2 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. He has given seminars or lectures in 20–25 universities in more than 10 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.

Maksim Jenihhin received his B.Sc. and M.Sc. degrees in Computer Engineering from Tallinn University of Technology (TUT) in 2003 and in 2004, respectively. In 2004–2007, he was employed as a researcher in ELIKO Technology Development Center, Tallinn. Currently he is a Ph.D. student at TUT. He has co-authored 15 papers.

Appendix IV

Research paper IV

Jenihhin, Maksim; Tšepurov, Anton; Tihhomirov, Valentin; Hantson, Hanno; Raik, Jaan; Ubar, Raimund; Bartsch, Günter; Meza-Escobar, Jorge Hernan; Wuttke, Heinz-Dietrich. "Automated Design Error Localization in RTL Designs". IEEE Design & Test of Computers, 1, 2014, pp.83–92.

Contributes to Section 4.2 of this Thesis. The author's contributions are: performing experiments using Apricot software and presenting the paper at 13th Latin-American Test Workshop (Best Paper Award). Paper IV was an extended version of the latter.

Automated Design Error Localization in RTL Designs

Maksim Jenihhin, Anton Tšepurov, Valentin Tihhomirov, Jaan Raik, Hanno Hantson, and Raimund Ubar
Tallinn University of Technology

Jorge Hernan Meza Escobar and Heinz-Dietrich Wuttke
Ilmenau University of Technology

Günter Bartsch
zamiaCAD

Editors' notes:
This paper presents a fault model-free approach for automated design error localization by combining statistical analysis with HDL slicing.
—Nicola Nicolici, McMaster University

Rapidly growing systems' complexity has led to increasing design costs and verification has become one of the most expensive tasks in the design process. While several approaches and tools focusing on identifying the occurrences of errors exist, scalable solutions for design error or bug localization are missing. On one hand, the designer is faced with too much information provided by the verification tools. On the other hand, there is not enough information in order to unambiguously locate the bug. Therefore, manual bug localization activity is very time consuming and there is a need for automated approaches.

In this article, we consider the case where a design described in aHardware Description Language (HDL) has been identified as erroneous during functional verification and, thus, design error localization is required. However, due to the enormous complexity of modern Register-Transfer Level (RTL) designs, several bugs may escape verification and are consequently handled by post-silicon validation, e.g., [1].

A majority of the works on automated error localization are in the software development domain [2], [3]. Design error localization approaches for hardware designs are mostly based on formal techniques, such as model-checkers [4] or SAT/SMT-solvers [5], [6], and thus cannot be applied to large designs. Some works address the scalability issue by proposing abstraction and refinement techniques [7]. However, in order to localize bugs in complex designs, efficient simulation-based approaches should be developed.

In this article, we present a scalable bug localization tool togetherwith a case study of pin-pointing real-world design bugs within a processor design project. The tool has been implemented on top of a highly scalable HDL-centric open source framework zamiaCAD [8], [9] andthe case study has been carried out on a real processor design ROBSY [10], where the design team has documented the set of discovered bugs. Development of this tool was supported by EU's FP7 research initiative DIAMOND [11].

The bug localization method relies on simulation-based statistical ranking of potential bug locations that is refined by the dynamic slicing technique known from software testing. The basis of dynamic slicing is static slicing, however the presence of concurrent constructs in HDLs makes static slice computation considerably more complicated than

Digital Object Identifier 10.1109/MDAT.2013.2271420
Date of publication: 28 June 2013; date of current version: 20 February 2014.

in software [12]. In the proposed approach, we apply static slice computation based on reference graph generation using a through-signal-assignment search from the zamiaCAD semantically elaborated models of processor designs. This allows applying static slicing in a practical and scalable manner for realistic-size industrial designs.

The automated localization method presented in this article goes beyond the state-of-the-art in debugging by providing the following.

- Support for very large industrial designs due to the scalability of zamiaCAD elaboration.
- Accurate localization due to combining statistical analysis with HDL slicing.
- Hierarchical localization in code items. Where applicable the analysis of code statements is refined by bug location candidates in branches and conditions.
- Cone inspection supported by the zamiaCAD infrastructure of through-signal assignments.

The approach is fault-model free, i.e., there is no need to explicitly enumerate the bug types. It also supports localization of multiple bugs. In addition, the method can be executed on the functional test set (i.e., regression suite) of the processor and there is no need for dedicated diagnostic test generation, however it is required that the test set is divided into separate test cases.

zamiaCAD framework

The bug localization method, described here in Figure 1, has been implemented on top of an open source HDL-centric framework zamiaCAD [8], which puts emphasis on scalability and non-intrusiveness. The front-end of zamiaCAD includes a parser and an elaboration engine that both support full VHDL 2002 standard specification. On the back-end side the framework allows design simulation, static analysis and other applications such as synthesis and design structure visualization. zamiaCAD has an Eclipse IDE plug-in based graphical user interface for advanced design entry and navigation.

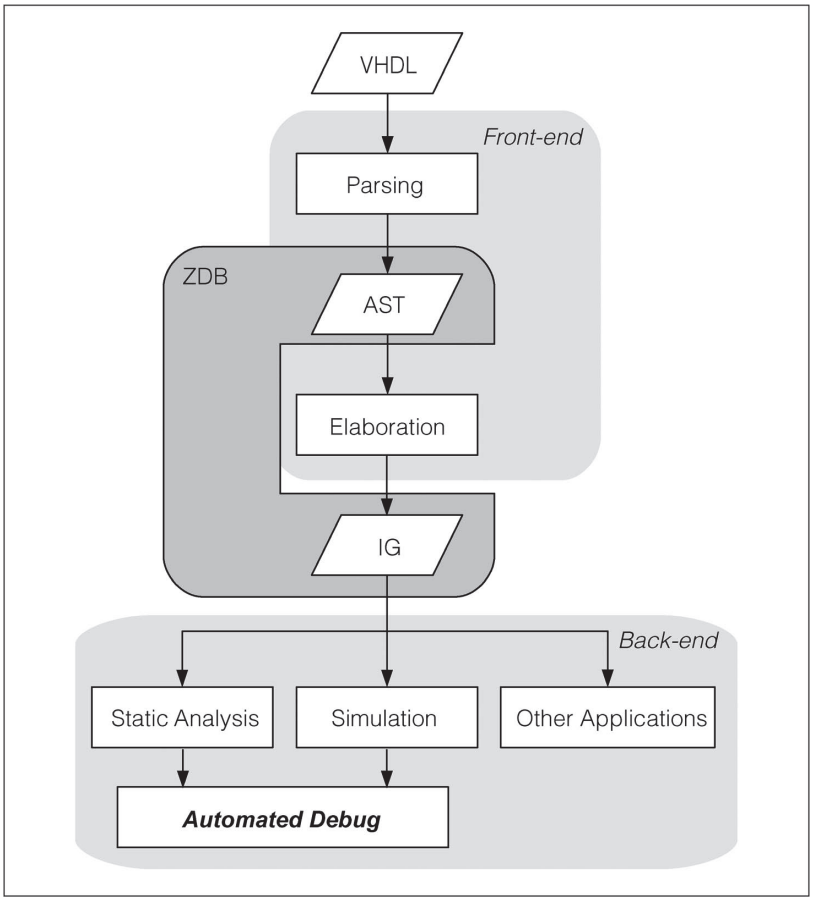


Figure 1. zamiaCAD framework.

An object database ZDB (zamiaCAD Data Base), custom-designed and highly optimized for scalability and performance is used for zamiaCAD applications. Full elaboration in zamiaCAD semantically resolves the Abstract Syntax Tree (AST) generated by the parser and results in a set of scalable Instantiation Graph (IG) data structures, stored in ZDB. *Instantiation Graph* is a data structure represented by a densely connected graph of semantically resolved objects representing elements

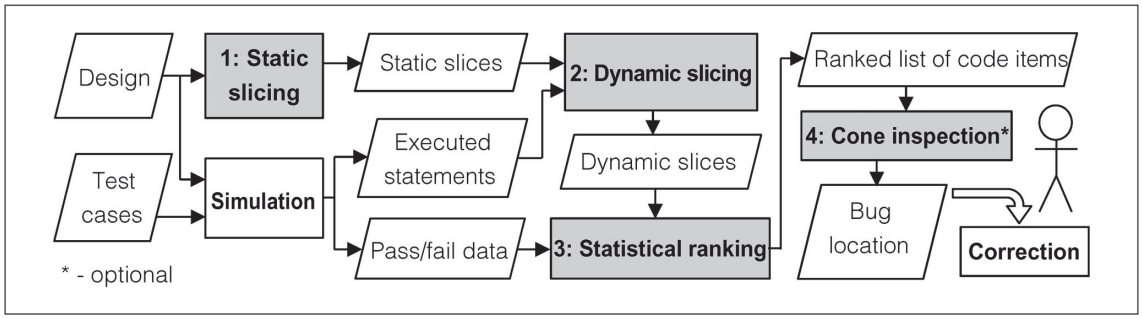


Figure 2. Statistical bug localization flow.

of the hardware design. IG is the basis for zamiaCAD applications. As demonstrated in [9] the framework is capable of handling very large industrial multicore designs consisting of tens of millions of VHDL code lines.

Statistical bug localization flow

The statistical bug localization method assumes that design verification has been performed and an erroneous behavior at observable outputs of the design has been detected. The method is based on four main phases: 1) staticslicing, 2) dynamic slicing, 3) statistical suspiciousness ranking of the HDL code items and, an optional, 4) cone inspection phase. First, the design is simulated in order to obtain the list of executed statements and information about passed and failed test cases from the test set. A test case is considered to be passed if the simulated output responses match with expected ones and it is regarded as failed otherwise. Then, *static slicing* computation is performed based on generating reference graphs. Subsequently, *dynamic slicing* reduces the debugging analysis to all the code items that actually affect the design's faulty behavior for a given test case. Finally, the *statistical suspiciousness ranking* assigns a *suspiciousness score* to each code item based on its presence in the dynamic slices and on the information of passed/failed test cases. Intuitively, if a codeitem occurs very frequently in executions revealing the error, it is very likely to contain a bug. The ranking is performed for the statement items in the HDL code. In order to reveal the bug locations more accurately, the suspiciousness ranking can be performed hierarchically considering also the branches and conditions that the highly ranked statements may have (see Details of hierarchical localization of one bug for the hierarchical localization of code items).

In this article, we consider debugging as a process of locating the failure, with the correction task being left to the designer. After the latter has received the ranked list of code items the following task is to localize the root cause of the erroneous behavior. Likely locations for bugs are in those code items having the highest suspiciousness scores in the list. Ina simple case the designer has to inspect code items at the top of the ranked list, whose score is higher than a preselected threshold value $S_{\text{threshold}}$. However, there exist cases where the statistical ranking does not directly pin-point the root location of the error. In those cases the *cone inspection* phase should be applied. Our case study shows that it is easy to locate the bug by activating depth-limited forward and backward cones from the signals included to the highest ranked items. This type of cone activation is supported by the zamiaCAD infrastructure of through-signal-assignments search. The study showed that only low depth cones (up to 1 level) starting from the signals of the highest ranked code item need to be inspected in practice. Figure 2 presents the statistical bug localization flow.

Motivational example

Consider the motivational design example shown in Figure 3 that presents a VHDL implementation of a signal chopper design named *chopper* [12]. The *chopper* design has three processes calculating four outputs representing different chops for the input signal SRC based on the design configuration by inputs INV and DUP. It is assumed that the design has five individual tests T1–T5 of

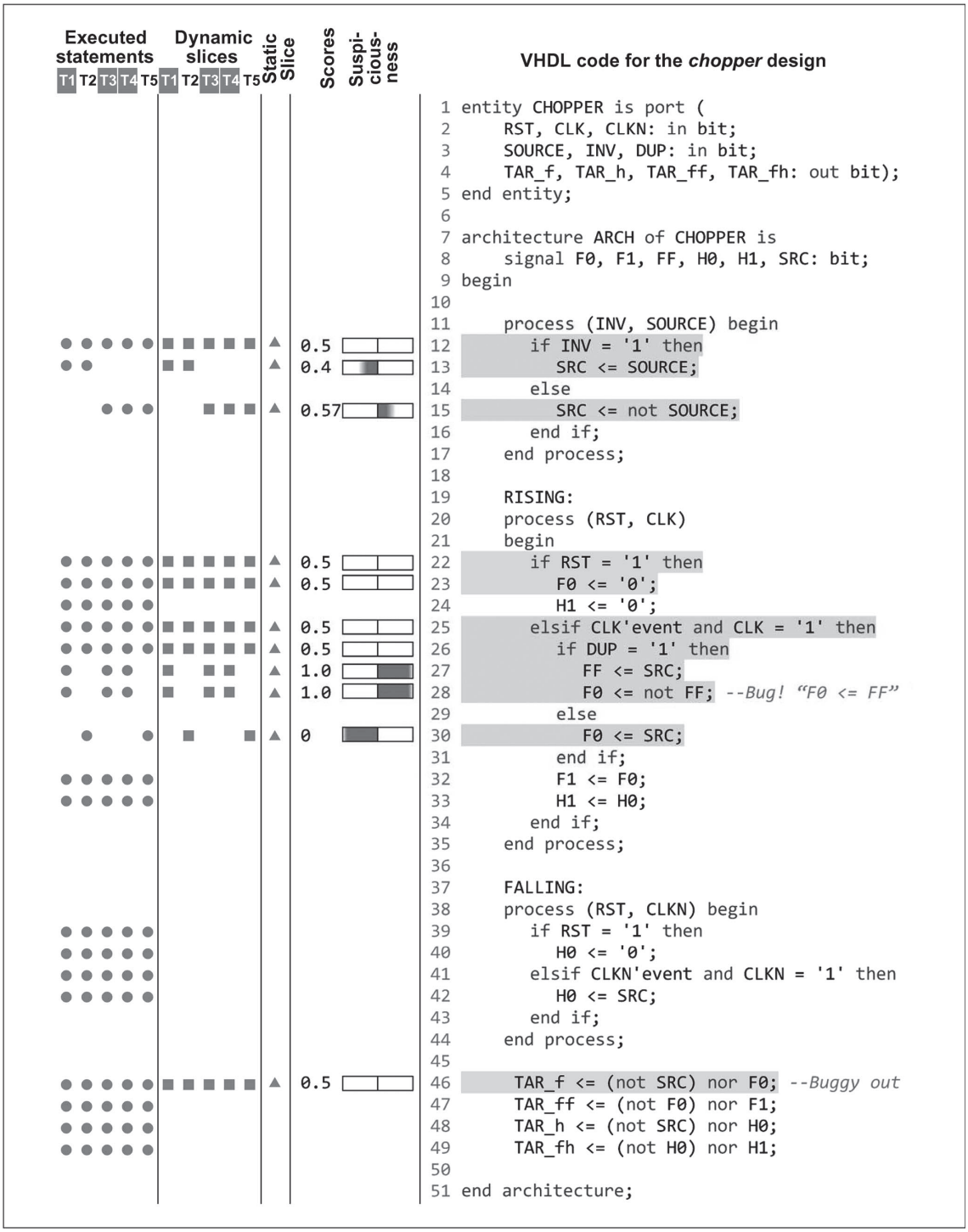


Figure 3. Bug localization on a motivational example.

varied length. The design has a bug on line 28 (i.e., "F0 < = not FF;"instead of "0 < = FF;"). Test cases T1, T3, and T4 are *failing tests*, while test cases T2 and T5 are *passing tests*, respectively. The faulty behavior of the design caused by the failing tests is observed at output TAR_f (assigned at line 46).

Static and dynamic slicing

The proposed bug localization approach uses dynamic slicing to reduce the analysis space, where static slicing serves a basis for dynamic slicing. The presence of concurrent constructs such as the ones found in HDLs versus sequential software languages makes static slice computation considerably more complicated [12]. zamiaCAD exploits its analytical model IG for this purpose [8]. Given the IG model, it is possible to perform a through-signal-assignment search both backward to find the dependencies and forward to find other signals and variables influenced by the signal. The resulting *reference graph* has the signals and variables at its nodes and the dependencies are expressed by directed edges. An example reference graph computed for the *chopper* design's output TAR_f is shown in Figure 4.

Given the reference graph, the HDL statements representing the signal and variable dependencies in its edges are collected into a set representing a *static slice* on the signal of interest. The column Static Slice in Figure 3 marks VHDL statements of a static slice on the TAR_f output by *triangles*. As a result in the *chopper* design example the entire process FALLING and a large part of other statements were excluded from the further analysis.

A subset of statements executed by simulation of a test case selected by static slice on a signal of interest is referred as *dynamic slice* of the design on this signal. In our approach we also consider information if the given test case has failed or passed for the dynamic slice. The column Executed Statements in Figure 3 marks the VHDL statements executed during design simulation by *circles*. The VHDL statements taking part in the dynamic slices are marked in the column Dynamic Slices by *rectangles*. Thus due to dynamic slicing the analysis space for the current example was reduced by 2.2× (42 covered statements in dynamic slices versus 92 statement executions by the diagnostic test).

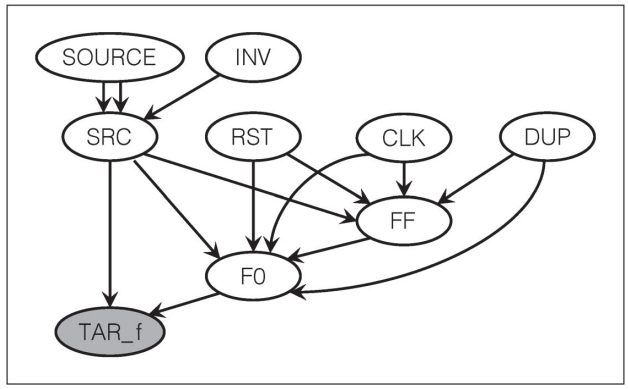


Figure 4. A reference graph based on backward through-signal-assignmentsearch on the signal TAR_f in the *chopper* design.

Statistical suspiciousness ranking

The list of bug location candidates is obtained by using statistical suspiciousness ranking on dynamic slices. Let $Passed(i)$ and $Failed(i)$ be the counts of passing and failing tests that covered the code item i in a dynamic slice, while *TotalPassed* and *TotalFailed* are the total numbers ofthe passing and failing tests in the complete diagnostic test set, respectively. Using this information, the *statistical suspiciousness ranking* assigns a score to each code item i using the following formula (1) (also used in [3]):

$$S(i) = \frac{\frac{Failed(i)}{TotalFailed}}{\frac{Passed(i)}{TotalPassed} + \frac{Failed(i)}{TotalFailed}}. \quad (1)$$

This score is a ratio between the number of statements executed during runs resulting in erroneous and correct behaviors. Intuitively, if a statement occurs very frequently in erroneous executions, it is very likely to containa bug.

The zamiaCAD environment visualizes by colors the suspiciousness level of the HDL code items (i.e., statements, branches, conditions) based on their scores $S(i)$.The score values are interpreted as follows:

- $S(i) = 1$—code item i is highly suspicious to contain, or to lead to the bug;
- $S(i) = 0$—code item i is above suspicion;
- $S(i) = S$—code item i cannot be emphasized by the analysis.

Here $0 < S_{\text{threshold}} < 1$ is a *suspiciousness threshold* specified by the designer and by default equal to

0.5. The *ranking* of code items is performed according to the score values starting from the highest. Code items without a score are either eliminated from the analysis by the static slice filter or not covered by the diagnostic test.

We apply the described ranking to statements and when necessary hierarchically to branches and conditions. The ROBSY processor case study has emphasized an important general category of design errors that are difficult to localize. They are bugs in complex condition expressions of conditional statements. E.g., Bug 1 in this case study is an erroneous comparison of one of the 35 conditions in a conditional assignment *when* of the ALU module. Localization of such bugs requires suspiciousness ranking of conditions following the same approach as described above for statements.

An example of applying the proposed suspiciousness ranking to the *chopper* design is demonstrated in Figure 3. Here the assignment statements at lines 27 and 28 are calculated as the most suspicious (score $S = 1.0$) and are assigned with the first rank. The statement at line 15 has score $S = 0.57$ and therefore a lower rank. The assignment statements at lines 13 and 30 have scores 0.4 and 0.0 correspondingly and, therefore, considered above suspicion and not assigned with suspiciousness ranks.

Cone inspection

The proposed approach considers an optional step of static slice based *cone inspection*. Generally, the automated suspiciousness ranking step should provide a small set of bug location candidates with actual bug location within this set. In rare cases, e.g., for particular locations in the design structure or for poor test suits, some bugs can stillget a low suspiciousness score and escape from the set of candidates. However, some related correct statements influenced by propagation of the design error effect can get high suspiciousness scores. In such cases, zamiaCAD offers infrastructure to build cones of influence of a limited depth (restricted static slices) from the signals involved into highly ranked suspected statements. Similarly to the unrestricted static slices exploited for the dynamic slicing step this procedure relies on through-signal-assignment search on IG model. This procedure selects a set of additional candidates that influence a correct but highly suspected code item.

We show the need for this optional cone inspection step in A case study, where the small original functional test used for diagnosis was not sufficient to localize all bugs by automated suspiciousness ranking. The study showed that only low depth cones (up to 1 level) starting from the signals of the highest ranked code item need to be inspected in practice. The task of analyzing this set of additional candidates and correcting the design is left to the designer.

A case study

As a case study, the proposed method was evaluated by debugging an industrial processor developed as a part of the ROBSY (Reconfigurable On Board self test SYstem) project. This custom processor follows a new test approach [10] to improve the fault coverage and reduce the test time of boards during the manufacturing process, and it is developed in cooperation with a major vendor of board testing equipment. The current implementation of the processor core contains 17 K lines of VHDL code. There are 481 direct signal assignment statements, 413 branches, and 1573 conditions, respectively.

In order to verify the correct functionality of the Instruction Set Architecture (ISA), a functional test was developed by the ROBSY design team. The functional test consists of a test program written in Assembler to test all the instructions supported by the processor. The test program is divided into test cases, where each test case is in charge of testing a specific instruction and setting the result register to a specific value that acts as a test case label (error code). During the test case execution, it is evaluated whether the values obtained in the registers, flags, etc. are as expected. Dedicated test cases target other functionality of the design such as interrupts and registers.

Set of documented design errors

The ROBSY design team has documented a set of VHDL coding bugs that occurred during the development:

- Bug 1) A *wrong register* is used as one of the conditions in a very long conditional expression (35 operators) inside a conditional signal assignment.
- Bug 2) An entire *conditional subexpression* (three operators) resides in the wrong branch of a

#	Stm. score	Bran. score	Cond. score	Line	Source code lines
			alu.vhd		
6	0.51			88	
5	0.55			104	
2	0.67			108	
1	0.80			110	
5	0.55			116	
5	0.55			127	
				...	
3	0.64	0.64		260	svFlag_new(0) <= '1' when afClass=cfClass_1
			0.64^{1}_{T}	261	and ((svOp_mux(cnD_w)=REG_SOURCE_DEST_IN(cnD_w)--add case
				262	and svOp_mux(cnD_w)/=svRes(cnD_w)
				263	and ((aCmd=cvCmd_ADD_R_R and c_en_ADD_R_R)
				264	or (aCmd=cvCmd_ADD_R_IMM and c_en_ADD_R_IMM)))
			0.51^{1}_{T}	265	or (svOp_mux(cnD_w)/=REG_SOURCE_DEST_IN(cnD_w)--sub case
					-- Bug: correct compar. between REG_SOURCE_DEST_IN and svRes
				266	and svOp_mux(cnD_w)/=svRes(cnD_w)
			0.69^{1}_{T}	267	and ((aCmd=cvCmd_SUB_R_R and c_en_SUB_R_R)
			0.55^{2}_{T}		
			0.75^{2}_{T}	268	or (aCmd=cvCmd_SUB_R_IMM and c_en_SUB_R_IMM)
			0.57^{1}_{T}	269	or (aCmd=cvCmd_CMP_R_R and c_en_CMP_R_R)
			0.55^{2}_{T}		
			0.51^{1}_{T}	270	or (aCmd=cvCmd_CMP_R_IMM and c_en_CMP_R_IMM)))
				271	or (REG_SOURCE_DEST_IN(cnD_w)/=svRes(cnD_w)--shift cases
			0.55^{1}_{T}	272	and ((aCmd=cvCmd_SHL_R and c_en_SHL_R)
			0.55^{2}_{T}	273	or (aCmd=cvCmd_SHR_R and c_en_SHR_R))))
5	0.55	0.55		274	else '0' when afClass=cfClass_1 --overflow reset
				275	and ((aCmd=cvCmd_ADD_R_R and c_en_ADD_R_R)
				276	or (aCmd=cvCmd_ADD_R_IMM and c_en_ADD_R_IMM)
				277	or (aCmd=cvCmd_SUB_R_R and c_en_SUB_R_R)
				278	or (aCmd=cvCmd_SUB_R_IMM and c_en_SUB_R_IMM)
				279	or (aCmd=cvCmd_CMP_R_R and c_en_CMP_R_R)
				280	or (aCmd=cvCmd_CMP_R_IMM and c_en_CMP_R_IMM)
				281	or (aCmd=cvCmd_SHL_R and c_en_SHL_R)
				282	or (aCmd=cvCmd_SHR_R and c_en_SHR_R))
NA	0.50			283	else svFlag(0);
			data_interface_mod.vhd		
2	0.67			155	
2	0.67			158	
			gprs_mod.vhd		
4	0.60			97	
			state_machine.vhd		
4	0.60			100	
4	0.60			123	
4	0.60			168	

Figure 5. Localization of Bug 1 in the ROBSY processor.

conditional signal assignment, which contains nine branches in total.

- Bug 3) Both, a *missing branch* and a *missing driver* in a short conditional signal assignment.
- Bug 4) A *wrong enumeration constant* is used in a comparison operation inside a conditional signal assignment.
- Bug 5) A *wrong driver* is used in a conditional signal assignment. More specifically, register *R* is not updated with its newly computed value typically stored in *R_next* or *R_new* signal. Instead, the register *R* is used as a driver for itself, which indicates an obvious copy-paste error.
- Bug 6) A *missing conditional subexpression* (three operators out of six required ones) in one of the four branches of a conditional signal assignment.
- Bug 7) *One bit* of a register is always and unconditionally set to "0". The whole code line to blame is unnecessary and incorrect.

Details of hierarchical localization of one bug

Figure 5 demonstrates the proposed *hierarchical localization* of *Bug 1*. The gray areas denote information omitted from the figure. First dynamic slices were generated for all of the test cases and then the statistical suspiciousness ranking was performed. This analysis resulted in 14 statement candidates in the design whose suspiciousness score S was above the suspiciousness threshold $S_{\text{threshold}} = 0.5$. The first column in the figure shows ranks of the suspected statements (six ranks in total) and column *Stm.score* shows their scores. Most of the statements with high scores were found in the `alu.vhd` file. The figure demonstrates a part of the actual VHDL code in the file for the conditional assignment of an overflow flag signal `svFlag_new(0)`. Bug 1 is located in the condition expression at line 266, where comparison is made between unintended signals. The automated localization procedure iteratively advises the designer to consider as bug location candidates the statements starting from the highest rank. As the fifth candidate it will advise the designer the statement at line 260 in `alu.vhd` (rankis 3, score $S = 0.64$). The hierarchical analysis will proceed with score computation of the branches of this statement (see column *Bran.score*)and suspiciousness scores of separate conditions related to these branch items. The ones that have score $S > 0.5$ are shown in column *Cond.score*. One of the highest scores in the expression has the logical operator *and* at line 267. One of its operands is actually the incorrect signal comparison documented as *Bug 1*.

In the current case study, we have split the original functional test (i.e., the Assembler program) into 28 independent test cases, each targeting a separate instruction. Each of the seven buggy versions of the processor was simulated with the resulted diagnostic test set.

Table 1 demonstrates the statistics of applying the proposed bug localization approach to all of the seven bugs. The second column depicts the ratio of failing versus passing test cases for these bugs. The third column in the table shows how many statements were proposed in total as bug location candidates by the statistical ranking step. The column also demonstrates these numbers in percentage of the total number of the statements which was 481. The fourth column shows the rank of the statement actually containing the bug. If ranking alone was not

Table 1 Bug localization on the ROBSY processor.

Bug data		The proposed automated localization					Manual debug
		Statistical Ranking		Cone inspection		Time (min)	
Bug name	Failed/Passed Test cases	Statements cand. / %	Located stm. rank	Cone dir. / depth	Added stm. cand.		Time
Bug 1	4 / 24	14 / 2.9%	3	-	-	2	4 hours
Bug 2	2 / 26	7 / 1.4%	1	-	-	2	2 hours
Bug 3	2 / 26	20 / 4%	3	-	-	2	4 hours
Bug 4	1 / 27	6 / 1.2%	(1)	fw / 1	21	2+(5)	4 hours
Bug 5	2 / 26	11 / 2.3%	1	-	-	2	2 hours
Bug 6	1 / 27	8 / 1.7%	(1)	bw / 1	13	2+(10)	5 hours
Bug 7	1 / 27	21 / 4.3%	(1)	fw / 1	10	2+(1)	1 hours

sufficient then the column shows in brackets the rank of the statement from which cone inspection was activated. Column five shows the direction (i.e., backward/forward) and the depth of the cone if cone inspection was required while column six shows the number of statements added as bug candidates by this step.

As the table shows the test suite was sufficient to automatically localize four of the seven bugs by the automated ranking step only. Pessimistic estimation of the candidates' count with the shown rank or higher that was necessary to check before the bug discovery is 5, 1, 12, and 4 for Bugs 1, 2, 3 and 5, respectively. Localization of the remaining three bugs required cone inspection as an addition step. The cones of a limited depth were generated by the through-signal-assignment reference search from the signals involved in the highly ranked assignment statements. In the current case study Bugs 4, 6, and 7 were present within the cones of depth 1 on the signals from the statements with the highest rank. These cones have added 21, 13, and 10 additional candidates as shown in column six.

The last two columns in Table 1 compare time required for bug localization by the proposed automated localization approach and conventional manual debug process. The time values for the manual process are reported by the ROBSY processor designers based on their experience with locating these bugs using commercial design environments. The time reported for the automated approach consists, first, of time spent for the statistical ranking step, which is mainly design elaboration and simulation of the 28test cases and constantly equals to 2 minutes, and second, of estimation of time spent for manual cone inspection (shown in brackets). The runtime required for the static slices and cones construction in zamiaCAD takes a fraction of second and can be neglected.

The presented case study showed that statistical bug localization is efficient and allows pin pointing causes of errors in large processor designs in a very accurate manner. The main contribution of this approach is, first, combining statistical analysis with HDL slicing, second, performing hierarchical localization in statements, branches and conditions of the code and, third, developing an efficient cone inspection technique in concurrent HDL descriptions. The open source zamiaCAD framework applied as the platform for the bug localization uses a highly scalable elaboration engine supporting industrial scale RTL designs. To probe further it is possible to repeat the described experiment with another processor design following a tutorial on the webpage http://zamiacad.sf.net. ■

Acknowledgment

Earlier versions of this article have been presented at the IEEE Latin American Test Workshop, 2012 and theIEEE International Workshop on Microprocessor Test and Verification, 2012.

References

[1] S. B. Park and S. Mitra, "IFRA: Post-silicon bug localization in processors," in *Proc. HLDVT*, 2009, pp. 154–159.

[2] W. E. Wong, V. Debroy, and B. Choi, "A family of code coverage-based heuristics for effective fault localization," *J. Syst. Software*, vol. 83, no. 2, pp. 188–208, 2010.

[3] J. A. Jones and M. J. Harrold, "Empirical evaluation of the Tarantula automatic fault-localization technique," in *Proc. Int. Conf. Autom. Softw. Eng.*, 2005, pp. 273–283.

[4] B. Peischl and F. Wotawa, "Automated source-level error localization in hardware designs," *Des. Test Comput.*, vol. 23, no. 1, pp. 8–19, 2006.

[5] T. Matsumoto, S. Ono, and M. Fujita, "An efficient method to localize and correct bugs in high-level designs using counter examples and potential dependence," in *Proc. IEEE/IFIP VLSI-SoC*, 2012, pp. 291–294.

[6] K.-H. Chang, I. Wagner, V. Bertacco, and I. L. Markov, "Automatic error diagnosis and correction for RTL designs," in *Proc. IWLS*, May 2007, pp. 106–113.

[7] S. Safarpour and A. Veneris, "Automated design debugging with abstraction and refinement," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* vol. 28, no. 10, pp. 1597–1608, Oct. 2009.

[8] zamiaCAD Framework Webpage. [Online]. Available: http://zamiaCAD.sf.net

[9] A. Tšepurov, G. Bartsch, R. Dorsch, M. Jenihhin, J. Raik, and V. Tihhomirov, "A scalable model based RTL framework zamiaCAD for static analysis," in *Proc. VLSI-SoC*, Santa Cruz, CA, USA, 2012.

[10] J. H. MezaEscobar, J. Sachsse, S. Ostendorff, and H. D. Wuttke, "Automatic generation of an FPGA based embedded test system for printed circuit board testing," in *Proc. LATW2012*, Quito, Ecuador, 2012, pp. 75–80.

[11] DIAMOND Project Web Page. [Online]. Available: http://fp7-diamond.eu

[12] E. M. Clarke, M. Fujita, S. P. Rajan, T. Reps, S. Shankar, and T. Teitelbaum, "Program slicing for VHDL," in *Proc. Charme 99*, Bad Herrenalb, Germany, Sep. 1999.

Maksim Jenihhin is senior research fellow at Tallinn University of Technology (TUT), Estonia. He received his PhD in computer engineering from TUT and his research interests include manufacturing test, functional verification, and debug of hardware and EDA. He is a Member of IEEE.

Anton Tšepurov is a PhD student at Tallinn University of Technology (TUT), Estonia. He received his MSc in computer engineering from TUT and his research interests include hardware modeling for functional verification and debug. He is a Student Member of IEEE.

Valentin Tihhomirov is a PhD student at Tallinn University of Technology (TUT), Estonia. He received his MSc in computer engineering from TUT and his research interests include hardware debug and CAD design.

Jaan Raik is a professor at Tallinn University of Technology (TUT), Estonia. He received his PhD in computer engineering from TUT and his research interests include high-level test pattern generation, verification, and automated debug of digital systems. He is a Member of IEEE.

Hanno Hantson is a PhD student at Tallinn University of Technology (TUT), Estonia. He received his MSc in computer engineering from TUT and his research interests include verification of digital systems and mutation analysis.

Raimund Ubar is a professor at Tallinn University of Technology (TUT) and Head of the Centre for Integrated Electronic Systems and Biomedical Engineering in Estonia. He received his PhD degree from the Bauman Technical University in Moscow, and his DSc degree from the Latvian Academy of Sciences. His scientific interests include computer science, diagnostics of technical systems, and application of embedded systems in biomedical engineering. He is a Golden Core member of the IEEE Computer Society.

Günter Bartsch is a freelance software developer and systems administrator. He received his computer science diploma from University of Stuttgart, Germany. His research and developer interests include compilers, EDA tools, electronics, and free software.

Jorge Hernan Meza Escobar is a scientific assistant and a PhD candidate at the Ilmenau University of Technology (IUT), Germany. His research interests include embedded processor design, board-level testing, and FPGA based testing. He has an Electronics Engineering degree from the Universidad del Valle, Colombia. He is a Member of the IEEE.

Heinz-Dietrich Wuttke is a senior researcher and lecturer at the Ilmenau University of Technology (IUT), Germany. He has a PhD in computer engineering from the IUT. His research and teaching interests include the design and validation of digital and parallel systems. He is a member of the German Informatics Society.

■ Direct questions and comments about this article to Maksim Jenihhin, Department of Computer Engineering, Tallinn University of Technology, Tallinn, Harjumaa 12618, Estonia; maksim@ati.ttu.ee.

Appendix V

Research paper V

Raik, Jaan; Repinski, Urmas; Hantson, Hanno; Jenihhin, Maksim; Di Guglielmo, Giuseppe; Pravadelli, Graziano; Fummi, Franco. "Combining Dynamic Slicing and Mutation Operators for ESL Correction". Proceedings of the 17th IEEE European Test Symposium, IEEE Computer Society Press, 2012, pp. 1–6.

Contributes to Section 4.3 of this Thesis. The author's contributions are: developing the mutation-based fault model in cooperation with Giuseppe Di Guglielmo from the Univesity of Verona during the author's stay in Verona, proposing an improved classification of faults.

Combining Dynamic Slicing and Mutation Operators for ESL Correction

Urmas Repinski, Hanno Hantson, Maksim Jenihhin, Jaan Raik,
Raimund Ubar
Department of Computer Engineering,
Tallinn University of Technology, Estonia
{urmas|hanno|maksim|jaan|raiub}@pld.ttu.ee

Giuseppe Di Guglielmo, Graziano Pravadelli, Franco Fummi
Department of Computer Science,
University of Verona, Italy
Email: {name.surname}@univr.it

Abstract

Verification is increasingly becoming the bottleneck in designing digital systems. In fact, most of the verification cycle is not spent on detecting the occurrences of errors but on debugging, consisting of locating and correcting the errors. However, automated design-error debug, especially at the system-level, has received far less attention than error detection. Current paper presents an automated approach to correcting system-level designs. We propose dynamic-slicing and location-ranking-based method for accurately pinpointing the error locations combined with a dedicated set of mutation operators for automatically proposing corrections to the errors. In order to validate the approach, experiments on the Siemens benchmark set have been carried out. The experiments show that the proposed method is able to correct three times more errors compared to the state-of-the-art mutation-based correction methods while examining fewer mutants.

1 Introduction

Increasing design costs are the main challenge facing the semiconductor community today. In particular, assuring the correctness of the electronic design, since the early stages of the design cycle, contributes to a major part of the problem [1]. However, *localization* and *correction* of design errors, i.e., *debug*, has received far less attention than *error detection*, both, in terms of research works and industrial tools introduced [2]. As a consequence, in industrial practice, debug is still a human-based activity that affects time-to-market [3].

The debugging approaches proposed in the past for logic and register-transfer (RT) levels, e.g. [16][17][18], cannot be applied for designs at electronic-system level (ESL) or their feedbacks are not sufficiently readable for this abstraction level. At ESL, designs are described in an algorithmic way with a high level of abstraction with respect to the final hardware implementation [1].

In this context, more effective methods for automating ESL debug are highly requested. Consider the case where a designer has a bug-affected ESL implementation. That is, an erroneous behavior of the implementation, with respect to the expected functionality, has been already (automatically) detected. Automated debug of errors consists of two steps: error localization and error correction. Error localization identifies the portion of the design responsible for the erroneous behavior, while error correction is responsible for locally modifying the functionality of the identified portion.

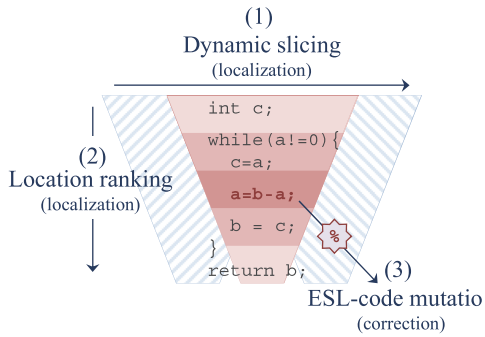


Figure 1. The proposed automated debugging approach for ESL designs relies on dynamic slicing and location ranking for error localization and code mutation for error correction.

For error localization, simulation-based [4][5][6][7][8][20] and formal approaches [9][10] are known. It is widely accepted that simulation-based techniques scale well with the design sizes, but are not exhaustive; while, formal techniques provide a high grade of confidence in the results, but are susceptible of the design complexity.

For error correction, error matching [12][13] and re-synthesis [14][15][16][17][18] have been investigated in literature. In particular, re-synthesis provides a correction which is represented as a partial truth table based on the stimuli under consideration. This kind of correction is not readable and cannot be easily understood and verified by the design engineer. Moreover, the resynthesized erroneous portion of the design is likely to fail when new stimuli will be added to the suite.

Recently, a more effective approach has been proposed for software debugging, which relies on simulation-based localization and error-matching correction [19]. The authors apply a diagnosis tool, i.e., Tarantula [20], for calculating suspiciousness scores for design portions and exploit mutation-based techniques to repair C and Java applications. However, the methodology is still affected by application size (in terms of number of mutations per lines of code) and targets mainly the control flow of software applications.

In this paper, we propose an approach, which, as opposed to [19], implies a dynamic slicing based methodology for error localization combined with a dedicated set of mutation operators for automated

978-1-4673-0697-3/12/$31.00 ©2012 IEEE

correction of system-level design errors. Different from [19] we consider corrections also in the data part of the design. However, the number of mutations required by the presented method is still smaller than in [19] due to more accurate localization achieved by dynamic slicing versus considering executed sentences applied in the Tarantula tool [20].

Figure 1 provides an overview of the proposed approach, which is based on three main iterative phases: *dynamic slicing*, *location ranking*, and *ESL-code mutation*.

The dynamic slicing reduces the debugging analysis to all the statements that actually affect the (erroneous) value of an output for a given input. In particular, it provides the relevant subset of the design statements where the location-ranking phase has to investigate for the possible cause of the error. As shown in Figure 1, a design may be thought of as a collection of "threads", each computing the value of an output. The dynamic slicing isolates the threads computing the erroneous outputs for the given inputs, i.e., the erroneous slice.

Then, the location ranking assigns a score to each statement in the erroneous slices. This score is the ratio between the number of statements executed during runs resulting in erroneous and correct behaviors. Intuitively, if a statement occurs very frequently in erroneous executions, it very likely contains a bug. In Figure 1, a more intense hue is associated with highly scored statements.

Finally, the ESL-code mutation addresses the correction of statements highly ranked as sources of the erroneous behavior. Typically, mutation techniques are used in testing for modeling realistic faults on correct design; in the adopted error-correction methodology, the mutation of a bug-affected design may result in a realistic correction.

A final observation is necessary on the golden model, i.e., the reference behavior of the ESL design during the debug phase. In the adopted simulation-based approach, the golden model may be indifferently (i) industrial test cases, i.e., input stimuli and expected results, specified by module designers, (ii) input stimuli and assertions supplied with the design or (iii) results obtained by abstract or different reference, e.g., executable-UML models and alternative implementations.

The main contributions of the present work are:

- the development of an accurate-localization methodology of bugs based on location ranking and dynamic slicing that scale well with design sizes;
- the definition of a dedicated set of mutation operators to model realistic errors in ESL designs;
- the development of a readable error-matching-based correction for system-level designs.

As experiments show, our approach provides three times higher success rate in terms of fixed errors in comparison to the results published in [10] and [19], while examining significantly fewer mutants.

The rest of the paper is organized as follows. Section 2 provides an overview of the related works. Section 3 describes the methodology for design-error localization based on dynamic slicing. Section 4 introduces the mutation-based-correction methodology for system-level design errors. Section 5 provides experimental results validating the ESL-repair approach on the realistic buggy code versions represented in the Siemens benchmark suite [10]. Finally, conclusions are drawn in Section 6.

2 Related works

In debugging, the error localization is considered the most time expensive activity and its quality affects the following (manual or automatic) correction phase [21]. In manual error localization, engineers run the design with some input stimuli till they observe a failure; then, they iteratively place breakpoints, analyze the system status, and backtrack to the error origin using a source-level debugger, e.g., GNU GDB [22].

On the other hand, automatic error localization is based on different methodologies. In particular, they may be simulation-based and use coverage information [6][7][20], binary search [8], and statistical analysis [4][5]. As well, formal approaches for error localization exist that are very effective but may suffer the state-explosion of the underlying solver [9][10]. Of all these solutions, the Tarantula [20] coverage-based approach has been proven suitable for real-world designs. In the present work, we provide an improvement for error localization, which significantly reduces the overhead of the error-correction phase based on ESL-code mutation.

After an error is detected and localized, it should be corrected. Design-error correction for combinational circuits has been thoroughly studied for decades. There exist, both, error-matching-based [12][10][13] and re-synthesis [14] approaches. There have also been attempts to generalize the above methods for design-error correction of sequential circuits [14][15]. In particular, the SAT-based correction and re-synthesis approach developed by Smith et al. [16] has been extended to higher abstraction levels such as register-transfer level [17][18]. The re-synthesis approach for high-level design-error correction has two main limitations. The correction is *not readable* and thus cannot be checked by the designer. Moreover, the *correction is limited* to the set of used stimuli: this is due to the logic optimization freedom created by the partial truth table of the portion to be repaired.

Finally, in [10] a symbolic-simulation-based approach is proposed for both error correction and localization in ESL designs described as C programs. All the reasoning is done with a Satisfiability Modulo Theory (SMT) solver [11], thus it can be classified as a formal method. In particular, the approach performs the error correction by using approximation heuristics and a template-based methodology, which gives readable corrections. In the experimental-result section, we provide

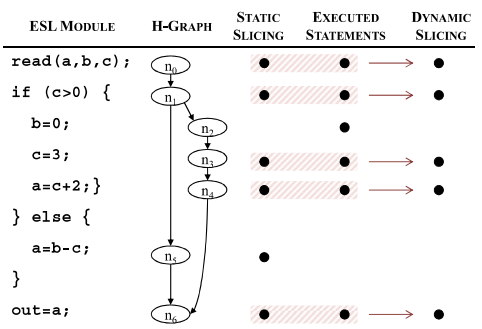


Figure 2. The ESL description is modeled as a flowgraph, i.e., hammock graph. Simulation and slicing are performed on the model representation.

comparisons of our approach and [10], showing better correction capability and preserving correction readability.

3 Dynamic Slicing for Error Localization

This section describes the proposed error localization methodology, which is based on dynamic slicing. In particular, Section 3.1 describes the adopted model for ESL descriptions; Section 3.2 summarizes the dynamic slicing methodology; finally, Section 3.3 describes the methodology for ranking the error location. The accurate localization of errors permits to significantly reduce the overhead of the following mutation-based correction phase.

3.1 Flowgraph based modeling of the algorithmic level

In this paper, we consider automated debugging of ESL designs. In order to formally represent the ESL algorithmic descriptions we have chosen the flowgraph model as an underlying model. In such flowgraph, there is a one-to-one correspondence between the program statements and nodes and edges represent the control flow of the program. More precisely, the model representation is a special case of flowgraph known as the *hammock graph* [25], which was proposed for program slicing in [26]. Hammock graph is defined as follows.

Definition 1: A *hammock graph* is a structure $H=<N, E, n_0, n_e>$, where N is a set of nodes, E is a set of edges in $N\times N$, n_0 is the *initial node* and n_e is the *end node*. If (n, m) is in E then n is an *immediate predecessor* of m and m is an *immediate successor* of n. A *path* from a node n_1 to a node n_2 is a list of nodes $p_0, p_1, ..., p_k$ such that $p_0 = n_1$, $p_k = n_2$, and for all i, $1 \le i \le k-1$, (p_i, p_{i+1}) is in E. There is a path from n_0 to all other nodes in N. From all nodes of N, excluding n_e, there is a path to n_e.

Figure 2 presents a simple ESL functionality in C language, i.e., column ESL MODULE, and the corresponding flowgraph H, i.e., column H-GRAPH. In the following, we introduce some definitions in order to explain the slicing process on flowgraph structures.

3.2 Model slicing

We apply dynamic slicing in order to narrow the search of the causes of design errors in algorithmic descriptions. In this Section, we provide a brief introduction to slicing techniques and explain how we implement dynamic slicing in design error localization.

Program slicing [26] is a technique for extracting portions of a program affecting a selected set of variables of interest. By focusing on the computation of only few variables the slicing process can be used to discard portions of the program, which cannot influence these variables, thereby reducing the size of the program. The reduced program is called a slice.

Slices reproduce a projection from the behavior of the initial program. This projection represents the values of certain variables as seen at certain statements.

Definition 2: A *slicing criterion* of a program P is a tuple (x,V), where x is a statement in P and V is a subset of the variables in P.

Informally, given a slicing criterion $C = (x, V)$, a static program slice S consists of all statements in program P that may affect the value of $v \in V$ for a set of all possible inputs at the point of interest, i.e., at the statement x. Static slices are computed by finding consecutive sets of indirectly relevant statements, according to data and control dependencies. Unfortunately, the size of the slices so defined may approach that of the original program. Indeed, static slicing preserves the behavior of the original program for *all the possible* input values. In this case, the usefulness of the slices in debugging tends to diminish as the size of the slices increases.

In [27], Korel and Laski introduced a more accurate slicing technique, i.e., *dynamic slicing*. Dynamic slicing provides more narrow slices, preserving the behavior of the original program and consisting of only the statements that influence the value of a variable for *a given input*.

Figure 2 illustrates the concepts of static and dynamic slicing applied to the flowgraph representation of a ESL functionality. In particular, the Figure reports an intuitive correlation between static slicing, execution trace, and dynamic slicing. Let us consider, for example, the slicing criterion $C = (n_6, \{out\})$. In this case the n_6 is the end node n_e of the hammock graph. The black dots in the column STATIC SLICING indicate the statements included into the slice in case of static slicing. These mark the statements that are needed in order to calculate the value of variable a at node n_6. As we can see the node n_2 is excluded from the slice because the statement b=0 is not necessary for calculating the value of variable *out* at node n_6.

The column DYNAMIC SLICING refines that slice according to the execution trace obtained with actual value assignments. Assuming that variables get

assignments a=2, b=4 and c=7, we obtain the slice shown in the last column of Figure 2. The *else* branch of the condition is not activated by these input values and therefore the respective statement are not included into the slice. The column EXECUTED STATEMENTS shows all the statements that were executed in current trace with the given input assignments. As we can see, the statements occurring in the dynamically-computed slice are a proper subset of the statements in the statically-computed slice and execution trace. This narrows the search space of the following step for ranking the error locations.

3.3 Slicing-based error localization

In current paper, we consider a design-error-localization approach, where ESL implementations fail on some of the given test cases. The error localization relies on error detection results. The mechanisms of the latter are out of scope of this paper and may involve for instance the golden output responses specified by the test cases, assertions supplied with the test environment or results obtained from a analyzing the specification (e.g. UML, SW program, etc.).

The error localization proposed in current paper is based on calculating the dynamic slices for all the observable outputs of the system with all the test cases. Depending on whether an output response obtained by a given slice is correct or not, the slice is marked as a passed or failed one, respectively. Then, a statistical and coverage-based approach is implemented assigning score to flowgraph nodes based on the number of times they were included into failed slices with respect to the number of times they occur in the previous executions. Finally, the flowgraph nodes are ranked according to this score, referred to as the *suspiciousness score*.

In details, the error ranking and localization takes place as follows. Let T be a test suite consisting of test cases t_i for verifying the functionality of the ESL description. Let H be the flowgraph associated with the description. Let y_j be the observable output variables of the design. Finally, let the nodes n_j of H be the respective nodes were value assignments to y_j are made. Over each test case t_i and, in turn, over each observable output variable y_j we generate a dynamic slice d_{ij} according to the values of current test case t_i and a slicing criterion $C = (x_j, \{y_j\})$, where x_j is the statement at the flowgraph node n_j.

If y_j resulted in a correct value at test case t_i, then the dynamic slice d_{ij} is included into the set of passed slices D_{PASSED}. Otherwise, it is included to the failed slices, i.e. $d_{ij} \in D_{FAILED}$. Each node n_k of flowgraph H gets a score according to the number of times c_{FAILED} it is included into the set of failed slices D_{FAILED} and the number of times c_{PASSED} it is included into the set of passed ones, i.e. D_{PASSED}. This score of suspiciousness is calculated as follows:

Table 1. List of mutation operators for correction

MUTATION OPERATOR	C OPERATORS/EXAMPLES
AOR (arithmetic operator replacement)	+, -, *, /, %
ROR (relational operator replacement)	==, !=, >, <, >=, <=
LCR (logical connector replacement)	&&, ‖
ASOR (assignment operator replacement)	+=, -=, *=, /=, %=, =
UOR (unary operator replacement)	+, -, ~, !
Bitwise operator replacement	<<, >>, &, |, ^
Bitwise assignment operator replacement	<<=, >>=, &=, |=, ^=
Increment/decrement operator replacement	*x*++, ++*x*, *x*--, --*x*
Number mutation (decimal digit replacement in integers, floats and array indexes)	0...9
Constant replacement unary minus/ unary plus/ zero	+*C*, 0, -*C*

$$suspiciousness(n_k) = \frac{c_{FAILED}}{c_{FAILED} + c_{PASSED}}$$

The nodes n_k are ranked according to the suspiciousness score with more probable candidates for error correction having higher score values. This ranking is used for selecting statements to be corrected by the mutation-based methodology presented in the following section.

4 Mutation-Based Error Correction

Mutation is a process, where syntactically-correct functional changes are inserted into the program [28]. Traditionally mutations are performed by perturbing the behavior of the program in order to see if the test suite is able to detect the difference between the original program and the mutated versions. The effectiveness of the test suite is then measured by computing the percentage of detected, or *killed*, mutations.

In this paper, we apply mutation operators for correcting erroneous circuits. The goal is to develop an error-matching based correction approach, which would be capable of modeling realistic design errors. Moreover, it is crucial to select a limited number of mutation operators, because the perturbation and simulation of erroneous design implementations with a large number of error locations and mutant operators would become prohibitively time-consuming.

Table 1 presents the set of ESL-mutation operators which were implemented in the error-matching based correction method developed in this paper. Since we target ESL descriptions in C language, we only focus on algorithmic aspects of the description and do not consider software-specific constructs and related errors, such as dynamic-memory allocation, pointer arithmetic, and file

Table 2. Characteristics of the Siemens benchmarks.

DESIGN	LOC	TEST-CASE #	FAULTY-VERSION #
replace	507	5542	32
schedule	397	2650	9
schedule2	299	2710	9
tcas	174	1608	41
tot_info	398	1052	23
print_tokens	539	4130	7
print_tokens2	489	4115	10

I/O. This permits to reduce the overhead of the code-mutation phase and address only system-level issues.

In particular, the mutation operators include replacement of C language operators, which have been divided into several groups: arithmetic operators, relational operators, assignment operators, unary operators, etc. In addition, number mutations are performed by replacing each decimal digit in the numeric values one-by-one with other decimal values. This includes both, integer and floating point numbers and it covers also the array indexes. Also, constants are mutated by inserting unary operators + and – as well as replaced by zero.

Figure 3 explains the mutation-based correction process. Subsequent to the error location step described in Section 3, which ranks the statements of the program, the suspected error locations are iteratively tried according to their rank. The operators in the statements are, in turn, iteratively substituted by mutation operators, i.e., valid operators from the same category. In other words, replacing arithmetic operators by arithmetic operators, relational operators by relational ones etc. These iterations stop when the simulation result confirms that the mutated program provides output responses equal to the golden output responses, in other words, a correction has been found. Otherwise the process continues until there exist untried error locations and/or mutant operators, or when a user-specified time limit is reached.

The mutation-based correction method proposed in this paper is an error-matching approach. Error-matching is known to have the limitation that it is generally not capable of fixing errors that are not included to the model. On the other hand, the mutation-based error-matching provides easy-to-read corrections of system-level descriptions. Moreover, our experiments show that the mutation-based approach can fix some of the not modeled errors by proposing alternative but equivalent fixes.

5 Experimental Results

The proposed debugging approach has been implemented as a module of a larger tool, i.e., FoREnSiC [24], which also features formal and semi-formal approaches for debugging of ESL design [10]. Our framework supports debugging of algorithmic descriptions of hardware in C language. In order to evaluate the proposed method, experiments on Siemens

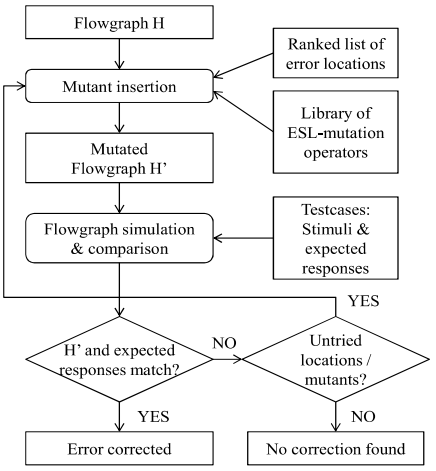


Figure 3. The mutation-based error correction flow.

benchmark suite [23] comparing it to a recently published formal [10] and dynamic [19] technique were carried out. We applied the front-end of FoREnSiC for generating the flowgraph models for the C language designs.

In Table 2, the main characteristics of the benchmark circuits are presented. Column LOC shows the number of lines of code for the corresponding C designs; column TEST-CASE # shows the number of test cases for the design, which include both failing test stimuli and passing stimuli; finally, column FAULTY-VERSION # shows the number of faulty versions of the benchmark programs. We excluded one faulty version from benchmark *schedule2* because the design error did not result in any test case failure making the correction process meaningless.

In Table 3, the results of the design error correction experiments are presented. Current method is compared to two recently published methods: a symbolic-simulation-based method [10] and a mutation-based method [19]. For each methodology, columns # FIXED show the number of corrected faulty model versions and Columns % FIXED show the percentage of corrected models from the total number of faulty model versions.

As it can be seen from the table, the proposed approach clearly outperforms [10], where only 8 faulty versions (out of 41) of *tcas* design are analyzed. The approach in [10] is able to correct 7 out of these 8 faulty versions, whereas our approach corrects all 8. Furthermore, due to the underlying solver, the formal approach [10] is only able to model the designs whose bit-width is reduced from 32 to 8 bits.

With respect to [19], current method increases the percentage of successful corrections from 16.0% to 50.3%. Thus, the rate of corrections is increased by the factor of three.

It is important to stress that the increase in successful fixes does not come at the expense of more mutants to be

Table 3. Design error repair experiments.

DESIGN	FMCAD'11 [10]		STVV'10 [19]		CURRENT METHOD		
	# FIXED	% FIXED	# FIXED	% FIXED	# FIXED	% FIXED	MUTANTS EXAMINED
replace	-	-	3	9.4	**12**	**37.5**	855.2
schedule	-	-	0	0.0	**2**	**22.2**	188.0
schedule2	-	-	1	11.1	**3**	**33.3**	460.7
tcas	7	17.1	9	22.0	**26**	**63.4**	131.1
tot_info	-	-	8	34.8	**15**	**65.2**	781.3
print_tokens	-	-	0	0.0	**1**	**14.3**	825.0
print_tokens2	-	-	0	0.0	**7**	**70.0**	952.3
Total:		*N/A*		*16.0*		***50.4***	*599.1*

considered. The last column of Table 3 shows the localization accuracy in terms of the average number of examined mutants per design error. In fact, this number is 599.1, which is even slightly less than 642 mutants in average obtained in [19].

The significant increase in successful corrections with respect to [19] is due to the selection of mutation operators, which are not limited to control flow errors. The run-time advantages in terms of the number of mutants examined comes partly from the more accurate diagnosis method based on dynamic slicing and location ranking.

6 Conclusions

The paper presents a method for correcting design errors in algorithmic descriptions of system-level hardware. The method applies dynamic slicing and location ranking to accurately pinpoint the error locations and combines it with a dedicated set of ESL-mutation operators for automatically proposing fixes to the errors. In order to validate the approach, experiments on the Siemens benchmarks have been carried out. The experiments show that the proposed method is able to repair three times more errors than previously achievable by mutation-based repair while examining fewer mutants. In addition, the method clearly outperforms a recent formal correction approach.

Acknowledgements

The work has been partly supported by European Commission FP7-ICT-2009-4-248613 DIAMOND project, by Research Centre CEBE funded by European Union through the European Structural Funds, by Estonian Science Foundation grants 9429 and 8478 and by Estonian Academy of Security Sciences.

References

[1] B. Bailey, G. E. Martin, and A. Piziali, "ESL design and verification: a prescription for electronic system-level methodology," Morgan Kaufmann Publisher, 2007.

[2] F. Rogin and R. Drechsler, "Debugging at the electronic system level," Springer Verlag, 2010.

[3] R. S. Pressman, "Software Engineering – A Practitioner's Approach," McGraw Hill, 1992.

[4] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 15-26, 2005.

[5] G. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff, "Statistical debugging: A hypothesis testing-based approach," *IEEE Trans. on Software Engineering*, vol. 32, no. 10, pp. 831-848, 2006.

[6] W. E. Wong, V. Debroy, and B. Choi, "A family of code coverage-based heuristics for effective fault localization," *Journal of Systems and Software*, vol.83, no.2, pp.188-208, 2010.

[7] W. E. Wong and Y. Qi, "BP neural network-based effective fault localization," International Journal of Software Engineering and Knowledge Engineering, vol. 19, no. 4, pp. 573-597, 2009.

[8] H. Cleve and A. Zeller, "Locating causes of program failures," *Proc. of Int. Conf. on Software Engineering*, pp. 342-351, 2005.

[9] S. Staber, B. Jobstmann, and R. Bloem, "Finding and fixing faults," *Proc. of Conference on Correct Hardware Design and Verification Methods*, pp. 35-49, 2005.

[10] R. Konighofer and R. Bloem, "Automated error localization and correction for imperative programs," *Proc. of Formal Methods in Computer Aided Design*, pp. 91-100, 2011.

[11] L. De Moura and N, Bjorner, "Satisfiability modulo theories: An appetizer," *Formal Methods: Foundations and Applications*, pp. 23-36, 2009.

[12] J. C. Madre, O. Coudert, and J. P. Billon, "Automating the Diagnosis and the Rectification of Design Errors with PRIAM," *Proc. of International Conference on CAD*, 1989, pp. 30-33.

[13] M. S. Abadir, J. Ferguson, and T. E. Kirkland, "Logic design verification via test generation," *IEEE Transactions on Computer-Aided Design*, vol. 7, no. 1, 1988.

[14] Ali, M. F., Safarpour, S., Veneris, A., Abadir, M. S., and Drechsler, R. Post-verification debugging of hierarchical designs. *In Proceedings of ICCAD*, pp. 871-876. 2005.

[15] A. Wahba, D. Borrione. Design error diagnosis in sequential circuits. *Lecture Notes In Computer Science*; Vol. 987, pp. 171 – 188, Springer, 1995.

[16] A. Smith, A. Veneris and A. Viglas, "Design Diagnosis Using Boolean Satisfiability", *Proc. Asia and South Pacific Design Automation Conference (ASPDAC)*, 2004, pp. 218-223.

[17] Kai-hui Chang, et al., "Automatic Error Diagnosis and Correction for RTL Designs", *Proc. High-Level Design and Validation Workshop (HLDVT)*, Irvine, CA, November 2007.

[18] Kai-hui Chang; Markov, I.L.; Bertacco, V.; , "Fixing Design Errors With Counterexamples and Resynthesis," *IEEE Trans. on CAD of ICs and Systems,* , vol.27, no.1, pp.184-188, Jan. 2008

[19] V. Debroy, W. E. Wong, "Using Mutation to Automatically Suggest Fixes for Faulty Programs", *Proc. of Int. Conf. on Software Testing, Verification and Validation*, 2010, pp. 65-74.

[20] J. A. Jones and M. J. Harrold, "Empirical evaluation of the Tarantula automatic fault-localization technique," *Proc. Int. Conf. on Automated Software Engineering*, pp. 273-283, 2005.

[21] I. Vessey, "Expertise in debugging computer programs," International *Journal of Man-Machine Studies: A Process Analysis*, vol. 23, no. 5, pp. 459-494, 1985.

[22] R. M. Stallman and R. H. Pesch, "Using GDB: A guide to the GNU source-level debugger," Free Software Foundation, 1991.

[23] Siemens benchmark suite: http://pleuma.cc.gatech.edu/aristotle/Tools/subjects/

[24] DIAMOND project website: http://www.fp7-diamond.eu/

[25] V.N. Kas"janov, "Distinguishing Hammocks in a Directed Graph", *Soviet Math. Doklady*, vol. 16, no. 5, pp. 448-450, 1975.

[26] M. Weiser, Program slicing, *IEEE Transactions on Software Engineering*, vol. 10, no. 4, 1984, pp. 352–357.

[27] B. Korel and J. Laski, Dynamic program slicing. *Information Processing Letters*, vol. 29, no. 3, 1988, pp. 155-163.

[28] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. "Hints on Test Data Selection: Help for the Practicing Programmer". *IEEE Computer*, pp. vol. 11, Issue 4, 34–41, 1978.

DISSERTATIONS DEFENDED AT TALLINN UNIVERSITY OF TECHNOLOGY ON *INFORMATICS AND SYSTEM ENGINEERING*

1. **Lea Elmik**. Informational Modelling of a Communication Office. 1992.

2. **Kalle Tammemäe**. Control Intensive Digital System Synthesis. 1997.

3. **Eerik Lossmann**. Complex Signal Classification Algorithms, Based on the Third-Order Statistical Models. 1999.

4. **Kaido Kikkas**. Using the Internet in Rehabilitation of People with Mobility Impairments – Case Studies and Views from Estonia. 1999.

5. **Nazmun Nahar**. Global Electronic Commerce Process: Business-to-Business. 1999.

6. **Jevgeni Riipulk**. Microwave Radiometry for Medical Applications. 2000.

7. **Alar Kuusik**. Compact Smart Home Systems: Design and Verification of Cost Effective Hardware Solutions. 2001.

8. **Jaan Raik**. Hierarchical Test Generation for Digital Circuits Represented by Decision Diagrams. 2001.

9. **Andri Riid**. Transparent Fuzzy Systems: Model and Control. 2002.

10. **Marina Brik**. Investigation and Development of Test Generation Methods for Control Part of Digital Systems. 2002.

11. **Raul Land**. Synchronous Approximation and Processing of Sampled Data Signals. 2002.

12. **Ants Ronk**. An Extended Block-Adaptive Fourier Analyser for Analysis and Reproduction of Periodic Components of Band-Limited Discrete-Time Signals. 2002.

13. **Toivo Paavle**. System Level Modeling of the Phase Locked Loops: Behavioral Analysis and Parameterization. 2003.

14. **Irina Astrova**. On Integration of Object-Oriented Applications with Relational Databases. 2003.

15. **Kuldar Taveter**. A Multi-Perspective Methodology for Agent-Oriented Business Modelling and Simulation. 2004.

16. **Taivo Kangilaski**. Eesti Energia käiduhaldussüsteem. 2004.

17. **Artur Jutman**. Selected Issues of Modeling, Verification and Testing of Digital Systems. 2004.

18. **Ander Tenno**. Simulation and Estimation of Electro-Chemical Processes in Maintenance-Free Batteries with Fixed Electrolyte. 2004.

19. **Oleg Korolkov**. Formation of Diffusion Welded Al Contacts to Semiconductor Silicon. 2004.

20. **Risto Vaarandi**. Tools and Techniques for Event Log Analysis. 2005.

21. **Marko Koort**. Transmitter Power Control in Wireless Communication Systems. 2005.

22. **Raul Savimaa**. Modelling Emergent Behaviour of Organizations. Time-Aware, UML and Agent Based Approach. 2005.

23. **Raido Kurel**. Investigation of Electrical Characteristics of SiC Based Complementary JBS Structures. 2005.

24. **Rainer Taniloo**. Ökonoomsete negatiivse diferentsiaaltakistusega astmete ja elementide disainimine ja optimeerimine. 2005.

25. **Pauli Lallo**. Adaptive Secure Data Transmission Method for OSI Level I. 2005.

26. **Deniss Kumlander**. Some Practical Algorithms to Solve the Maximum Clique Problem. 2005.

27. **Tarmo Veskioja**. Stable Marriage Problem and College Admission. 2005.

28. **Elena Fomina**. Low Power Finite State Machine Synthesis. 2005.

29. **Eero Ivask**. Digital Test in WEB-Based Environment 2006.

30. **Виктор Войтович**. Разработка технологий выращивания из жидкой фазы эпитаксиальных структур арсенида галлия с высоковольтным p-n переходом и изготовления диодов на их основе. 2006.

31. **Tanel Alumäe**. Methods for Estonian Large Vocabulary Speech Recognition. 2006.

32. **Erki Eessaar**. Relational and Object-Relational Database Management Systems as Platforms for Managing Softwareengineering Artefacts. 2006.

33. **Rauno Gordon**. Modelling of Cardiac Dynamics and Intracardiac Bio-impedance. 2007.

34. **Madis Listak**. A Task-Oriented Design of a Biologically Inspired Underwater Robot. 2007.

35. **Elmet Orasson**. Hybrid Built-in Self-Test. Methods and Tools for Analysis and Optimization of BIST. 2007.

36. **Eduard Petlenkov**. Neural Networks Based Identification and Control of Nonlinear Systems: ANARX Model Based Approach. 2007.

37. **Toomas Kirt**. Concept Formation in Exploratory Data Analysis: Case Studies of Linguistic and Banking Data. 2007.

38. **Juhan-Peep Ernits**. Two State Space Reduction Techniques for Explicit State Model Checking. 2007.

39. **Innar Liiv**. Pattern Discovery Using Seriation and Matrix Reordering: A Unified View, Extensions and an Application to Inventory Management. 2008.

40. **Andrei Pokatilov**. Development of National Standard for Voltage Unit Based on Solid-State References. 2008.

41. **Karin Lindroos**. Mapping Social Structures by Formal Non-Linear Information Processing Methods: Case Studies of Estonian Islands Environments. 2008.

42. **Maksim Jenihhin**. Simulation-Based Hardware Verification with High-Level Decision Diagrams. 2008.

43. **Ando Saabas**. Logics for Low-Level Code and Proof-Preserving Program Transformations. 2008.

44. **Ilja Tšahhirov**. Security Protocols Analysis in the Computational Model – Dependency Flow Graphs-Based Approach. 2008.

45. **Toomas Ruuben**. Wideband Digital Beamforming in Sonar Systems. 2009.

46. **Sergei Devadze**. Fault Simulation of Digital Systems. 2009.

47. **Andrei Krivošei**. Model Based Method for Adaptive Decomposition of the Thoracic Bio-Impedance Variations into Cardiac and Respiratory Components. 2009.

48. **Vineeth Govind**. DfT-Based External Test and Diagnosis of Mesh-like Networks on Chips. 2009.

49. **Andres Kull**. Model-Based Testing of Reactive Systems. 2009.

50. **Ants Torim**. Formal Concepts in the Theory of Monotone Systems. 2009.

51. **Erika Matsak**. Discovering Logical Constructs from Estonian Children Language. 2009.

52. **Paul Annus**. Multichannel Bioimpedance Spectroscopy: Instrumentation Methods and Design Principles. 2009.

53. **Maris Tõnso**. Computer Algebra Tools for Modelling, Analysis and Synthesis for Nonlinear Control Systems. 2010.

54. **Aivo Jürgenson**. Efficient Semantics of Parallel and Serial Models of Attack Trees. 2010.

55. **Erkki Joasoon**. The Tactile Feedback Device for Multi-Touch User Interfaces. 2010.

56. **Jürgo-Sören Preden**. Enhancing Situation – Awareness Cognition and Reasoning of Ad-Hoc Network Agents. 2010.

57. **Pavel Grigorenko**. Higher-Order Attribute Semantics of Flat Languages. 2010.

58. **Anna Rannaste**. Hierarcical Test Pattern Generation and Untestability Identification Techniques for Synchronous Sequential Circuits. 2010.

59. **Sergei Strik**. Battery Charging and Full-Featured Battery Charger Integrated Circuit for Portable Applications. 2011.

60. **Rain Ottis**. A Systematic Approach to Offensive Volunteer Cyber Militia. 2011.

61. **Natalja Sleptšuk**. Investigation of the Intermediate Layer in the Metal-Silicon Carbide Contact Obtained by Diffusion Welding. 2011.

62. **Martin Jaanus**. The Interactive Learning Environment for Mobile Laboratories. 2011.

63. **Argo Kasemaa**. Analog Front End Components for Bio-Impedance Measurement: Current Source Design and Implementation. 2011.

64. **Kenneth Geers**. Strategic Cyber Security: Evaluating Nation-State Cyber Attack Mitigation Strategies. 2011.

65. **Riina Maigre**. Composition of Web Services on Large Service Models. 2011.

66. **Helena Kruus**. Optimization of Built-in Self-Test in Digital Systems. 2011.

67. **Gunnar Piho**. Archetypes Based Techniques for Development of Domains, Requirements and Sofware. 2011.

68. **Juri Gavšin**. Intrinsic Robot Safety Through Reversibility of Actions. 2011.

69. **Dmitri Mihhailov**. Hardware Implementation of Recursive Sorting Algorithms Using Tree-like Structures and HFSM Models. 2012.

70. **Anton Tšertov**. System Modeling for Processor-Centric Test Automation. 2012.

71. **Sergei Kostin**. Self-Diagnosis in Digital Systems. 2012.

72. **Mihkel Tagel**. System-Level Design of Timing-Sensitive Network-on-Chip Based Dependable Systems. 2012.

73. **Juri Belikov**. Polynomial Methods for Nonlinear Control Systems. 2012.

74. **Kristina Vassiljeva**. Restricted Connectivity Neural Networks based Identification for Control. 2012.

75. **Tarmo Robal**. Towards Adaptive Web – Analysing and Recommending Web Users` Behaviour. 2012.

76. **Anton Karputkin**. Formal Verification and Error Correction on High-Level Decision Diagrams. 2012.

77. **Vadim Kimlaychuk**. Simulations in Multi-Agent Communication System. 2012.

78. **Taavi Viilukas**. Constraints Solving Based Hierarchical Test Generation for Synchronous Sequential Circuits. 2012.

79. **Marko Kääramees**. A Symbolic Approach to Model-based Online Testing. 2012.

80. **Enar Reilent**. Whiteboard Architecture for the Multi-agent Sensor Systems. 2012.

81. **Jaan Ojarand**. Wideband Excitation Signals for Fast Impedance Spectroscopy of Biological Objects. 2012.

82. **Igor Aleksejev**. FPGA-based Embedded Virtual Instrumentation. 2013.

83. **Juri Mihhailov**. Accurate Flexible Current Measurement Method and its Realization in Power and Battery Management Integrated Circuits for Portable Applications. 2013.

84. **Tõnis Saar**. The Piezo-Electric Impedance Spectroscopy: Solutions and Applications. 2013.

85. **Ermo Täks**. An Automated Legal Content Capture and Visualisation Method. 2013.

86. **Uljana Reinsalu**. Fault Simulation and Code Coverage Analysis of RTL Designs Using High-Level Decision Diagrams. 2013.

87. **Anton Tšepurov**. Hardware Modeling for Design Verification and Debug. 2013.

88. **Ivo Müürsepp**. Robust Detectors for Cognitive Radio. 2013.

89. **Jaas Ježov**. Pressure sensitive lateral line for underwater robot. 2013.

90. **Vadim Kaparin**. Transformation of Nonlinear State Equations into Observer Form. 2013.

92. **Reeno Reeder**. Development and Optimisation of Modelling Methods and Algorithms for Terahertz Range Radiation Sources Based on Quantum Well Heterostructures. 2014.

93. **Ants Koel**. GaAs and SiC Semiconductor Materials Based Power Structures: Static and Dynamic Behavior Analysis. 2014.

94. **Jaan Übi**. Methods for Coopetition and Retention Analysis: An Application to University Management. 2014.

95. **Innokenti Sobolev**. Hyperspectral Data Processing and Interpretation in Remote Sensing Based on Laser-Induced Fluorescence Method. 2014.

96. **Jana Toompuu**. Investigation of the Specific Deep Levels in *p*-, *i*- and *n*-Regions of GaAs p^+-*pin*-n^+ Structures. 2014.

97. **Taavi Salumäe**. Flow-Sensitive Robotic Fish: From Concept to Experiments. 2015.

98. **Yar Muhammad**. A Parametric Framework for Modelling of Bioelectrical Signals. 2015.

99. **Ago Mõlder**. Image Processing Solutions for Precise Road Profile Measurement Systems. 2015.

100. **Kairit Sirts**. Non-Parametric Bayesian Models for Computational Morphology. 2015.

101. **Alina Gavrijaševa**. Coin Validation by Electromagnetic, Acoustic and Visual Features. 2015.

102. **Emiliano Pastorelli**. Analysis and 3D Visualisation of Microstructured Materials on Custom-Built Virtual Reality Environment. 2015.

103. **Asko Ristolainen**. Phantom Organs and their Applications in Robotic Surgery and Radiology Training. 2015.

104. **Aleksei Tepljakov**. Fractional-order Modeling and Control of Dynamic Systems. 2015.

105. **Ahti Lohk**. A System of Test Patterns to Check and Validate the Semantic Hierarchies of Wordnet-type Dictionaries. 2015.