Thesis on Information and Communication Technology CX

Model-based error localization and mutation-based error correction algorithms and their implementation for C designs

Urmas Repinski

Tallinn 2016

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

This dissertation is accepted for the defense of the degree of the doctor of philosophy in the computer science on 01th of July, 2016

Supervisors: prof. Tanel Tammet, Department of Computer Science, Tallinn University of Technology,
dots. Juhan-Peep Ernits, Department of Computer Science, Tallinn University of Technology,
prof. Jaan Raik, Department of Computer Engineering, Tallinn University of Technology.

Opponents: prof. Erik G. Larsson, Department of Electrical Engineering, Linköping University,
prof. Varmo Vene, Institute of Mathematics and Statistics, University of Tartu.

Defense: 01th of June, 2016

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

/Urmas Repinski/

ISSN 1406-4731
ISBN 978-9949-23-250-5 (publication)
ISBN 978-9949-23-251-2 (PDF)

Info ja Kommunikatsioonitehnoloogia CX

Mudeli-põhine vigade lokaliseerimine ja mutatsiooni-põhine vigade korrektsiooni meetodid ja nende rakendamine C programmide jaoks

Urmas Repinski

Tallinn 2016

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia Teaduskond
Arvutiteaduse Instituut

Doktoritöö on lubatud kaitsmisele filosoofiadoktori kraadi taotlemiseks infotehnoloogia erialal 01. Juulil, 2016

Juhendaja: Tanel Tammet, professor, Arvutiteaduse Instituut, Tallinna Tehnikaülikool,
Juhan-Peep Ernits, professor, Arvutiteaduse Instituut, Tallinna tehnikaülikool,
Jaan Raik, professor, Arvutitehnika Instituut, Tallinna Tehnikaülikool.

Opponendid: Erik G. Larsson, professor, Elektrotehnika Instituut, Linköping University,
Varmo Vene, professor, Matemaatika ja Statistika Instituut, Tartu Ülikool.

Doktoritöö kaitsmise aeg: 01. Juuni, 2016

Deklaratsioon:

Deklareerin, et käesolev väitekiri, mis on minu iseseisva töö tulemus, on esitatud Tallinna Tehnikaülikooli filosoofiadoktori kraadi taotlemiseks ja selle alusel ei ole varem taotlenud akadeemilist kraadi.

Urmas Repinski ….....

ISSN 1406-4731
ISBN 978-9949-23-250-5 (publicatsioon)
ISBN 978-9949-23-251-2 (PDF)

Many thanks to Robert Könighofer, Georg Hofferek and their PhD supervisor Roderick Bloem from the Graz University of Technology, to Alexander Finder and his PhD supervisor Goerschwin Fey from the University of Bermen for their help with developing the open-source FORENCSIC tool during the DIAMOND FP7 project and with publishing the results obtained with the tool.

Abstract

Within contents of the dissertation the FORENSIC tool, that is a tool for automatic localization and correction of the errors in the C designs, that is the result of the DIAMOND FP7 project, is described. In the simulation-based back-end of the FORENSIC tool, the model-based error localization and mutation-based error correction algorithms and functionalities are implemented, that were developed by the author of this dissertation.

Dynamic slicing algorithm, that can be applied during the model-based error localization, that increases the error localization accuracy, were implemented within contents of the FORENSIC tool, with experimental data.

Using number of mutations, applied during the mutation-based error localization algorithm's execution, as a meter of the model-based error localization accuracy, in contrast with number of statements, that is required to inspect in order to reach the error location, is a novel approach, that is implemented within contents of the FORENSIC tool.

Novel ranking approaches for the model-based error localization, that are simple ranking and consecutive ranking, with comparison of the experimental data, achieved with the FORENSIC tool, and with data, published in related works, are described.

Specification format for the error localization and error correction tool is described within contents of the dissertation, together with other novel approaches, that were implemented in the FORENSIC tool, that are simulation of the model using C functionality.

~~Experimental results, obtained using the~~ Experience, obtained with the FORENSIC tool can be used in the process of the industrial software development for creation of the industrial tool, that will be responsible for

design verification, error localization and error correction. This can reduce the development time and, therefore, the development expenses also.

Kokkuvõte

Väitekirjas on kirjeldatud FORENSIC tööriist, mis on vahend vigade automaatseks lokaliseerimiseks ja parandamiseks C programmide jaoks, mis oli projekteeritud ja implementeeritud DIAMOND FP7 projekti jooksul. FORENSIC tööriista simuleerimisel-põhinev (ingl. k. simulation-based) back-end implementeerub mudeli-põhise vigade lokaliseerimise (ingl. model-based error localization) ja mutatsioonil-põhineva veakorrektsiooni (ingl. mutation-based error correction) algoritmid ja funktsioonaalsus, mis on välja töötatud väitekirja autori poolt.

Dünaamilise viilutamise algoritm (ingl. dynamic slicing algorithm). on implementeeritud FORENSIC tööriistas. Seda saab rakendada mudeli-põhise vealokaliseerimise algoritmi juures ning see suurendab vigade lokaliseerimise täpsust, mida näitavad ka katseandmed.

Mutatsioonide arvu, mis on saadud mutatsioonil-põhineva veakorrektsiooni algoritmi käigus, kasutatakse selleks, et mõõta mudeli-põhise vealokaliseerimise algoritmi täpsust, mis on uudne lähenemine vastukaaluks programmi põhilausete arvuga, mida on vajalik läbida, et jõuda vea asukoha. See lähenemisviis on rakendatud ka FORENSIC tööriistas.

Kirjeldatud on uudsed järjestamise lähenemisviisid (ingl. ranking approaches) mudeli-põhise vea-lokaliseerimise algoritmi jaoks (ingl. k. simple ranking ja consecutive ranking) koos vastava katseandmetega, mida on võrreldud seotud töödes publitseeritud andmetega.

Väitekirjas on kirjeldatud spetsifikatsiooni formaat vea lokaliseerimine ja vea parandamise tööriista jaoks, koos teise uudse meetodiga, mis on rakendatud FORENSIC tööriistas, milleks on mudeli simulatsioon kasutades C funktsionaalsust.

~~Katsetulemused, mis on saadud~~ Kogemus, mis on saadetud FORENSIC tööristaga on võimalik kasutada tööstuslikuks tarkvaraarenduse protsessis, et

automaatselt lokaliseerida ja parandada vigu tarkvaras. See võib vähendada arendusaega ja seega arenduskulusid.

Table of contents

List of Publications

Raik, Jaan; Repinski, Urmas; Hantson, Hanno; Jenihhin, Maksim; Di Guglielmo, Giuseppe; Pravadelli, Graziano; Fummi, Franco (2012). Combining Dynamic Slicing and Mutation Operators for ESL Correction. 17th IEEE European Test Symposium (1−6). IEEE.

Repinski, Urmas; Raik, Jaan (2012). Comparison of Model-Based Error Localization Algorithms for C Designs. Proc. of 10th East-West Design & Test Symposium, Kharkov, Ukraine, September 14-17, 2012 (1−4). IEEE Computer Society Press.

Hantson, H.; Repinski, U.; Raik, J.; Jenihhin, M.; Ubar, R. (2012). Diagnosis and Correction of Multiple Design Errors Using Critical Path Tracing and Mutation Analysis. In: 13th IEEE Latin-American Test Workshop Proceedings (27−32). IEEE Computer Society Press.

Raik, Jaan; Repinski, Urmas; Jenihhin, Maksim; Chepurov, Anton (2011). High-Level Decision Diagram Simulation for Diagnosis and Soft-Error Analysis. Design and Test Technology for Dependable Systems-on-Chip (294−309). IGI Publishing.

Raik, Jaan; Repinski, Urmas; Ubar, Raimund; Jenihhin, Maksim; Chepurov, Anton (2010). High-level design error diagnosis using backtrace on decision diagrams. 28th Norchip Conference 15-16

November 2010, Tampere, Finland, IEEE.

Other Related Publications

Урмас Репинский (2012), Верификация на основе симуляции с нахождением и исправлением ошибок для с-дизайнов, Програмные продукты и системы, № 100, 2012 год, стр. 229-237.

Urmas Repinski (2012). Model-based Verification with Error Localization and Error Correction for C Designs, Програмные продукты и системы (international journal Software and Systems), № 100, 2012, pp. 221-229.

Bloem, Roderick; Drechsler, Rolf; Fey, Goerschwin; Finder, Alexander; Hofferek, Georg; Koenighofer, Robert; Raik, Jaan; Repinski, Urmas; Suelflow, Andre (2012). FoREnSiC - An Automatic Debugging Environment for C Programs, Haifa Verification Conference (HVC'2012), pp 1-6, IBM Research Labs, Haifa, Israel: IBM.

Author's Contribution to the Publications

27. Bloem, Roderick; Drechsler, Rolf; Fey, Goerschwin; Finder, Alexander; Hofferek, Georg; Koenighofer, Robert; Raik, Jaan; Repinski, Urmas; Suelflow, Andre (2012). FoREnSiC - An Automatic Debugging Environment for C Programs, Haifa Verification Conference (HVC'2012), pp 1-6, IBM Research Labs, Haifa, Israel: IBM.

This group publication describes structure of the developed within contents of the DIAMOND FP7 project FORENSIC tool in details. Article includes experimental data, achieved using all back-ends of the FORENSIC tool, with description of the back-ends and front-end of the tool. As the article were made before the end of the project (project ended at 01.01.2013), the experimental data, and algorithms descriptions, provided in the article, are not final, that will be corrected in the next articles.

43. Repinski Urmas. 2012. Model-based Verification with Error Localization and Error Correction for C Designs. Программные продукты и системы (international journal Software and Systems), № 100, 2012, 221-229.

I am the only author of the publication, where I described principles of implemented in the FORENSIC [16] tool error localization and error correction algorithms and dynamic slicing in details, with corresponding experimental data provided. Does include mathematical definitions of the model and of the algorithms described.

44. Repinski, U., Hantson, H., Jenihhin, M., Raik, J., Ubar, R., Di Guglielmo, G., Pravadelli, G., Fummi, F. 2012. Combining dynamic slicing and mutation operators for ESL correction. Test Symposium (ETS), 2012 17th

IEEE European, 1-6, 2012.

This is the group publication. This article in describe algorithms of model-based error localization and dynamic slicing, that are implemented in the FORENSIC tool, but does not include strict mathematical definitions of the model and of the algorithms, includes sections, that were developed by the co-authors also.

45. Repinski Urmas, Raik Jaan. 2012. Comparison of Model-Based Error Localization Algorithms for C Designs. Proc. of 10th East-West Design & Test Symposium, Kharkov, Ukraine, September 14-17, IEEE Computer Society Press, 1-4.

I am the main author of the publication, where principles of implemented in the FORENSIC [16] tool error localization and error correction and dynamic algorithms in details, with corresponding experimental data provided. The publication includes description of the novel approaches, implemented in the FORENSIC [16] tool, like simulation using C functionality of the model and usage of novel simple ranking for model-based error localization.

Hantson, H.; Repinski, U.; Raik, J.; Jenihhin, M.; Ubar, R. (2012). Diagnosis and Correction of Multiple Design Errors Using Critical Path Tracing and Mutation Analysis. In: 13th IEEE Latin-American Test Workshop Proceedings (27−32). IEEE Computer Society Press.

This publication is about implementation of Mutation-Based Error Correction algorithm for High-Level Decision Diagrams, that is a structure, that allows to store parsed representation of “specific” C deigns, that are not included into Siemens Benchmarks [6] set. High-Level Decision Diagram's Mutation-Based Error Correction Algorithm were extended and improved, and further applied in FORENSIC [16] tool to Siemens Benchmarks [6].

Raik, Jaan; Repinski, Urmas; Jenihhin, Maksim; Chepurov, Anton (2011). High-Level Decision Diagram Simulation for Diagnosis and Soft-Error Analysis. Design and Test Technology for Dependable Systems-on-Chip (294−309). IGI Publishing.

This publication is about implementation of Error Localization algorithm for High-Level Decision Diagrams, that is a structure, that allows to store parsed representation of “specific” C deigns, that are not included into Siemens Benchmarks [6] set.

Glossary

Animation

A process of execution or simulation of the specification, with a goal to obtain reference outputs from specification inputs. Animation is impossible to implement if the specification is non-explicit.

Black box

A device, system, object or design that can be viewed only in terms of its input, output and transfer characteristics without any knowledge of its internal structure: its implementation is "opaque" (black).

Candidates for correction

A data structure that stores the results of the error localization and is used as an input for error correction. Contains references to the design model components with possible additional information.

Debugging

The process of error localization and/or correction.

Design

Debugged program code. Using the FORENSIC tool [16] design is parsed, verified and processed using error localization and error correction algorithms during debugging.

Design Verification

Design verification is a process of ensuring that the design

corresponds to its specification: one of the components of the software development cycle.

DIAMOND

European Union's 7th Framework Programme's collaborative research project FP7-2009-IST-4-248613 DIAMOND - Diagnosis, Error Modeling and Correction for Reliable Systems Design - aiming at improving productivity and the reliability of software designs by providing a systematic methodology and an integrated environment for the error localization and correction [14].

Dynamic slicing for the model-based error localization

An algorithm that is applied to the model-based error localization's simulation results and allows to reduce the number of candidates for correction. The idea behind the dynamic slicing is following: some amount of nodes of the model are activated during simulation of the error localization, but do not have any influence to the simulation output. Dynamic slicing allows to discard those nodes from the simulation trace.

Error correction

An algorithm that allows to correct errors in the design in case the design verification fails. mutation-based error correction algorithm applies different possible mutations to the design model to correct errors.

Error localization

An algorithm that allows to locate errors in the design in case the design verification fails. model-based error localization algorithm simulates the design model and uses the simulation trace to extract the most probable candidates for correction from the nodes of a model.

Flow Chart Node

An element of the FORENSIC model that represents the parsed design for its future processing. Flow Chart Node is further divided into an Operation Node (operational statements like assignments and functions) and Condition Node (conditional operations in the design).

FORENSIC

The FORENSIC (FOrmal Repair ENgine for SImple C) model/tool was implemented during the DIAMOND FP7 project. It is an implementation of the simulation-based verification with the model-based error localization and

the mutation-based error correction algorithms in the *C* designs. Its front-end parses a design into the model and the various back-ends are implementations of the algorithms [16].

Formal specification

During the process of the software development formal specification provides a description of the created design and serves as a document for the verification engineers, describing the verification approaches of the created design. In our case it is also used as a component for simulation-based verification, generating reference outputs.

Formal verification

Formal verification includes two different verification approaches: determining whenever two designs are equivalent (equivalence checking) and whether a design has a specific property (property verification).

IDE – Integrated Development Environment

A software application that provides comprehensive facilities to computer programmers for software development.

The model

"White-box" representation of the processed design that can be simulated in order to obtain the design simulation trace and to apply any changes within its structure for the correction. Those properties are required for model-based error localization and mutation-based error correction.

Model-based error localization algorithm

A process of selecting the most probable candidates for correction that contain error(s) and extracting the most probable erroneous components of the processed design model from the total amount of the components of the model.

Mutation-based error correction algorithm

A process of correcting errors in the design model by applying mutations and using the candidates for correction structure, provided by the model-based error localization. Mutated design is verified: in case the verification passes, the correctional mutation is found, otherwise further mutations are applied in the future while all the possible mutations are checked.

Object Oriented – OO

A programming paradigm using "objects": data structures that consist of data fields and methods together with their interactions.

Observation Points

Locations in the design and in the specification where the designed behavior is compared with the correct behavior defined by the specification. Can be of the Input and Output types.

One/multiple error assumption

An assumption that there is only one possible error in the processed design or that there are several errors. The mutation-based error correction algorithm implementations differ for different error assumptions.

Ranking for the model-based error localization

An algorithm that is applied to the design model simulation results during model-based error localization with the goal to select the most probable error candidates from the total amount of the candidates captured during the simulation.

Requirements

A Software requirements specification (SRS), is a description of the behavior of a design to be developed, it contains use cases and non-functional requirements.

Simulation

A way to execute the design or the model to get the outputs from the inputs.

Simulation-based verification

A process of simulating the design or the model together with the animation of the specification and the comparison of outputs. In case the outputs match for all inputs, the simulation-based verification of design is passed, otherwise the verification fails.

Specification

Correct behavior of the processed potentially erroneous design. Specification outputs are used for verification.

White box

In contrast with the “black box”: a device, system, object or design whose internals can be viewed and modified. In our case the design model used by the error localization and correction algorithms.

Chapter 1. Introduction

Already in 1980-s the process of software development changed from small tasks involving a few people to large tasks involving several groups – designers, programmers, verification engineers. During this process verification itself has also undergone a change - from an informal process performed by the software engineer to a separate activity in the overall software development life cycle. Contemporary verification is highly formalized [51].

A specification is used to define the correct behavior of the design during verification. It can be also used for the future maintenance of the design. The specification format is important, and a lack of the specification will inflict high indirect losses: because of lacking or incomplete specifications it is sometimes impossible to rebuild fully functional software with the new requirements for the software, and as a result new specification, and together with that, new software should be developed from scratch - including the corresponding research and project development phases, which is expensive and can be avoided if the design's specifications were in a suitable format.

Using informal specifications instead of formal ones introduces “human factor” into the process of verification and negates any profit from the use of specifications for the verification.

During the period 01.01.2010 – 01.01.2013 the DIAMOND FP7 [14] project was conducted by the Tallinn University of Technology and partners: University of Bremen, Graz University of Technology, Linköping University, IBM and Ericsson research groups. The goal of the project was to develop new algorithms and methods for error localization and error correction in the C designs, to implement them in a usable tool and to publish the experimental results, that can be used further for developing industrial error localization

and error correction tools.

The front-end of the FORENSIC[16] tool, that is the result of the project, was developed at the University of Bremen by Alexander Finder and colleagues from Computer Architecture group. The model, that stores parsed C design for processing, was developed at the Graz University of Technology by Robert Könighofer and colleagues from the Institute for Applied Information Processing and Communications – IAIK. Various back-ends were implemented at the Linköping University, Graz University of Technology (SMT-solver based back-end) [27], and at the Tallinn University of Technology, Simulation-based back-end, by the author of the dissertation.

1.1 Problem formulation

Ready-to-use software design is a “black box”, when during its work inputs are required and provided and outputs are generated, but the entire design structure is irrelevant. If we want to have a possibility to debug the design, to verify correspondence to the specification and to localize and correct errors in the design if design's verification fails, then it should be presented as a "white box", or, in other words, it should be parsed. Parsed representation of the design is stored in a corresponding data structure, let's name it a model; it allows to obtain a simulation trace of the design and to apply any changes to its structure. Those model properties are used for error localization and correcting implementations – localization algorithms use the trace of design's simulation in order to select the most probable erroneous components of design, correction requires full access to any part of the processed design. Possibility to set Observation Points into design allows to verify any part of the design that is critical for verification, error localization and further error correction.

Model, on the other hand, can be seen as an Integrated Development Environment (IDE) for error localization and correction, as a place where those algorithms can be implemented, and a functionality that can be implemented, strongly depends on the model structure and it's properties. FORENSIC (FOrmal Repair ENgine for SImple C) tool [16], that were implemented within the contents of the DIAMOND FP7 project [14], is designed for error localization and error correction. It satisfies the requirements stated above and includes a functionality to store corresponding parsed representation of design.

Algorithms, that were implemented in the tool and will be described within contents of the dissertation, are model-based error localization, dynamic slicing and mutation-based error correction. The algorithms use

simulation-based verification and allow to correct simple, frequently found errors in the *C* designs like misuse of arithmetical operator in the design or error in one digit of the number. Applying an additional dynamic slicing algorithm for model-based error localization will make error localization more accurate and, therefore, error correction less time consuming.

Other critical questions are how the correct behavior of the processed design is presented for verification, the specification language and the format of the specification. Popular verification languages are Vera, e, C/C++, and Java [33]. *C* is used as a specification language in the FORENSIC tool[16] – it is the same as the design language, thus allowing to use the same functionality for the design simulation and specification animation.

1.2 Requirements for the tool

Requirements for the tool that can locate and correct errors in the design are following:

- The tool should be well organized and structured, have a clear documentation and tutorial to be understandable and suitable for the application.
- The tool should be able to process any kind of processed design. If the tool is designed for simulation-based verification with error localization and correction of the C-type designs, then any C-type design should be processable with the tool.
- The tool should have outputs and inputs in clear format. It should be possible to define, modify and extend the tool by the researcher, who is actually using the tool.

1.3 Contributions and novel approaches

Various functionalities were developed and implemented, and novel approaches, that were used and applied within contents of the DIAMOND FP7 project [14], are described in this dissertation and corresponding publications and are stated below:

- Usage of novel rankings for model-based error localization – Consecutive and Simple rankings along with the re-implementation of already known Ample, Jaccard, Ochiai and Tarantula rankings using a clearer and simpler mathematical representation form, that is different with one, used in Spectrum-Based error localization [1], refer to Chapter 4.
- Usage of the compiler functionality for the model's simulation,

specification animation and implementation of the dynamic slicing algorithm, instead of direct execution of the model's nodes, refer to Chapter 3.

- Usage of the compiler's functionality for model's simulation is applicable for an arbitrary imperative programming language. This approach allows to implement dynamic slicing with model-based error localization for more precise error localization, that is shown using experimental data, refer to Section 3.2 and to Section 6.1.
- Usage of the compiler's functionality for model's simulation allows to obtain any necessary data during design's execution with not losing any original design's functionality, and getting benefits like speed of the simulation and simplicity of the approach, comparing with the direct execution of the model. Additionally, we do not get any programming errors when we want to re-implement programming language's functionality. Refer to Section 3.2.

◆ Usage of the number of applied mutations instead of a number of statements required to reach the error as a measure of the error localization accuracy.

- This is more accurate meter of the error localization accuracy than number of statements. The metric is more accurate in the sense that every statement can contain many different operators in original design, and, therefore, many different nodes in the corresponding model, and many different possible correctional mutations, and number of mutations, applied for error correction, should be more “realistic” meter.
- Possible mutations are defined in supported error classes for error correction and described in Section 5.2. Implemented mutations for the mutation-based error correction algorithm process logic like operators, numbers, constants.

◆ Implementation of the model-based error localization, mutation-based error correction and dynamic slicing algorithms in the corresponding tool.

- Model-based error localization algorithm uses the simulation trace of the model with various inputs. It assigns passed or failed counters to nodes that belong to every simulation trace. Finally, various rankings are applied to the nodes using counters. Nodes that have highest ranks are candidates for correction and are stored in the corresponding data structure.
- Dynamic Slicing algorithm can be applied to the activated nodes of every simulation. It allows to discard those nodes from the trace that do not have any influence on the simulation output.
- Mutation-based error correction algorithm uses information provided

by the candidates for correction and applies various correction mutations to the nodes. It starts with the nodes with the highest rank and re-verifies the design according to the specification. If successful, then the correctional mutation had been found, an error is localized and corrected. If not, the process is continued until all the possible mutation are applied or the correction is found.

- Error localization and error correction algorithms implemented enable performing practical experiments demonstrating the developed functionality in practice.

◆ Definition of the specification format for simulation-based verification.

- Definition of the specification format and specification language for the verification tool has been performed within the contents of the dissertation together with the Observation Point's definition necessary for the verification.

◆ Finally, the FORENSIC tool can be used in the process of software development. The experimental results obtained with the tool make it possible to implement similar tools with a more user-friendly interface, yet based on the same verification principles. This is the most important contribution achieved in the DIAMOND project.

1.4 Organization of the Dissertation

The dissertation is organized as follows:

◦ **Chapter 1 Introduction** includes introduction, problem formulations, requirements for the error localization and error correction tool, contributions and novel approaches.

◦ **Chapter 2 Background** describes background of fault theory and design verification basics, that include hardware and software design verification and development and verification complexity. Simulation-based verification is compared with formal verification, basics of the model-based error localization and mutation-based error correction is introduced, in Chapter 4 and Chapter 5 those algorithms will be described in details. Specification format for the error localization and error correction tool will be chosen. Will be explained the decision why C specification language were chosen to be a specification language for the FORENSIC [16] tool.

- **Chapter 3 Data structures for the error localization and error correction implementation** defines the data structures, used for error localization and error correction. Additionally, principles used in order to implement simulation-based verification, model-based error localization and mutation-based error correction are described. We define of the model where the processed design is stored, principles of the simulation of the model and the animation of the specification along with the definition of Observation Points. In the last sections we describe the structure of the FORENSIC tool [16], when simulation-based verification, model-based error localization and mutation-based error correction is implemented.

- **Chapter 4 Model-based error localization** describes the algorithm of the model-based error localization in details, including the mathematical definition of the model-based error localization algorithm. Model definitions, made in the Chapter 3, will be used for error localization definition. Model-based error localization algorithm uses the simulation trace of the model with various inputs. It assigns passed or failed counters to nodes that belong to every simulation trace. Finally, various rankings are applied to the nodes using counters. Nodes that have highest ranks are candidates for correction and are stored in the corresponding data structure.

- **Chapter 5 Mutation-based error correction** describes the algorithm of the mutation-based error correction, including the corresponding mathematical definitions, like definition of the mutation-based error correction algorithm and definitions of error classes, that are supported by the algorithm. Mutation-based error correction algorithm uses information, provided by the candidates for correction, and applies various correction mutations to the nodes. It starts with the nodes with the highest rank and re-verifies the design according to the specification. If successful, then the correctional mutation had been found, an error is localized and corrected. If not, the process is continued until all the possible mutations are applied or the correction is not found.

- **Chapter 6 Experimental Results** contains experimental results, performed using Siemens designs as inputs for the FORENSIC tool [16]. Those are open-source *C* programs with a number of different erroneous versions and inputs for design verification and simulation. Three types of experiments are described in the current Chapter: model-based error

localization, dynamic slicing and mutation-based error correction experiments, all using the Siemens designs [6], experiments are compared with similar ones, published in [12, 60, 1]. Mutation-based error correction experiments are compared with the corresponding experiments made by Debroy and Wong [12], and model-based error localization experiments are compared with the corresponding results by Abreu, Zoeteweij and van Gemund [1].

- **Chapter 7 Conclusions and future extensions** contains conclusions and possible future extensions of FORENSIC tool [16].

- **Appendix 1 Dynamic slicing experiments using FORENSIC** presents the dynamic slicing experiments: model-based error localization using simple ranking with and without dynamic slicing, using Siemens designs as inputs for the FORENSIC tool [16].

- **Appendix 2 Ranking comparison for the model-based error localization of FORENSIC** describes a comparison of experimental results for the model-based error localization - model-based error localization using various rankings with the dynamic slicing with mutation-based error correction algorithms. Again, Siemens designs is used as inputs for the FORENSIC tool [16].

- **Appendix 3 Curriculum Vitae** is English language version of author's Curriculum Vitae.

- **Appendix 4 Elulookirjeldus** is Estonian language version of author's Curriculum Vitae.

1.5 Chapter summary

Current chapter is introductory chapter and includes introduction to the dissertation, problem formulation, requirements for the error localization and error correction tool, contributions and novel approaches.

Chapter 2. Background

Two groups of people are essential for a successful design project: the design team and the verification team. Designers usually get their training from schools and universities. Verification engineers, in contrast, learn their craft at work, as their tasks require a practical understanding of the structure and principles of the verified software design [33]. Verification engineers need tools to make their work easier and less human-dependent. This Chapter contains descriptions, definitions and principles that are used by the verification engineers in their practice, together with the principles of the tool, that can be used for automatic error localization and error correction.

The Chapter is organized as follows. First of all, software development principles for hardware design are introduced, then it is shown how important is the verification process for software and hardware development, hardware development phases and basic verification principles are described. Then we discuss the principles of the simulation-based and formal verification along with the error localization and correction principles. The implementation details of the latter will be described in detail in the next Chapters, in Chapter 4 and Chapter 5. The decision, that is more suitable to use – formal verification or simulation-based verification - is also made in this Chapter. Most suitable type of the specification format for verification is described in this Chapter also, and decision is made what type of specification is most suitable for our verification tool.

As defined in [2] the following terminology is being used in the Chapter:

• A failure is an event that occurs when a delivered service deviates from the correct service.

• An error is a system state that may cause a failure.

• A fault is cause of an error in the system.

2.1 Development phases

The following phases can be recognized during the lifetime of a hardware product: component manufacturing, board manufacturing, system manufacturing and actual use of the product. Product quality has to be ensured during every phase [23]. Relationship of the verification and correction costs during each of these phases can be approximated with a rule-of-ten [11]: if the cost of verification and correction in the component manufacturing phase is *R*, then in the board manufacture phase it is *10R*, in the system manufacturing phase it is *100R*, and during actual use phase it is *1000R*. The reason for this is an increasing difficulty of locating an erroneous component (refer to *Figure 2.1*).

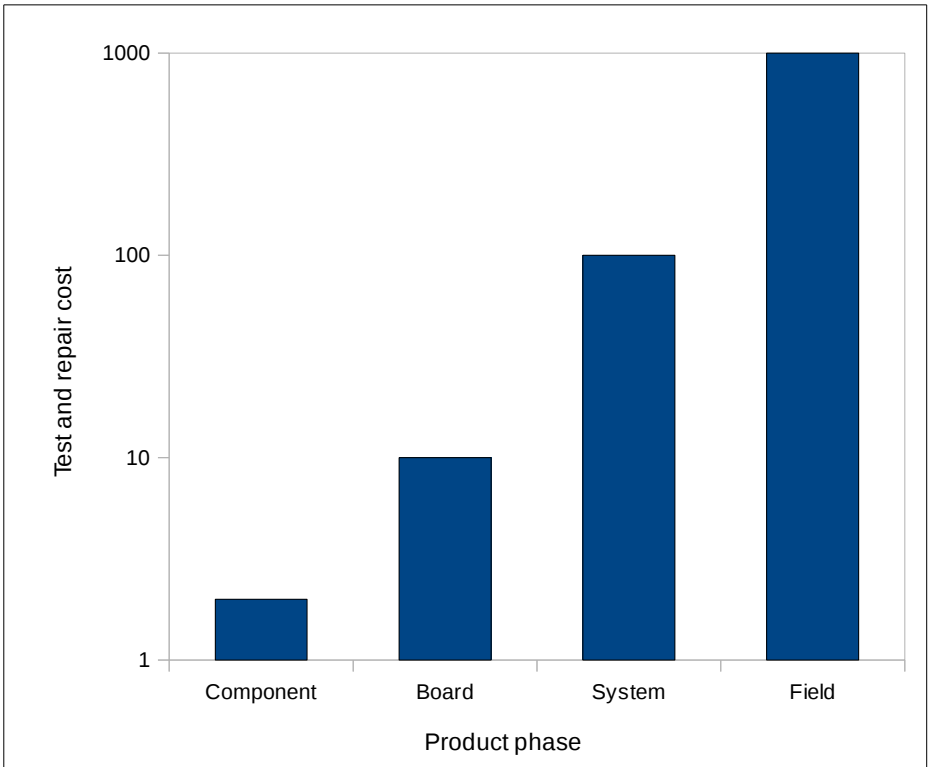

Figure 2.1: Rule-of-ten [11].

Hence the verification phase during hardware development is critical,

especially at the first phase of the component manufacturing. However, for the software development the situation is not so critical and the rule-of-ten is not applicable, as it is possible - in contrast to hardware development - to perform corrective updates at maintenance stage with no significant additional costs.

The process of software development can be separated into different phases and, for example, can be described using the waterfall model: refer to *Figure 2.2* [37]. The waterfall software development model is a sequential design process often used when the progress is seen as flowing steadily downwards (like a waterfall) through phases of Conception, Initiation, Analysis, Design, Construction, Testing, Production/Implementation and Maintenance.

Terms *validation* and *verification* differ, according to [57], that is cited in [40], *validation* means establishing and documenting evidence which provides a high degree of assurance that a process will consistently produce a result or product meeting its predetermined specifications, *verification* means confirmation by examination and provision of objective evidence that specified requirements related to a product or process have been met. Validation is used in earlier development phases, when developed product does not actually exist yet, verification is used when some version of the product is already exists. According to [40] unit testing is thorough testing of each unit of software, either by the developer or another independent team member, product verification testing verifies correctness with respect to the design; function input/output, path coverage and error guessing.

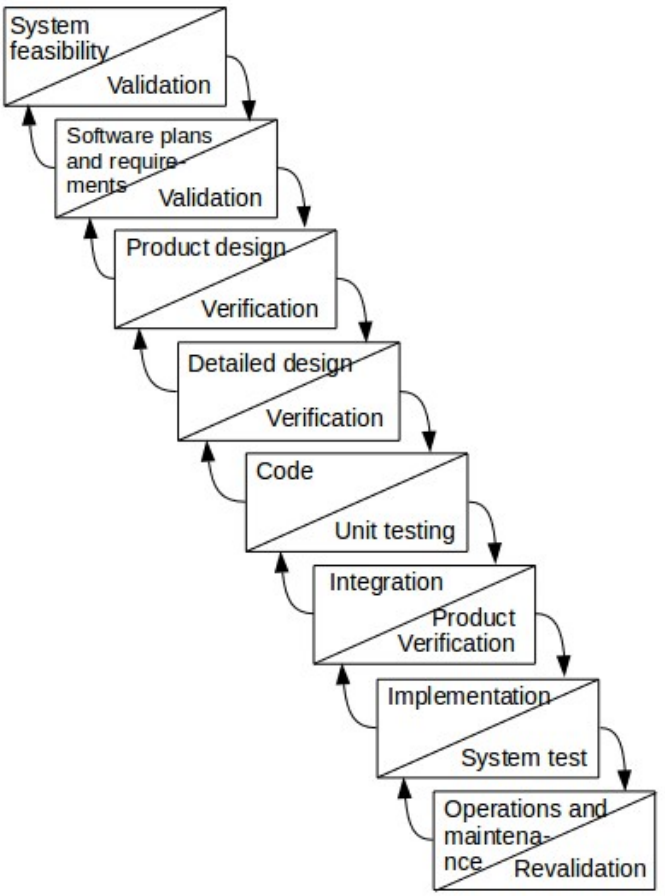

Figure 2.2 Waterfall Software Development model [37].

Each phase has its own abstraction level, own requirements and own errors that can appear during each phase of the development process. Higher-level abstraction phases are located above lower-level abstraction phases at the *Figure 2.2.*

Maintenance is usually already planned into the process of design. Most errors are viewed as “planned” and can be corrected with lower expenses during maintenance. Examples of maintenance is a process of monthly

updates of operating systems or regular updates of our favorite web browser.

2.2 Design verification

Design is a process of transforming the specification into an implementation of the specification, using a particular programming language. Both the specification and implementation are a form of the functionality description, at different abstraction levels [33]. The process of turning an abstract description into a concrete description is called refinement. For example, the design process refines a specification and produces concrete implementations. The different development stages of the design process are shown at *Figure 2.2* Waterfall Software Development model using down-directed arrows: from the higher abstraction levels to the lower abstraction levels.

Design verification is a reverse of the design process: it checks whether the implementation at the lower abstraction level corresponds to the specification at the higher abstraction level. Verification at different development stages is shown at *Figure 2.2* using up-directed arrows: from lower abstraction levels to the higher abstraction levels.

2.3 Verification complexity

Verification is a procedure of checking whether a design corresponds to its specification. Specifications are created at earlier development phases. The design step turning a specification into design requires the verification step ensuring that the design performs the functionality determined in the specification [33].

Verification complexity increases together with the design complexity. There is a non-linear relationship between design complexity and verification complexity, as shown in *Figure 2.3* [15].

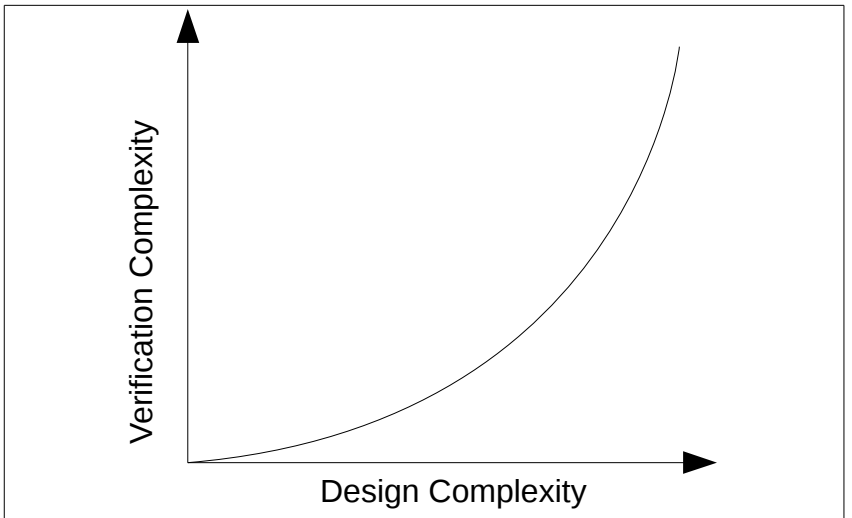

Figure 2.3 Verivication complexity according to design complexity.

Therefore verification of the more complicated design is more expensive, and, as the software complexity is getting higher nowdays, the verification is starting to play a major role in software development, as more and more expensive process.

2.4 Verification approaches

In order to verify the design we can choose between two general approaches of verification, those are simulation-based verification and formal verification. Semi-formal verification is a combination of simulation-based verification and formal verification principles.

◆ The simulation-based verification paradigm consists of four components: design, inputs, specification and a comparison mechanism. Design is simulated using inputs and the simulation output is compared with the reference outputs, obtained by the specification animation. Comparison mechanism samples simulation results and determines their equality with the reference output [33]. The basic principle behind simulation-based verification is illustrated in *Figure 2.4*.

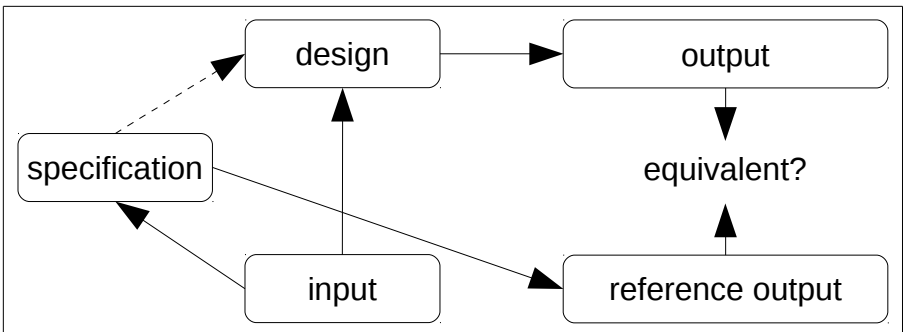

Figure 2.4: The basic principle of simulation-based verification of a design.

As it is seen from *Figure 2.4*, in order to use simulation-based verification of the design:

- reference outputs should be provided for the design with high design coverage to completely verify the design,
- the specification should be instrumented by the Observation Points insertion (refer to Chapter 3) onto design,
- the specification should be animated [61], for that it should be *explicit*, in order to get reference outputs.

Simulation-Based verification is practical approach, that is widely used during different phases of software development.

◆ Formal verification approach to error correction is divided into two categories: equivalence checking and property verification [31, 50]. Equivalence checking tests whether two designs are functionally equivalent. Property verification tests whether the verified design has a particular property. Formal verification is a vital aspect of safety-critical system design, not only to ensure proper functionality but also to provide formal proof of that functionality to regulators and oversight committees [32]. The basic principles of formal verification are illustrated at *Figure 2.5*.

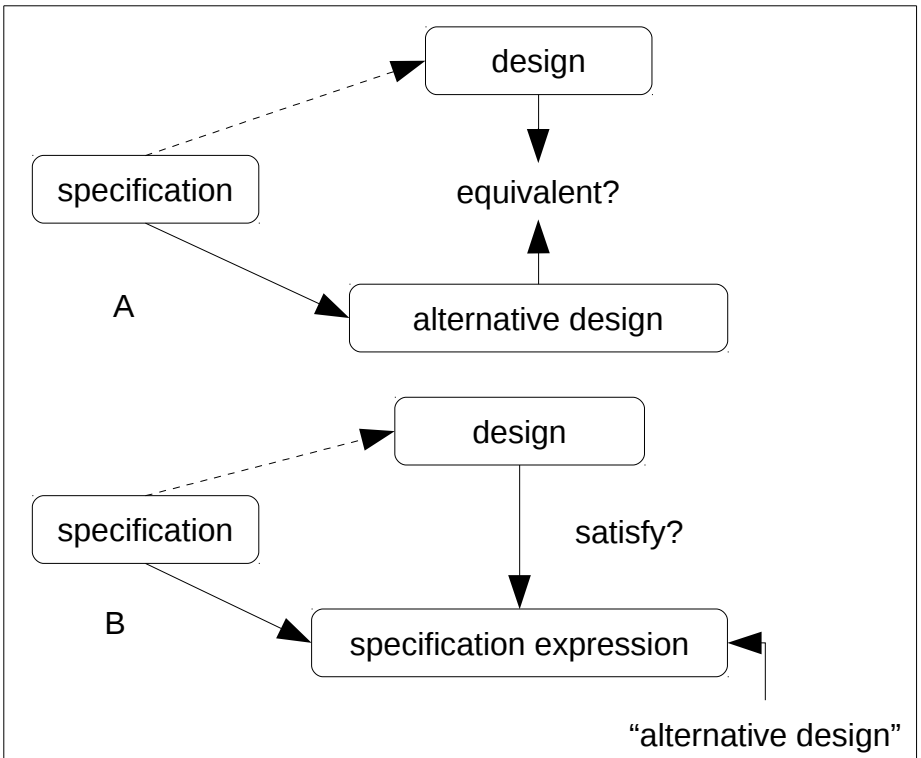

Figure 2.5: The basic principles of formal verification. A – equivalence checking, B – property checking.

Both simulation-based verification and formal verification approaches have their own advantages and disadvantages.

Disadvantages of the approaches:

- Simulation-based verification requires inputs with high design coverage, otherwise the verification is ineffective.
- Formal verification detects if two designs are NOT equivalent only if the design does NOT have a specific property [33]. Equivalence checking cannot state that designs are completely identical, since only the components are defined by the operator are checked.

Advantages of the approaches:

- Formal verification does not require pre-specified input data.
- Simulation-based verification approach is simpler and therefore easier to implement, more suitable for practical implementation and widely used.

Simulation-based verification is used in various tools [33], investigated

and describe in the number of works [47] – in [12] and [1] and is used in the FORENSIC tool [16].

2.5 Model-based error localization algorithm

The problem formulation for the error localization is the following:

Assume that there exists a design with an error, a model of the design – "white box" representation and a specification of the design (a description without the error). Assume that verification fails for the processed design. The task of error localization is to extract the most probable components from the model of the processed design that contain the error. model-based error localization algorithm requires a simulation trace of the processed design. Therefore the "white box" representation of the design model should be used for error localization.

There has been a lot of work in the area of model-based error localization over the past 20 years, and error localization algorithms have arisen from the artificial intelligence approaches [28, 42]. Currently there exists number of a similar research, named as a "spectrum-based error localization" [1], comparison of this algorithms is in Chapter 4.

Briefly, the basic algorithm of model-based error localization is as follows: (a) the parsed model of the processed design is verified using simulation-based verification, (b) simulation trace is stored, (c) depending on whether verification passes or fails, the passed/failed counters of the components activated during verification are incremented. After this the ranking algorithms are applied to the activated components using counters: components with highest ranks are selected as candidates for correction. A detailed description of the model-based error localization algorithm is presented in the Chapter 4 of the dissertation and in [45].

2.6 Mutation-based error correction algorithm

The problem formulation for error correction is following:

Assume that verification fails for the processed design and error localization has generated candidates for correction: the most probable erroneous components of the model of the design. The task of error correction is to correct errors found in the design to make the design correspond to its specification. mutation-based error correction algorithm applies a wide range of mutations: mutations in operators, numbers and constants in the design model. Then the algorithm checks whether the mutated design corresponds to

the specification. The algorithm uses simulation-based verification, is simple, flexible and reliable.

The mutation-based error correction's mutations and supported error classes implemented in the FORENSIC tool [16] are described in the Chapter 5 of the dissertation and in [45].

2.7 Specification for the error localization and correction tool

As shown in *Figure 2.4*, in order to implement the simulation-based verification algorithm, it is necessary to have a specification in a valid format. It should be possible to animate the specification, i.e. to produce reference outputs from inputs. The outputs generated will then be compared with the outputs of the design.

In general, a specification is a document that clearly and accurately describes essential technical requirements for items, materials, or services. A specification in the software development context provides a description of the system to be created and a guidance to the verification engineers for design verification.

Specifications can be formal or informal. In the context of business information systems, informal specifications and graphical modeling have been used at least since late 70s. In addition, a number of formal specification languages have been developed, such as Larch [20], VDM [19], Z [56], FOOPS [4] and OBJ [18]. “The formal specification techniques attempt to mathematically specify the structure, function, and behavior of information systems” [21]. The programming languages like C/C++/SystemC and Java are also used as specification languages [33].

When informal specification is used, then there exists a so-called “human factor”: a verification engineer should write test cases for verification by himself, recreating the formal data to guarantee that the design will correspond to the specification. It is more accurate (and sometimes cheaper) to use formal specifications. In this case the verification engineer should re-arrange the specification, should create test cases without insertion of the additional data or criteria, that minimizes “human factor” influence on the created test cases. In case of the formal specification the reference outputs can be produced by specification automatically, by specification animation. Elimination of the “human factor” from the process of the software verification is highly important, as this can guarantee the correctness of the test cases.

Formal specifications can be divided into two categories: explicit and non-explicit specifications. Specifications, which can be animated are called

explicit and are defined in the way that the reference outputs of the specification can be directly derived from the inputs [24]. It is is impossible to animate non-explicit specifications.

A language used for the specification format is called a verification language [33].

A format of specification is important for the implementation of simulation-based verification. It defines how the specification can be animated and a functionality that can be described using the specification. Requirements for the specification for the error localization and error correction tool using the C language are the following:

1. The specification format should be standardized, simple and easy to create.
2. It should be possible to animate the specification and to get reference outputs from inputs: in other words, the specification should be explicit.
3. It should be possible to specify any level of required functionality, of the C design.

Different specification approaches are described in the current Section.

2.7.1 Specification using PSL

Property Specification Language PSL [9] is a verification language developed by Accellera for specifying properties with the help of assertions in the hardware designs. Properties can be simulated or formally verified. Since September 2004 standardization of the language has been conducted by the IEEE 1850 working group. In September 2005 the IEEE 1850 Standard for Property Specification Language was announced.

Property Specification Language is targeted for several electronic system design languages such as:

- VHDL (IEEE 1076),
- Verilog (IEEE 1364),
- System Verilog (IEEE 1800),
- SystemC (IEEE 1666) by OSCI.

Properties in PSL are composed of Boolean expressions written in the host language together with temporal operators and sequences native to PSL [9].

Our designs for the FORENSIC tool [16] are written in the *C* programming language. *C* does not contain temporal operators like *VHDL*, *Verilog* or *SystemC*, and rest of the PSL functionality is not enough to completely describe any possible *C* design.

As a result a conclusion is made that there is no need to use any

functionality of the PSL specifications for the verification of C designs [9].

2.7.2 Specification using UML

Unified Modeling Language UML [49] is a general-purpose standardized modeling language. The standard was created and is managed by the Object Management Group. It was first added to the list of OMG adopted technologies in 1997 and has since become an industry standard for modeling software-intensive systems. UML includes graphic notation techniques allowing to create visual models of object-oriented software-intense systems.

UML allows to describe static and dynamic properties of the developed system and to describe class and functional structures of the design [5]. This type of specifications is easy to construct and can describe C designs with a good accuracy. Therefore the requirements 1 and 3 for C design verification specifications are satisfied.

However, UML specifications cannot be animated: they are non-explicit, as they define a structure of the design and possibly some fragments of functional behavior. The requirement 2 is not satisfied and hence UML specifications are not suitable for verifying C designs, although the UML diagrams can still be useful for the software designer.

2.7.3 Specification using Z notation

Z notation language [56] is standardized, well known and has a number of tools available for formal proofs, equivalence checking and type-checking.

In case a Z specification is explicit - i.e. outputs are strictly derived from the inputs of the specification, it is possible to animate the specification and to get reference outputs from the inputs. A study of Z specifications collected in Specification Case Studies [25] shows that an overwhelming majority - 94% - of operation schemes are explicit or can be made explicit with minor modifications [24].

Requirements 1 and 2 for the verification specification format for C designs are satisfied. However, it is difficult to use *Z* specifications together with *C* designs: the languages are too different. *C* and especially *C++* are complicated programming languages which can use a number of existing libraries and functions, like file processing functions or numbers with different accuracies. In contrast, Z specifications are more formal and do not contain much of the C functionality. They are used mostly for the scientific research and only rarely for actual design specification.

To summarize: Z notation is a well standardized specification language which is not compatible with the *C* designs, since the requirement 3 for the specification format is not satisfied and it is impossible to use *Z* specifications for *C* designs verification.

2.7.4 Specification using Uppaal

Uppaal [34] is an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata and extended with data types like bounded integers, arrays etc.

The tool has been developed in collaboration between the Design and Analysis of Real-Time Systems group at Uppsala University, Sweden and the Basic Research group in Computer Science at Aalborg University, Denmark [52].

The first release of UPPAAL was in 1999.

UPPAAL is appropriate for systems which can be modeled as a collection of non-deterministic processes with a finite control structure and real-valued clocks, communicating through channels or shared variables. Typical application areas include real-time controllers and communication protocols, in particular those where timing aspects are critical.

Uppaal consists of three components: a modeling language, a simulator and a model-checker together with a mu calculus based specification language. The specification format written in Uppaal is clear and the tool is developed for animation purposes, hence requirements 1 and 2 for the specification of designs are satisfied.

Unfortunately it is necessary for our purposes to specify the *C* designs and includes. Uppaal is not suitable for these purposes: it is too simple for writing a full description of a C design. The requirement 3 is not satisfied, hence Uppaal is not suitable for verifying C designs.

2.7.5 Specification using C as a specification language

As it has been seen from the descriptions of specification languages, the main difficulty of defining/selecting a specification format for *C* designs is specifics of the *C* programming language. The specification language should be able to describe any possible behavior of the *C* design, be able to generate reference outputs from inputs, and to be explicit. *C* design has no functionality for temporal operators; most of its functionality is composed of arithmetical procedures, conditions, functions and file processing.

The best solution for defining a specification of a system with the required properties is to write specifications using the *C* language, the same language as used by designs. Specifications in *C* are easy to create and use. For verification purposes it should be possible to animate a specification: *C* code is executable, fast and reliable.

Specifications using programming languages like *Java*, *C*, *C++* are widely used in the software development industry [33].

If a specification for a *C* design is given in a *C* specification language, then it is easy to produce outputs in the same format as those in the processed design: for that the Observation Points (refer to Chapter 3) can be added to both the design and specification in the same locations.

Internal format of the specification is not strictly defined in case of the specification and any *C* functionality can be described by the specification. As example, specification using *C* specification language can be implemented using test cases for arbitrary functions. Of course, it is most important that outputs of a specification correspond to the outputs of a design, or located in the same locations of the design and specification.

Therefore if C is used as a design language, then it is best to use *C* as a verification language as well. This approach has been implemented in the FORENSIC tool [16] of the DIAMOND project [14, 45].

2.8 Chapter summary

The current chapter presented a background of fault theory and design verification basics, that include hardware and software design verification and development, verification complexity were described briefly also. Simulation-based verification were compared with formal verification, and simulation-based verification were chosen to be used in out FORENSIC [16] tool. Basics of the model-based error localization and mutation-based error correction were introduced, in the next Chapters, Chapter 4 and Chapter 5, those algorithms will be described in details.

Specification format for the error localization and error correction tool were chosen, for that various specification formats, that were possible to implement were investigated, like PSL specification, UML specification and Uppaal specification, and C specification. Were explained the decision why C specification language were chosen to be a specification language for the FORENSIC [16] tool.

Chapter 3. Data structures for implementing error localization and error correction

In order to locate and to correct error in the *C* design, or, more precise, to be able to apply simulation-based verification with error localization and error correction, using various rankings for error localization and various mutation types for error correction in *C* designs, we should either use the functionalities of an existing tool or develop a new environment and to implementing the necessary algorithms within contents of the new tool.

In order to perform design verification, error localization and correction, we need to design and implement the model: a structure, where parsed representation of the verified design. The model is the core data structure of the error localization and correction tool and determines what kind of properties will the tool have, what will be possibilities of the tool, advantages and, possibly, minuses and disadvantages, or, in other words, what types of the designs will be able to process with such a tool, and what will not be able.

In this Chapter structure of the proposed model for the FORENSIC [16] tool will be described in details, with all required mathematical definitions. Later on in this Chapter general structure of the tool will be described, it will be shown how tool is divided into back-end and front-end parts, and implemented functionalities within contents of the tool will be described, that is implementation of the error localization and error correction algorithms. Several approaches are implemented of the FORENSIC [16] tool, one provide higher speed of the processing of the design without losing any functionality, so called more sophisticated approach, other is more simple, both approaches will be described in Subsection 3.4.5.

After other from model data structures, that are required tool's functioning, will be described.

It will be explained why it is more reliable to use simulation of the model using C functionality in order to get designs outputs, than direct simulation of the model [45].

In the next section we will take a closer look at error localization and error correction algorithms, implemented using model and data structures, described in current Chapter.

3.1 The model

The model is used for storing the parsed design representation in a valid "white box" form. Model defines functionalities that can be used for verification, error localization and correction.

The model format, *C++* structure and actual implementation for the FORENSIC [16] tool were designed and developed at the Graz University of Technology by Robert Könighofer and colleagues from the Institute for Applied Information Processing and Communications – IAIK [31, 27].

The parsed representation of the *C* design, the model can be viewed as a graph structure, example of the model with parsed data is shown in *Figure 3.1*. The model consists of *FlowChartNodes* class instances (black and green boxes) where every *FlowChartNode* is a statement or a part of a statement of the original design, that are further parsed into an abstract syntax tree nodes. The *FlowChartNode* can be a *ConditionNode* (green boxes) or an *OperationNode* (black boxes) or an abstract syntax tree node *AstNode* (yellow box). The nodes of the model will be further also called *components of the parsed design*.

Statements of the original design representation can be divided into a small number of sub-statements in the model. The entire model has to be completely in correspondence with the original design and to include the complete functionality of the processed design.

The model, that can be viewed as a hammock graph [28], is defined below:

Definition 1: *Digraph* is a directed graph structure $\langle N, E \rangle$, where N is a set of *nodes and* E is a set of *edges* in $N \times N$. If (n, m) is in E then n is an *immediate predecessor* of m and m is an *immediate successor* of n.

Definition 2: *Flow graph* is a structure $<N, E, n_0>$, where $<N, E>$ is a digraph, n_0 is in N and there exists a path from n_0 to all other edges in N. n_0 is called *the initial node.*

Definition 3: Hammock graph is a structure $H=<N, E, n_0, n_e>$ where both $<N, E, n_0>$ and $<N, E^{-1}, n_e>$ are *flow graphs*. Note that, as usual, $E^{-1}=\{(a,b)|$ (b,a) is in $E\}$. n_e is called *the end node*. [28]

Definition 4: Undirected graph is a structure $U=<N, E>$ where N is a set of nodes amd E is a set of simple edges in $N \times N$. There exists a simple path from an arbitrary node n in N to all the other nodes in N. A path is a list of the nodes $p_0, p_1, ..., p_k$ such that $p_0 = n_i, p_1 = n_{i+1}, \ldots, p_k = n_k$, and for all $i, 0 \le i \le k - 1$, (p_i, p_{i+1}) is in E.

Definition 5: *Abstract syntax tree* $A=<N, E, n_0>$ is a rooted undirected graph $U=<N, E^*>$ with the root n_0 where two nodes are connected exactly by one simple path.

A hammock graph [28] is a special case of the flow graph. The difference between a flow graph and a hammock graph is the following: hammock graph has one end node but a flow graph does not have to obey this rule. Hammock graphs are used to describe structure of the programs, like in [59], where a program is converted into several of hammock graphs by a series of transformations.

In the proposed model structure nodes in the hammock graph are statements or parts of statements of the original design, including assignments, conditions, arithmetical operations etc. This structure of a model is suitable for implementing model-based error localization. For dynamic slicing and the mutation-based error correction algorithm's implementation the model representation should be more detailed, as it is necessary to be able to manipulate with the statement's variables and numbers in the node – as it will be shown further in Chapter 4, for the dynamic slicing algorithm implementation it is necessary to process defined and referenced variables in the original statements. For these purposes every node in a hammock graph in the FORENSIC tool [16] is parsed further into an abstract syntax tree representation, that is a hierarchy of operators and functions with corresponding variables in the statement.

In the FORENSIC tool [16] every hammock graph node N is a statement or a part of the statement of the original design that can be presented using tuples of no more than 3 operands. This structure satisfies all the properties of the hammock graph representation for the dynamic slicing and mutation-

based error correction. This parsed structure of the processed C design can be obtained using GIMPLE GCC plug-in [17], and further is stored in the model, refer to the Subsection 3.4.1 of this Chapter. Front-end parser of the FORENSIC [16] tool were designed and implemented at the University of Bremen by Alexander Finder within the DIAMOND project [14].

An illustration of the model implemented in the FORENSIC tool is shown in *Figure 3.1*.

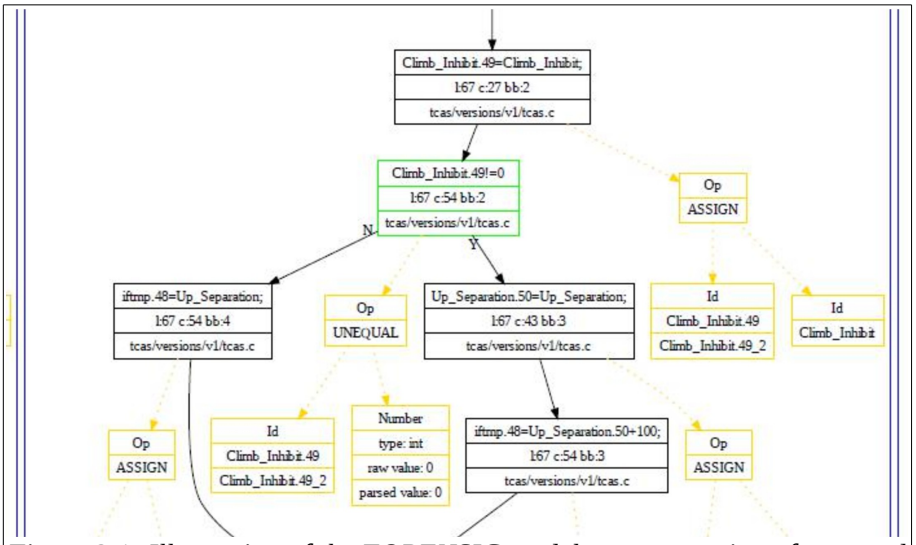

Figure 3.1: Illustration of the FORENSIC model: representation of a parsed design. Black and green boxes are hammock graph nodes, yellow boxes are abstract syntax tree nodes.

Mathematical structure of the FORENSIC [16] tool's model is published in [45].

3.2 Animation and simulation

Animation and simulation are the most time-consuming phases of the simulation-based verification, model-based error localization and mutation-based error correction algorithms. Therefore it is best to implement them as reliably and as well-optimized as possible. Simulation is responsible for the process of obtaining of the outputs from the inputs using the design or a model of the design. Animation is a process of getting reference outputs from

inputs using the specification.

Simulation of the design model using the *C* compiler functionality is a novel approach suggested within contents of the dissertation and implemented in the FORENSIC tool [16].

Basically two various ways of the model's simulation proposed in the dissertation: direct simulation of the model and simulation of a model using the functionality of the design language.

If to use a direct simulation, when nodes of the model are activated from start node till the end node, according to the inputs provided, then logical operations in the model have to be implemented and the values of the activated variables have to be stored somewhere during a simulation. Variables can have different types: pointers, arrays, compound data types and so on. In practice the memory model of the programming language of the design should be re-implemented in order to store the values during simulation. Hence it is reasonable to use the already existing memory model of the compiler, avoiding re-implementation of the functionality of the programming language of the design.

In order to to simulate a model using the functionality of the compiler, the model has to be instrumented in order to get necessary outputs, dumped into C program file and compiled into an executable file. Further executable can be executed, using inputs provided. The output results must be written to output files reachable after execution. This allows us to access all data about the execution process without losing any functionality. Otherwise, we get benefits like the speed of execution and the simplicity of the approach. We will not get any programming errors that can appear if we re-implement already existing functionality of the programming language. This approach - using the functionality of the compiler – allows us to implement dynamic slicing for the error localization in the FORENSIC tool [16]. [45]

Additionally, file processing operations in the original design can be substituted for by appropriate special *FORENSIC_...()* functions, that are observation points, and executed as ordinary functions, when compiler functionality is used for the simulation, refer to next section.

The same principles for the simulation of the model can be applied to designs written in arbitrary programming languages like Java/PHP/Fortran.

3.3 Observation Points and file processing

Any verification tool should allow the insertion of the observation points into the processed design. Observation points are locations where the behavior of the design is compared with the correct behavior. This property is

critical if we want to receive information about the behavior of the design at any point. An observation point is usually inserted at a critical locations of the program according to the specification.

Observation points can be divided into two categories – Input observation points and output observation points:

Input observation points can be:

- *FORENSIC_input_....()* functions – special functions.
- *stdin* functions in the original design – the functions of the programming language that are responsible for the standard input.

Output observation points can be:

- *FORENSIC_output_....() functions* – special functions.
- *assert()* statements.
- *stdout* functions in the original design – the functions of the programming language that are responsible for the standard output.

In order to use the special functions at the observation points locations of the processed design and, therefore, of the model, corresponding function's implementations file names should be included at the beginning of the dumped instrumented for simulation model's code or at the beginning of the specification code, as we use specification in the *C* format, refer to the Chapter 2. The implementations file of the special functions must be present in the third-party library of the corresponding tool, that executes instrumented model. This design, that contains special functions, is not compilable with an ordinary compiler any more, but is compilable and executable with the verification tool.

Input observation points define locations where data is read from the input file during simulation. In the FORENSIC [16] tool every input for every simulation is located in a single line in the input file. If there should be several lines in the one input for one particular simulation, then line breaks can be defined using “\n” marker in the input file to separate them. Input for each input observation point is separated by space “ ”. Example input file structure is following, inputs for two simulations:

```
abc 123 56 23 \n def 765 234 567 \n jhi 876 3 235
cba 6544 2 98 \n fed 234 876 234 \n ihj 234 890 276
…....
```

Output observation points define the output locations of the design and specification. The points can be viewed as places where we decide whether the design satisfies its specification or not, as those outputs are compared later

for verification. If an output observation point is defined using a special function in the original design, then the front-end parser should not add them into the parsed model representation, as they are observation points only and we don't care about their actual implementation. The special assert() and stdout output observation points should be parsed as usual functions.

In case there are only assert() output observation points in the design, then there is no need for the specification for verification for this type of the design, as no file output is performed and no comparison with the reference outputs and actual outputs is performed. The decision whether verification passed or failed is made during a simulation of the design and succeeds if no assertions are violated, refer to *Figure 3.2.*

3.4 Tool structure

The FORENSIC[16] (FOrmal Repair ENgine for SImple C) tool is an environment for implementing various error localization and error correction approaches for the *C* designs.

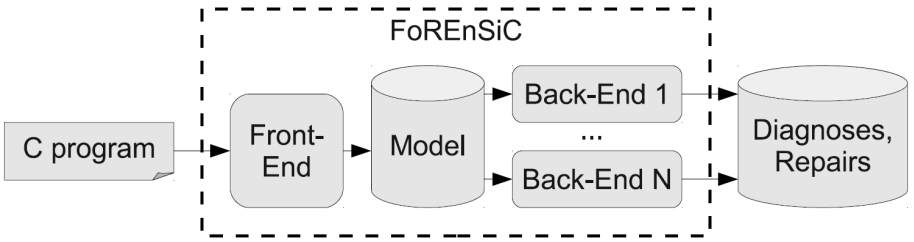

Figure 3.2: The architecture of the FORENSIC tool [27].

FORENSIC consists of various back-ends, and of a front-end parser, as can be seen in *Figure 3.2*. Front-end component is responsible for parsing of the design into the model representation, back-end(s) are responsible for error localization and error correction in the parsed model representation. In the current Chapter we pay most of attention on simulation-based back-end, but alternative back-ends, implemented within contents of the FORENSIC [16] tool of the DIAMOND [14] project will be described briefly also, those are symbolic back-end and equivalence checking back-end, refer to next Subsections.

3.4.1 Front-end

Before the error localization and correction can be started at the back-ends, it is necessary to parse the processed design into the model representation.

The GIMPLE plug-in of the GCC compiler [17] allows to obtain a three-address representation of the *C* design with tuples of no more than 3 operands. This representation of the design is suitable for storing in the model, as it contains all necessary information about the processed design.

The front-end parser for the C designs were designed and implemented at the University of Bremen by Alexander Finder within the DIAMOND project [14]. The front-end parser of the tool can be theoretically extended to support *C++* and *SystemC* designs.

The model structure is described in detail in Section 3.1.

3.4.2 Symbolic back-end

The symbolic back-end implements the debugging method of [31].

Verification and debugging are performed using symbolic or concolic execution, that is used to compute a formula defining when the program conforms to its specification. After, model-based diagnosis engine computes potentially faulty components [42]. Finally, the repair engine synthesizes new implementations of the faulty components. For additional descriptions refer to FORENSIC tool's manual at [16].

3.4.3 Equivalence checking back-end

Often a specification which defines the behavior of a circuit is given in a higher language like C/C++. From such a specification, an implementation can be derived through conversion to HDL. This back-end formally verifies the equivalence between a C program taken as reference specification and an implementation in HDL or vice versa. Similarly, two different implementations of a C program can be verified for equivalence. For additional descriptions refer to FORENSIC tool's manual at [16].

3.4.4 Simulation-based back-end

Simulation-based back-end is divided into error localization and error correction components, in such a way that the outputs of error localization component are used as input for the error correction component.

Model-based error localization component provides the error localization functionality of the simulation-based back-end and is responsible for the

generation of candidates for correction. Mutation-based error correction component uses generated candidates for correction and performs error correction in the processed design.

The implementation of the simulation-based back-end consists of several components. Those are:

- Inputs-outputs, i.e. data, that is required and used for simulation-based verification, error localization and error correction.
- Data structures, where data is stored during processing. Those include the model, that is the most significant data structure in the tool.
- Functionality of the various implementations of error localization, correction and simulation algorithms. The functionality is well developed and documented and can be further extended if necessary.

3.4.5 Algorithms and functional components

Figure 3.2a shows a basic tool structure for simulation based verification, a more sophisticated tool structure used in the FORENSIC [16] tool is given in *Figure 3.2b*. Both tool structures are implemented in the FORENSIC [16] tool, the basic approach is simpler and the detailed one is faster. The detailed approach is used for experiments, presented in Chapter 6.

Difference between the two tool structures is the following: in a) structure simulation of the processed design is used to generate outputs, while in b) there is no independent simulation of the design with the goal to get outputs. The process of the error localization is started immediately after parsing of the design into the model representation, and during simulation of the model outputs for verification are obtained.

The structure b) is more reasonable for experiments, as both error localization and correction algorithms require the simulation of the model. On the other hand, the advantage of structure a) is that the original design is used to get outputs for error localization, not a parsed representation, avoiding errors that can appear during parsing of the design into the model. But if design is parsed correctly, then outputs of the model simulation (parsed design) and original design simulation should be the same. In the actual implementations of tool back-ends any of proposed algorithms can be re-implemented and extended.

In both a) and b) representations simulation of the model is implemented using C functionality, how it is described in Section 3.3.

Inputs and outputs are marked in *Figure 3.3* as rectangles with white background, functionality in with light gray and data structures with dark

gray background. Let's take a closer look at the components.

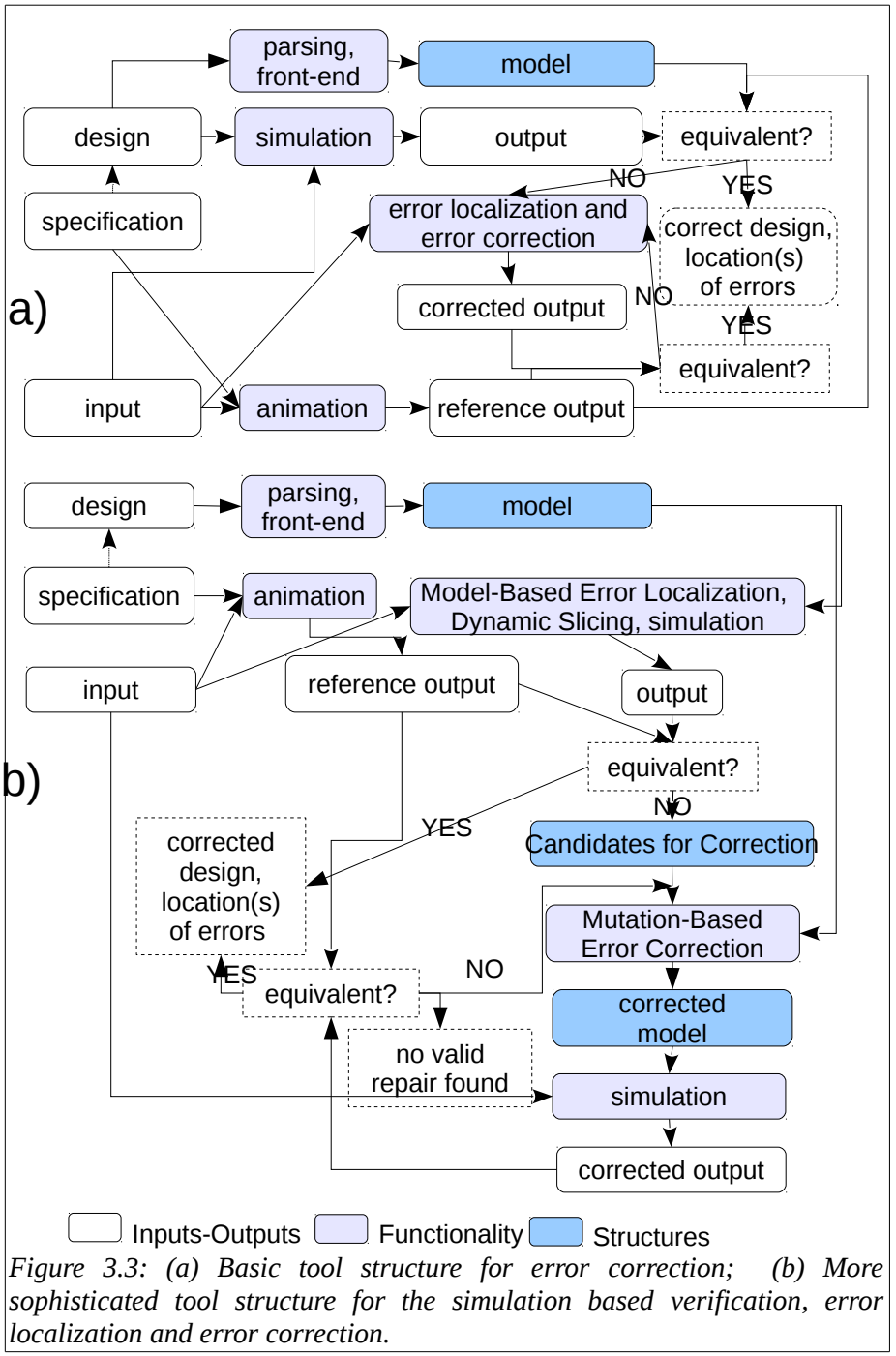

Figure 3.3: (a) Basic tool structure for error correction; (b) More sophisticated tool structure for the simulation based verification, error localization and error correction.

3.4.6 Inputs

Inputs and outputs are denoted as rectangles with white background in *Figure 3.3*. The inputs are required to be able to start processing, and should be provided at the time when the tool is launched.

Inputs are divided into following components:

- Design

The C design should be C code with appropriate include files for special instrumentation FORENSIC_...() functions, that are observation points. Included files of the design are not parsed and are not processed during error localization and error correction, but are included for correct design simulation and/or specification animation.

- Specification

Reference outputs are obtained during the animation of the specification. The verification language in the simulation-based back-end of FORENSIC is the C programming language. If not provided then reference outputs should be provided separately.

- Inputs

Inputs for the design and specification need to be in a valid format. The format of the inputs is defied by the location of the observation points in the processed design, refer to Section 3.3. In addition the *C* arguments can also be provided for the processed design.

Two types of inputs exists for the *C* designs in the FORENSIC tool [16]:

- File "..._file", contains data that is read during simulation by the design, at input observation points
- File "..._arg" contains arguments that are passed to the design/specification/model during simulation.

3.4.7 Outputs

Inputs and outputs are denoted as rectangles with white background in *Figure 3.3*.

Outputs is data that is generated during simulation of the model or execution of the processed design in the output observation points.

Output file contents are defined by the location of the output observation points in the design. The output observation points are defined using the *FORENSIC_output_...()* functions and *assert()* statements of *stdout* functionality of the *C* programming language, refer to Section 3.3.

Outputs are divided into following components:

- Reference outputs

Data, that is generated during the animation of the specification of the processed design. Observation points are used in the exactly same way as for outputs. Locations of the observation points in the specification and the design should match in order to be in valid for comparison and verification.

- Data structures

Data structures contain a hierarchy of C++ classes that is a model and candidates for correction, that are created by the error localization algorithm to be used during the error correction algorithm's execution.

- The model

The model is described in detail in Section 3.1.

- Candidates for correction

Back-end is divided into independent error localization and error correction components. Error localization is a functionality, that generates candidates for correction, and those candidates are used during the error correction algorithm's execution.

Candidates for correction is a data structure where the locations of the components of a model are stored with any required corresponding data like the rank of the component or the number of times *failed* or *passed* for the component during the process of error localization (refer to Section 4.2).

3.4.8 Functionality

Front-end parser is responsible for parsing functionality, that were described in Subsection 3.4.1.

Simulation and animation functionalities were described in Section 3.2.

Model-based error localization and mutation-based error correction functionalities will be described in details in Chapter 3 and Chapter 4.

3.5 Chapter summary

In the beginning of this Chapter the structure of the proposed model for the FORENSIC [16] tool were described in details, with all required mathematical definitions. It were explained why existing structure were chosen.

Later on in this Chapter structure of the FORENSIC tool [16] were described. It were shown how reference outputs are obtained from

specification using animation, how design is parsed into the model and later on outputs are obtained using inputs and simulation of the model using *C* functionality.

Several implementation approaches of the tool are introduced in the Chapter, one is simpler and more reliable but slower, second one is more sophisticated and faster but more error-dependent. Both approaches are implemented, and it is possible to choose tool's structure that is more suitable for any tool user.

The concepts and structures, described in this Chapter, can be seen as a development framework for simulation-based verification, error localization and error correction implementations.

Chapter 4. Model-based error localization algorithm

When the design verification fails, a two-step process is initiated, that consists of first determining the location of the error – error localization, and then correcting the error – error correction [38]. Support for error localization and error correction algorithms were implemented in the FORENSIC [16] tool in the simulation-based back-end in Tallinn University of Technology by the author of this dissertation. The current Chapter will give a detailed description of the model-based error localization algorithm and will provide details of the implementation. Next Chapter, Chapter 5, will give details of the error correction algorithm, implemented within contents of the FORENSIC [16] tool. Same principles can be applied to Designs, written in arbitrary programming languages like Java/PHP/Fortran.

Error localization and error correction implementation are one of the major achievements of the DIAMOND FP7 [14] project, implemented in the FORENSIC [16] tool.

Model-based error localization algorithm uses the simulation trace of the model with various inputs. It assigns *passed* or *failed* counters to nodes that belong to every simulation trace. Finally, various rankings are applied to the nodes using counters. Nodes that have highest ranks are candidates for correction and are stored in the corresponding data structure.

In the related works – [1] spectrum-based error localization algorithm were defined, that is similar to the FORENSIC [16] tool's error localization algorithm, algorithms will be compared and it will be shown, that they are actually same, but have different mathematical representation, and should produce same experimental results with same inputs. In the Chapter 6

experimental data is compared.

4.1 Model-based error localization algorithm

Error localization is triggered when outputs of the processed design do not match the reference outputs or violate some *assert()* statements during the verification phase, refer to *Figure 3.2*. Various algorithms can be used for error localization. In the current dissertation we focus on the model-based error localization algorithm.

As it were shown in Chapter 3 a model is a control flow graph of a program is a hammock graph [28], (refer to *Definition 3)*. During each run of model-based error localization with particular inputs, some nodes in the model get traversed, i.e. get activated. The activated nodes are defined as follows:

Definition 6: *Activated nodes* $N_a \subseteq N$ *are the nodes of the* model – a hammock graph $H=<N, E, n_0, n_e>$ *that when simulating the model with inputs* $s \in S$, belong to the activated path P from node n_0 to node n_e. The activated path P from node n to node m is a sequence of nodes with length $k+1$ $p_0, p_1, ..., p_k$, such that $p_0 = n$, $p_k = m$, and the transitions for all j, $0 \le j \le k-1$, $(p_j, p_{j+1}) \in E$. The a*ctivated path* P_a for input s is a path resulting in simulation with input s.

For the model-based error localization algorithm's description we need to introduce definition of algorithm's sorting procedure:

Definition 7: Assume that we have nodes N, every node has specific property r, that is integer or floating-point type number. Assume that we want to sort nodes N according to property r, when property r is increasing from minimum to maximum or deceasing from maximum to minimum. Assume that all property values r belong to R. Then sorted list of nodes N, named N_s, is presented using notation $N_s = \left\langle \begin{pmatrix} max(R) \dots min(R) \\ N \end{pmatrix} \right\rangle$, when list is sorted by property R in decreasing order, or using notation $N_s = \left\langle \begin{pmatrix} min(R) \dots max(R) \\ N \end{pmatrix} \right\rangle$ when list is sorted by property R in

increasing order.

Algorithm 1: Model-based error localization. During the model-based error localization algorithm's execution the model is simulated with inputs S and for each input $s \in S$ output of the simulation is compared with reference output of the specification. If comparison, i.e. verification, fails, then activated nodes $n \in N_a \in N$ of the hammock graph $H=<N, E, n_0, n_e>$ have *failed* counter increased, otherwise nodes have *passed* counter increased. During simulation with all inputs S set of total activated nodes N_t is accumulated by applying union of N_a for each input $s \in S$, and after ranking algorithms are applied to the counters.

The list of candidates for correction C_s, that is a sorted list of the activated with all inputs S nodes N_t, is calculated for all ranking algorithms, except consecutive, as follows:

For every node n, that is in N_t, rank r is calculated, using *passed* and *failed* information, R is set of all ranks, r is in R, and then $C_s = \left\langle \begin{pmatrix} max(R) \dots min(R) \\ N_t \end{pmatrix} \right\rangle$.

The list of candidates for correction C_s for the consecutive ranking algorithm is calculated as follows:

For every node n, that is in N_t, primary rank r_1 and secondary rank r_2 are calculated, using *passed* and *failed* information, R_1 is set of all primary ranks, R_2 is set of secondary ranks, and then, nodes initially sorted by the primary rank, $C^* = \left\langle \begin{pmatrix} max(R_1) \dots min(R_1) \\ N_t \end{pmatrix} \right\rangle$. Later on, sorting using secondary ranking is applied to the nodes, that have same primary ranking value. During this second sorting C_s is accumulated using operation of addition to the end of the already sorted list (push):

$$C_s = \emptyset,$$
$$c = 1,$$
$$r1 \in R_1^* \in C^*, r2 \in R_2^* \in C^*,$$
$$\forall r1_i \in R_2^* \in C^*, i = 1 \ldots |N_t^*| - 1:$$
$$if\ r1_i \neq r1_{i+1}:$$
$$C_s = C_s \leftarrow \left\langle \begin{pmatrix} max(r2_c \ldots r2_i) \ldots min(r2_c \ldots r2_i) \\ N_t^* \end{pmatrix} \right\rangle,$$
$$c = i + 1,$$
$$C_s = C_s \leftarrow \left\langle \begin{pmatrix} max(r2_c \ldots r2_{|N_t^*|}) \ldots min(r2_c \ldots r2_{|N_t^*|}) \\ N_t^* \end{pmatrix} \right\rangle.$$

When sorting algorithm's definition is done, we can define model-based algorithm using this definition. I will use two different definitions when consecutive ranking is used for the model-based error localization, and if any other ranking, except consecutive is used. Type of ranking to be used for the particular tool's execution can be setted in the tool's parameters (settings).

Ranking's definitions will be given further in the next subsection.

Algorithm's 1 *execution representation* 1: *Model* − *based error localization* ,
Any ranking exept consecutive.
For ranking definitions refer to *the next pharagraph.*

N_a − *activated nodes of the hammock graph* $H = \langle N, E, n_0, n_e \rangle$
$s \in S$ − *inputs.*

$N_t = \varnothing$,
$\forall s \in S$:
N_a *is generated* and *design is verified*
if verification passes :
$\forall n \in N_a$ *passed* = *passed* + 1
else
$\forall n \in N_a$ *failed* = *failed* + 1
$N_t = N_t \cup N_a$

$\forall N_t$ *rank* $r \in R$ *is calculated.*

Candidates for correction are generated , $C_s = \left\langle \begin{pmatrix} max(R) \ldots min(R) \\ N_t \end{pmatrix} \right\rangle$

Algorithm's 1 *execution representation* 2 : *Model* − *based error localization*,
Consecutive ranking.
For ranking definitions refer to *the next pharagraph*.

N_a − *activated nodes of the hammock graph* $H = \langle N, E, n_0, n_e \rangle$
$s \in S$ − *inputs*.

$N_t = \emptyset$
$\forall\, s \in S$:
  N_a *is generated* and *design is verified*
  if verification passes:
    $\forall\, n \in N_a$ $passed = passed + 1$
  else
    $\forall\, n \in N_a$ $failed = failed + 1$
  $N_t = N_t \cup N_a$

$\forall\, N_t$ *primary rank* $r_1 \in R_1$ and *secondary rank* $r_2 \in R_2$ *are calculated*, and N_t *is sorted by the primary rank* R_1, and *result is stored as* C^*:

$$C^* = \left\langle \begin{pmatrix} max(R) \dots min(R) \\ N_t \end{pmatrix} \right\rangle$$

Then sorting using secondary ranking $r2 \in R_2^*$ *is applied* to C^*:
$C_s = \emptyset$,
$c = 1$,
$r1 \in R_1^*, r2 \in R_2^*$,
$\forall\, r1_i \in R_1^* \in C^*, i = 1 \dots |N_t^*| - 1$:
  if $r1_i \neq r1_{i+1}$:

$$C_s = C_s \leftarrow \left\langle \begin{pmatrix} max(r2_c \dots r2_i) \dots min(r2_c \dots r2_i) \\ N_t^* \end{pmatrix} \right\rangle,$$

  $c = i + 1$,

$$C_s = C_s \leftarrow \left\langle \begin{pmatrix} max(r2_c \dots r2_{|N_t^*|}) \dots min(r2_c \dots r2_{|N_t^*|}) \\ N_t^* \end{pmatrix} \right\rangle.$$

Simulation-based verification algorithm therefore requires simulation of the model and the animation of the specification, and simulation is implemented using execution of the instrumented code, that is obtained by dumping of the model into C executable, refer to Section 3.2.

It is possible to apply additional technique to the activated during model-based error localization nodes – dynamic slicing, that increases error localization accuracy, that is shown in Section 4.3.

4.2 Rankings for the model-based error localization

There are several different ranking functions defined in the literature. The rankings perform differently in the case only one error or several errors are assumed to be present in the design. Those errors can further be attempted to be corrected by the error correction back-end.

The ranking functions implemented in FORENSIC are the following:

- **Ample** ranking[1]:

$$rank(n)=\left|\frac{failed(n)}{totalfailed}-\frac{passed(n)}{totalpassed}\right| \quad (4.2.1)$$

- **Consecutive** ranking[45]:

$$rank1(s)=\frac{failed(n)}{totalfailed}$$ Simple Ranking is followed by

$$rank2(s)=\frac{failed(n)}{\sqrt{totalfailed\cdot(passed(n)+failed(n))}}$$ Ochiai Ranking (4.2.2)

- **Jaccard** coefficient, Pinpoint framework ranking[1, 8]:

$$rank(s)=\frac{failed(n)}{passed(n)+totalfailed} \quad (4.2.3)$$

- **Ochiai** coefficient, used for computing genetic similarity in the molecular biology[1]:

$$rank(s)=\frac{failed(n)}{\sqrt{totalfailed\cdot(passed(n)+failed(n))}} \quad (4.2.4)$$

- **Simple** ranking [45]:

$$rank(s)=\frac{failed(n)}{totalfailed} \quad (4.2.5)$$

- **Tarantula** tool ranking [12]:

$$rank(s)=\frac{\frac{failed(n)}{totalfailed}}{\frac{passed(n)}{totalpassed}+\frac{failed(n)}{totalfailed}} \quad (4.2.6)$$

where *rank* of the activated node $n \in N_a \subseteq N$ is calculated using *passed(n)* – times passed for node *n*, *failed(n)* – times failed for node *n*, *totalpassed* – total passed simulations with inputs *S* and *totalfailed* – total failed simulations with all inputs in *S*.

For the consecutive ranking function any secondary ranking from the proposed ones can be used, Ochiai is shown as an example. The consecutive ranking function corresponds to Heuristic II and III, defined in [41], simple ranking was first introduced in [45, 43].

Comparison of similar ranking methods and error localization experiments using Siemens designs [6] is presented in [1] and is discussed later in this Chapter.

4.3 Dynamic slicing for the model-based error localization

Dynamic slicing is a technique that is applied to the activated nodes obtained during the simulation phase of model-based error localization and allows to reduce the number of nodes in the candidates for correction. The idea behind the dynamic slicing algorithm is the following: some amount of nodes is activated during simulation of model-based error localization, but do not have any influence on the simulation result. Those can be constant declarations, assignments inside code and etc. Dynamic slicing allows to discard those nodes and, therefore, corresponding statements in processed

design [44, 45].

In order define algorithm of dynamic slicing we need to introduce following definitions - defined *DEF(n)* and referenced *REF(n)* variables of the node $n \in N_a \subseteq N$ [53].

Definition 8: Let *V* be the set of variables, that are present in the model of the design, represented by a *hammock graph* $H=<N, E, n_0, n_e>$. Then for every node $n \in N_a \subseteq N$, two sets can be defined, each of them is in *V*: *REF(n) – referenced variables* - set of variables, whose values are used or referenced in *n*, and *DEF(n) - defined variable* - variable, where value is assigned to in *n* [53].

Definition 9: Dynamic slice of $N_a \subseteq N$ of the flow *graph* $H=<N, E, n_0>$ of the simulation with input $s \in S$ is a set of nodes, that have an influence on the simulation output. The list of d*ynamic variables - DV(n)* at the moment of processing of the node $n \in N_a \subseteq P_s$ during the execution of dynamic slicing includes variables, that have an influence on the output of further simulation, on processing of the nodes $(m_{i+1}, ..., m_e) \in N_a \subseteq P_s$, where $m_i=n$, ..., $m_e=n_e$.

Algorithm 2: Dynamic slicing. Let input $s \in S$ be the input that leads to an execution which activates nodes $N_a \subseteq P_s$. Nodes of the model are denoted by $n \in N_a$. The variables, that are in the set of dynamic variables - *DV(n)*, are updated when processing of each node *n.* Let *DV(n)* be the set of dynamic variables at the beginning of the processing of the node *n, and* let *DV*(n)* be the set at the end. The algorithm proceeds from the end activated node n_e to the initial activated node n_0, and $DV(n_{i-1}) \equiv DV^*(n_i)$, where $e \le i \le 1$. At the beginning of execution the sets of dynamic variables are empty, $DV(n_e)=\varnothing$, and $DV^*(n_0)=\varnothing$.

If *n* contains a *C* function or a condition, then the referenced variables *REF(n)* of the node *n* are added to the set of dynamic variables *DV(n)*: $DV^*(n) = DV(n) \cup REF(n)$. The output observation points or *assert()* statements can be last executed component - n_e - and in this case output variables should be initially stored in the list of dynamic variables $DV^*(n_e)$.

If a node *n* contains an assignment, then the referenced variables *REF(n)*

of the assignment are added to the set of dynamic variables *DV(n)* if and only if the defined variable *DEF(n)* of the assignment is already in *DV(n)*, otherwise node *n* is *not in the dynamic slice* and the list of dynamic variables is propagated further, *DV*(n)= DV(n)*. If *DEF(n)* is in *DV(n)* then defined variable *DEF(n)* should be removed from *DV(n)* before *REF(n)* addition: iff $DEF(n) \in DV(n)$ then $DV^*(n) = (DV(n) \setminus DEF(n)) \cup REF(n)$, otherwise $DV^*(n) = DV(n)$ and *n* is *not in the dynamic slice*.

Nodes, that are *not in the dynamic slice* can be removed from $N_a \subseteq P_s$ without influencing the output of simulation.

n example of *dynamic slicing* algorithm's execution is at Figure 4.1.

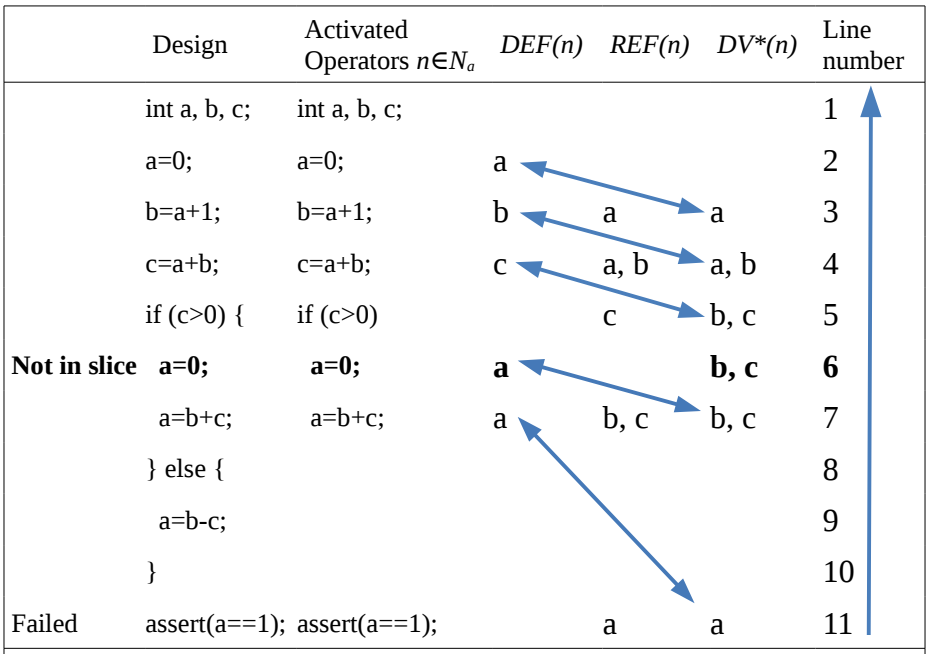

Figure 4.1. An example of the dynamic slicing algorithm's execution.

Depending on the type of the processed design, dynamic slicing can decrease the number of nodes in the candidates for correction data structure that is the result of error localization. A number of assignments can be made at the initialization part of the design, e.g. initializations of constants, arrays, and so on, and not all of initialized values may be used during simulation. The corresponding assignments will be discarded by the result of slicing.

In the implementation of the dynamic slicing algorithm in FORENSIC, the variables are presented using unique addresses, not variable names, as shown in *Figure 4.1*, and this makes it possible to process arrays and any other compound structures. Using the compiler's functionality for simulation makes this implementation possible. Dynamic slicing was first proposed in [30]. Since then it has been applied in several contexts. In [60] a dynamic slicing approach working on binary level has been evaluated. Such approach allows the slicing to be traced back to any programming language that is supported by the GCC compiler. Comparison of experiments, obtained in [60] and by FORENSIC tool [16] is in Chapter 6.

Empirical evaluation of the effects of dynamic slicing with the FORENSIC tool confirm that the slicing procedure can reduce the number of candidates and therefore the number of required mutations to correct an error in the design.

4.4 Error localization and error correction

In [12] - Vidroha Debroy and W. Eric Wong show how to use mutations to automatically suggest fixes for faulty programs. Tarantula [12, 1] tool and the corresponding ranking was used for model-based error localization . Mutation-based error correction was used to correct errors in C designs supporting mutations only in the operators.

In the current dissertation six various rankings are investigated, including Tarantula [12, 1] tool ranking, and mutations that were used for mutation-based error correction in the FORENSIC tool include additionally mutations in numeric values, constants, and up and down rounding of floating point numbers.

Experiments were performed using the same designs – Siemens designs [6]. The experimental results are compared in Chapter 6.

4.5 Related works — Spectrum based error localization

A number of rankings can be applied during the model-based error localization to extract most probable erroneous nodes by sorting the activated nodes of the model of the design. Four independent ranking algorithms were proposed in [1]. The goal of the work was not only to correct errors as in [12], but to compare ranking coefficients for model-based error localization. Coefficients were taken from various debugging tools – Pinpoint [8], Tarantula [12], and AMPLE (Analyzing Method Patterns to Locate Errors)

[10], and from the molecular biology domain (Ochiai coefficient). The Siemens designs that are targeted in the current dissertation were also used for experiments in [12, 1].

A program spectrum, defined in [46] also, is used for error localization experiments presented in [1]. “The obtained spectra, are called activated nodes and counters in the terminology of the current dissertation. The spectra are used to construct a binary matrix of size $M \times N$: M is the length of the vector of inputs and corresponds to the number of rows of Spectra, the number of columns corresponds to N blocks, that design consists of. Let us assume that during the simulations with some inputs an error is detected (simulation *failed*) and some simulations finish without any errors (simulation *passed*). The information is stored in another binary column vector E of length m. For each block of the design a similarity s_j is calculated from the column vector with error vector. Blocks with the highest similarity are most likely to be erroneous”. [1] In the terminology of the current dissertation, the similarity s_j is the rank of an activated node.

In [1] spectra is defined as:

$$M\,Spectra \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1N} \\ x_{21} & x_{22} & \cdots & x_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ x_{M1} & x_{M2} & \cdots & x_{MN} \end{bmatrix} \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_M \end{bmatrix} \tag{3.2}$$

Every component of block i is spectra element x_{ij}, where $i \in 1...M$ and j $\in$ $1...N$, has value 1 if it were activated and 0 if it were not activated during simulation with input j. Values e_j have error detection values information, have value 1 if simulation were *failed* and 0 if corresponding simulation *passed*. Later on, using notation in [22], four counters a_{11}, a_{10}, a_{01} and a_{00} can be introduced for every element of spectra x_{ij}, that correspond to four different cases – if element were activated and/or passed/failed, as shown below in *Table 4.1*:

Table 4.1. Counter definitions as defined in [1].

	values:	
counter:	x_{ij}	e_j

a_{11}	1	1
a_{10}	1	0
a_{01}	0	1
a_{00}	0	0

Using counters similarity coefficients are calculated. This type of representation for error localization is called spectrum-based error localization and widely investigated nowdays, refer to for details [26, 39, 58].

For FORENSIC tool's rankings a different terminology is used. The rank of an activated node is calculated using information obtained during simulation of model-based error localization.

In *Table 4.2* the similarity coefficient's of [1] definitions are presented adjacent to the corresponding ranking definitions implemented in FORENSIC.

Table 4.2. Comparison of ranking functions for FORENSIC model-based error localization and in [1].

Similarity coefficients s_j [1], spectrum-based error localization	*Ranking functions used in FORENSIC* [16], model-based error localization
$s_j = \frac{\frac{a_{11}}{a_{11}+a_{01}}}{\frac{a_{11}}{a_{11}+a_{10}}+\frac{a_{10}}{a_{10}+a_{00}}}$	$rank(n) = \frac{\frac{failed(n)}{totalfailed}}{\frac{passed(n)}{totalpassed}+\frac{failed(n)}{totalfailed}}$
$s_j = \frac{a_{11}}{a_{11}+a_{01}+a_{10}}$	$rank(n) = \frac{failed(n)}{passed(n)+totalfailed}$
$s_j = \frac{a_{11}}{\sqrt{(a_{11}+a_{01})\cdot(a_{11}+a_{10})}}$	$rank(n) = \frac{failed(n)}{\sqrt{totalfailed\cdot(passed(n)+failed(n))}}$

$s_j = \left\| \frac{a_{11}}{a_{01}+a_{11}} - \frac{a_{10}}{a_{00}+a_{10}} \right\|$	$rank(n) = \left\| \frac{failed(n)}{totalfailed} - \frac{passed(n)}{totalpassed} \right\|$
Counters a_{ij} are defined in *Table 4.1*.	rank of the activated node *n* is calculated using *passed(n)* – number of *passed* simulations for node *n*, *failed(n)* – number of *failed* simulations for node *n*, *totalpassed* – total times *passed* for all inputs and *totalfailed* – total times *failed* for all inputs.

In *Table 4.2*. the first row is the ranking used in the Tarantula tool [12], the second is the Jaccard ranking used in the Pinpoint framework [1, 8], the third is the Ochiai coefficient used for computing genetic similarity in molecular biology [1], and the last is the Ample ranking [1].

Let's re-present our novel simple and consecutive rankings in this spectrum representation, refer to *Table 4.3*.

Similarity coefficients s_j [1], spectrum-based error localization	*Ranking functions used in FORENSIC* [16], model-based error localization
$s_j = \frac{a_{11}}{a_{11}+a_{01}}$	$rank(n) = \frac{failed(n)}{totalfailed}$
$\boxed{s1_j = \frac{a_{11}}{a_{11}+a_{01}}}$ Simple Ranking is followed by $\boxed{s2_j = \frac{a_{11}}{\sqrt{(a_{11}+a_{01})\cdot(a_{11}+a_{10})}}}$ Ochiai Ranking	$\boxed{rank1(s) = \frac{failed(n)}{totalfailed}}$ Simple Ranking is followed by $rank2(n) = \frac{failed(n)}{\sqrt{totalfailed \cdot (passed(n)+failed(n))}}$ Ochiai Ranking
Counters a_{ij} are defined in *Table 4.1*.	rank of the activated node *n* is calculated using *passed(n)* – number of *passed* simulations for node *n*, *failed(n)* – number

	of *failed* simulations for node *n*, *totalpassed* – total times *passed* for all inputs and *totalfailed* – total times *failed* for all inputs.

In *Table 4.2.* the first row is the novel simple ranking, the second is the consecutive ranking.

4.6 Chapter summary

The principles of model-based error localization were described in the Chapter. The concepts are applicable for any tool, that needs to localize errors in program source code. The rankings can be applied for code written in an arbitrary programming language, including C/C++, Java. Concepts include rankings for model-based error localization and the definition of the principles of dynamic slicing for error localization. It is shown how data structures, discussed in the previous Chapter, can be used for model-based error localization.

Model-based error localization algorithm in the FORENSIC [16] tool uses the simulation trace of the model with various inputs. It assigns passed or failed counters to nodes that belong to every simulation trace. Finally, various rankings are applied to the nodes using counters. Nodes that have highest ranks are candidates for correction and are stored in the corresponding data structure.

In the related works – [1] spectrum-based error localization algorithm were defined, that is similar to the FORENSIC [16] tool's error localization algorithm, algorithms were compared and were shown, that they are actually same, but have different representation, and should produce same experimental results with same inputs. In the Chapter 6 experimental data is compared.

Chapter 5. Mutation-based error correction algorithm

Various error correction algorithms were implemented within contents of the FORENSIC [16] tool in the simulation-based back-end in Tallinn University of Technology by the author of this dissertation. In current Chapter mutation-based error correction algorithm will be described.

When the error localization has produced candidates for correction it is possible to try to correct the error in the design using mutation-based error correction algorithm. Mutation is a process of inserting syntactically-correct functional changes into the design [13]. In order to change the design, it is necessary to have access to the corresponding component of the design. Thus it is necessary to have a parsed model with high accuracy, when logical operators and variables are parsed into the model. FORENSIC [16] tool model is fully suitable for this purposes, refer to Chapter 3.

Mutation-based error correction algorithm is an error-matching approach. Error-matching means that algorithm cannot correct errors, that are included in the set of supported error classes [44]. Description of used error classes in the FORENSIC [16] tool is given within contents of the Chapter.

Mutation-Based error correction algorithm is model dependent, this means that if error will be corrected or not depends on how the design is presented in the model. Some errors can not be corrected only because of specific representation of the design in the model, that is shown by the example in current Chapter.

Number of mutations, applied during mutation-based algorithm's processing, is more accurate meter of the error localization accuracy than number of statements, required to be processed in order to reach the error location. The metric is more accurate in the sense that every statement can

contain many different operators in original design, and, therefore, many different nodes in the corresponding model, and many different possible correctional mutations, and number of mutations, applied for error correction, should be more “realistic” meter.

Difference between for one fault assumption and for many fault assumption implementation is described within contents of the Chapter also.

5.1 Model and the mutation-based error correction

As it was stated previously in Chapter 3, the FORENSIC model is a control flow graph in form of a *hammock graph* $H=<N, E, n_0, n_e>$ [28], where every node n in N is parsed further into an *abstract syntax tree (AST)* $A=<N^*, E^*, n_0^*>$, that represents the operators, numeric values, variables and functions in a node *n*. Therefore possible correction mutations, that can be implemented within the model can be divided into three categories:

- Mutations in the nodes N^* of the of the AST of the hammock graph – implemented in FORENSIC [16] tool.
- Mutations in the edges E of the hammock graph – not implemented.
- Mutations in the edges E^* of the AST of the hammock graph – meaningless to implement.

Mutations in the edges E of the hammock graph allow to correct design logic, errors like missing brackets in logical conditions, as each graph node represents either a statement or part of the original statement. This is because statements of the original design are presented by smaller sub-statements in the model, and the logic in the logical conditions of the design is presented by several graph nodes, connected using edges, and mutations in edges can correct errors in the logic of the *C* design. This types of mutations are not implemented in the FORENSIC [16] tool.

Mutations in the nodes of the abstract syntax tree, N^*, can possibly correct errors like misuse of operators, values and variables. Mutations in the edges of the abstract syntax tree, E^*, can change the type of the graph node in N, and to break the model structure, and therefore they are not considered in the current work. Additionally, many of such errors should be caught by the *C* compiler.

So based on the model, we have 2 possible mutations for the Mutation-Based Error Correction algorithm, and only one of them is implemented. Let's take a closer look at some actual designs, and the error classes, that can be found in them.

5.2 Error classes

Error classes are classes of errors that could be discovered in the processed design. A valid error correction algorithm should be able to correct to as many of them as possible.

Error types that were found in experimental Siemens designs [6], that will be used for experiments with FORENSIC later, are described in the table below. It is given additionally if corresponding error class supported by FORENSIC [16] tool's mutation-based error correction or not.

Table 5.1. Error classes for the mutation-based error correction algorithm.

Error class	Operators/examples	Supported by mutation-based error correction?	Number of Designs in the Siemens designs
Missing code/code added	/* && (Cur_Vertical_Sep > MAXALTDIFF); */	NO, but can be corrected by alternative mutation	31
Operator replacement	ADD (+), SUB (-), MULT (*), DIV (/), MOD (%), EQ (==), NEQ (!=), GT (>), LT(<), GE (>=), LE (<=)	Operator mutations	27
Logic change	Brackets removed	NO, but can be corrected by alternative mutation	26
Number mutation/replace	Digit replacement, axb.c → ayb.c	Mutations in constant's number's digits	20
	+1/-1 added/removed to number	General mutations in constants	6
	Number *2, /2	General mutations in constants	2
	Digit added/removed abc.d → bc.d, xabc.d → abc.d	Mutations in constants Number digits	2
Integer to float mutation	abc → abc.x	Double-precision number rounding	5
Other mutations	Mutation in array size declaration, double replaced with float	NO	5
Function replaced by a constant	Inhibit_Biased_Climb() → Up_Separation	Variable mutations	4
+/-1 added/removed	r = i → r = i+1, f(a, b, c+1) → f(a, b, c), if(i) → if(i+1)	Integer variable mutations	2
Variable replaced	Positive_RA_Alt_Thresh[Al	NO	1

by number	t_Layer_Value] → Positive_RA_Alt_Thresh[0]		

Some of the error classes are not supported by the current version of the mutation-based error correction algorithm. Some are not supported because of the limitation of the algorithm, as it does not support corrections of the complex errors. Correcting others requires serious code modification, that is not possible to achieve using mutation for the correction, like when some code is missing in the processed design.

The error classes can be divided into two categories: errors, where the erroneous statement exists in the design, and errors, where the erroneous statement is missing from the design. All errors listed above except +/-1 belong to the first category, the +/-1 removed error belongs to the second. Model-based error localization accuracy differs for those two groups, as will be shown in the next Chapter, subsection 6.1.3.

5.3 Mutation-based error correction algorithm

The mutation-based error correction algorithm, implemented in FORENSIC, corrects errors by applying mutations only in the nodes of the control flow graph, but can be extended with mutations in edges.

Following are implemented mutations:

- Operator mutation – operators are mutated with ones with similar argument types.
- Integer variable mutation – integer variables are mutated with ones increased by 1 - “*I → I+1”*, or decreased by 1 - “*I → I-1”*. The mutation is supposed to suggest a correction for off by 1 errors in code. In addition, negating the variable - “*I → -I”* can be supported.
- Mutations of the variables of primitive types – variables of primitive type (not compound – struct, union), that are located at the right of the assignment operator (rvalue), are mutated with functions, that have no parameters and the same return type and value, “*I → F()*”.
- General mutations of constants. Those include adding one to the constant - “*C → C+1”*, subtracting one from the constant - “*C → C-1”*, setting the constant to zero - *“C → 0”*, negating the constant *“C → -C”*, multiplying and dividing the constant with 2 - *“C → C*2”*, *“C → C/2”*.
- Mutations in the digits of a numeric constant. When the constant is mutated, values “*0...9*” are assigned to every digit., The numeric constants can be floats or integers - “axb.c → ayb.c”, “ab.c → xab.c”.

- Floting point number rounding mutations. Floating point numbers (floats and doubles) are mutated with rounded ones, the values are rounded up and down.

Operator mutations in groups of operators with same argument types:

- Arithmetical operators: addition (+), subtraction (-), multiplication (*), division (/), modulus (%).
- Assignment operators: assignment (=), assignment with increment (+=), assignment with decrement (-=), assignment with multiplication (*=), assignment with division (/=), assignment with modulus(%=).
- Comparison operators: less than (<), greater than (>), equal (==), not equal (!=), less than or equal (<=), greater than or equal (>=).
- Logical operators and bit shift: logical and (&&), logical or (||), bit shift left (<<), bit shift right (>>), bitwise and (&), bitwise or (|), bitwise xor (^).
- Bit shift with assignments operators: bit shift left assignment (<<=), bit shift right assignment (>>=), bitwise and assignment (&=), bitwise or assignment (|=), bitwise exclusive or assignment (^=).
- Unary arithmetical operators: unary plus (+), unary minus (-).
- Increment and decrement operators: value after increment (++x), value before increment (x++), value after decrement (--x), value before decrement (x--).
- Unary logical operators: logical not (!), bitwise complement (~).

The mutation-based error correction algorithm implementations differ for different error assumptions. An assumption that there is only one possible error in the processed design is single fault assumption, assumption that there are several errors is multiple fault assumption.

If we have one fault assumption for error correction suggestion then mutations are applied one-by-one to the candidates for correction, and the mutated model is verified again using the *C* compiler. If, however, it is assumed that there may be multiple faults, then the mutations should be applied to several candidates simultaneously to a get valid correction. During verification, the simulation outputs in output observation points are compared with reference outputs. If the outputs match, then a valid candidate for repair is found, if they differ, then the process of error correction is continued while all candidates for correction are processed, refer to *Figure 3.4.1 a),b*):

Algorithm 3: Mutation-based error correction. Let the list of candidates for correction be C_s=<N*>, where N* is an ordered sequence of the control flow graph nodes N_t that have been activated with inputs S. The control flow

graph corresponds to a *hammock graph* $H=<N, E, n_0, n_e>$et $n \in N^*$. Let *O(n)* be arithmetical and logical operator(s) in *n* in *N**, *I(n)* be integer variable(s) in *n* in *N**, that are located at the right part of the assignment operator, *V(n)* be variable(s) of primitive types in *n* in *N**, that are located at the right part of the assignment operator, (include *I(n)*), *Ct(n)* be the number constants in *n* in *N**, *D(n)* – floating point value in *n* in *N**, *F()* - functions in *H*, and then

$\forall n_i \in C_s \subseteq N, i = 1...|N|$:

$\forall O(n_i)$: `// Operator mutations`

$O(n_i)$ is substituted with one with same argument types,
new model's Verification passes ? Error Is Corrected,
infinite loop during Verification ? break;
$O(n_i)$ is restored,
Error Is Corrected ? End.

$\forall I(n_i)$: `// Integer variable mutations`

$I(n_i)$ is substituted with $(I(n_i)+1, I(n_i)-1)$,
new model's Verification passes ? Error Is Corrected,
infinite loop during Verification ? break;
$I(n_i)$ is restored,
Error Is Corrected ? End.

$\forall V(n_i)$: `// Mutations of variables of primitive types`

$V(n_i)$ is substituted with *F()* if return type is same as $V(n_i)$
new model's Verification passes ? Error Is Corrected,
infinite loop during Verification ? break;
$V(n_i)$ is restored,
Error Is Corrected ? End.

$\forall Ct(n_i)$: `// General mutations of constants`

$Ct(n_i)$ is substituted with $(0, Ct(n_i)+1, Ct(n_i)-1, Ct(n_i)*2, Ct(n_i)/2)$,
new model's Verification passes ? Error Is Corrected,
infinite loop during Verification ? break;
$Ct(n_i)$ is restored,
Error Is Corrected ? End.

$\forall Ct(n_i)$: `// Mutations in the digits of numeric constants`

Every digit of $Ct(n_i)$ is substituted with values (0..9), and values (0..9) are added to the constant at the beginning,
new model's Verification passes ? Error Is Corrected,
infinite loop during Verification ? break;
$Ct(n_i)$ is restored,
Error Is Corrected ? End.
$\forall D(n_i)$: `// Rounding mutations of floating point numbers`
$D(n_i)$ is rounded up and down,
new model's Verification passes ? Error Is Corrected,
infinite loop during Verification ? break;
$D(n_i)$ is restored,
Error Is Corrected ? End.
No correction suggestion found.

The mutation-based error correction algorithm cannot suggest corrections for complex errors, but has some advantages. The error classes supported by the mutation-based error correction algorithm are clear and strongly defined, the algorithm is fast, reliable and simple. It is possible to suggest corrections for simple errors using the above mutation-based error correction algorithm and apply a more complicated error correction algorithm to the processed designs model again later on, in order to not spend time fixing simple errors using complicated approaches.

Experimental results are described in detail in the next chapter. According to the experimental results, the processing time when applying the mutation-based error correction algorithm to the Siemens designs [6] that have 180-570 lines of C code, takes from 1 minute up to 6 minutes.

5.4 Properties of the mutation-based error correction algorithm

The mutation-based error correction algorithm has several useful properties. In order to measure the error localization accuracy, in [12] the number of statements to reach the error is used as a metric. But every statement of the processed design can have different complexity, for example it can be an assignment, an assignment where there is an arithmetic expression on the right hand side, a function with different number of parameters, every parameter of the function can again be a function or arithmetic expression, it can be a boolean expression. Therefore many

operators can be located in a single statement. In this case the number of statements that need to be processed is not very accurate measure of the error localization accuracy.

On the other hand, the number of mutations that is applied to the parsed model should be practically the same for the same functionality, no matter how many operators in one statement of the design. This is because the nodes of the hammock graph used in the FORENSIC model have no more than 3 operands, and large expressions in statements are presented by several graph nodes. Therefore the number of mutations, applied to the processed model, corresponds to the error localization accuracy.

So using mutation-based error localization algorithm it is possible to measure how accurate error were located using model-based error localization algorithm. This is one of the novel approaches, presented in the current dissertation.

5.5 Correcting error by alternative mutation

Another property of the mutation-based error correction algorithm is that it can suggest corrections for errors in the design using mutations in the location, different from one, where actual error exists. As mutations are applied to all operators, numbers and double-precision numbers from candidates for correction nodes, alternative mutations can modify the design is such a way that it can corresponds to the specification.

For example, in the original tcas design from Siemens designs [6] (tcas/source/tcas.c) has the following code, at Figure 5.1.:

```
    need_downward_RA = Non_Crossing_Biased_Descend() && 
Own_Above_Threat();

    bool Non_Crossing_Biased_Descend()
    {
       int upward_preferred;
       int upward_crossing_situation;
       bool result;

       upward_preferred = Inhibit_Biased_Climb() > Down_Separation;
       if (upward_preferred)
       {
          result = Own_Below_Threat() && (Cur_Vertical_Sep >= MINSEP)
```

```
&& (Down_Separation >= ALIM());
      }
      else
      {
        result = !(Own_Above_Threat()) || ((Own_Above_Threat()) &&
(Up_Separation >= ALIM()));
      }
      return result;
   }

   bool Own_Above_Threat()
   {
      return (Other_Tracked_Alt < Own_Tracked_Alt);
   }
```

Figure 5.1. Original tcas design with no error from Siemens Designs[6].

Version 32 of the design with an injected error is the following (versions/v32/tcas.c), Figure 5.2:

```
   need_downward_RA = Non_Crossing_Biased_Descend();
   bool Non_Crossing_Biased_Descend()
   {
      int upward_preferred;
      int upward_crossing_situation;
      bool result;

      upward_preferred = Inhibit_Biased_Climb() > Down_Separation;
      if (upward_preferred)
      {
        result = Own_Below_Threat() && (Cur_Vertical_Sep >= MINSEP)
&& (Down_Separation >= ALIM());
        result = result && (Other_Tracked_Alt < Own_Tracked_Alt);
      }
      else
      {
        result = !(Own_Above_Threat()) || ((Own_Above_Threat()) &&
(Up_Separation >= ALIM()));
        result = result && (Other_Tracked_Alt <= Own_Tracked_Alt);
```

```
    }
    return result;
}

bool Own_Above_Threat()
{
    return (Other_Tracked_Alt < Own_Tracked_Alt);
}
```

Figure 5.2. Tcas v32 design with error from Siemens Designs[6].

In the currect example Own_Above_Threat function's functionality is transferred into Non_Crossing_Biased_Descend() function, and is erroneous only is if upward_preferred variable is *false*.

A suggestion for correction of the error were found using model-based error localization algorithm of the FORENSIC [16] tool using simple ranking and using mutation-based error correction of the FORENSIC tool [16] and is following –

to replace

```
bool Own_Above_Threat()
{
    return (Other_Tracked_Alt < Own_Tracked_Alt);
}
```

with

```
bool Own_Above_Threat()
{
    return ((Other_Tracked_Alt - 1) < Own_Tracked_Alt);
}
```

or, that is equivalent:

```
bool Own_Above_Threat()
{
    return (Other_Tracked_Alt <= Own_Tracked_Alt);
}
```

This correction suggestion does not match location of the existing actual error, as the latter is in the Non_Crossing_Biased_Descend function, but when the design if verified with the suggested correction no difference between the design and the specification is found.

Alternatively, when applying the Ochiai ranking for the error localization, then the candidates for correction data structure contains the nodes of the model in order, different from one, that were described above and the correctional mutation matches the actual location of the error, by replacing the line

```
result = result && (Other_Tracked_Alt <= Own_Tracked_Alt);
```

with

```
result = result && (Other_Tracked_Alt < Own_Tracked_Alt);
```

in the Non_Crossing_Biased_Descend() function.

Therefore, many errors can be corrected using mutations in several locations in the design, some of them may differ from actual error locations, and for different rankings locations of the correctional mutations can differ.

5.6 Inlfuence of the model format on the error localization and correction

Error localization and error correction algorithm's functionality depends on the representation of the parsed design – on the model. Different error localization and correction algorithms are more or less model dependent. Let us explain this dependency on an example.

Let us take a closer look at the *print_tokens* designs from the Siemens designs, more specifically at the *print_tokens/v7* design.

In *print_tokens/v7* design the following error exists: *“80” → “10”* in function *“numeric_case”* of the design. During the execution of the mutation-based error correction algorithm mutations in constants are applied, and *“10”* should be mutated back to *“80”*, that should correct the error. This type of mutation is included by the mutation-based error correction algorithm's error classes, mutation in constants digits (axb.c → ayb.c), refer to Section 5.2. But a correction suggestion for the error will not be found for this design.

In detail, the statement with the error in the design is following:

```
if(token_ind >= 10) break;
```

instead of the one in the specification:

```
if(token_ind >= 80) break;
```

The representation of the statement in the model of the design is following:

```
if (token_ind>9){
….
} ...
```

From the point of view of the *C* compiler there is no difference between *">=10"* and *">9"*, *as* those 2 statements are equivalent, and gcc front-end parser decides that *">9"* will be used for the design representation, possibly because of *">9"* is "simpler" than *">=10"*. Thus correction mutations are applied to the digit *"9"* during the execution of the mutation-based error correction algorithm. The particular mutations in constants used by the error correction algorithm are not be able to correct this error due to the representation, as there is no rule to make *"79"* from *"9"*, as the correct representation of model should be the following:

```
if (token_ind>79){
….
} ...
```

This simple integer error is not corrected using the expected "axb.c → ayb.c" mutation because of the model representation of the processed design, but will still be suggested a correction for by the alternative mutation "ab.c → xab.c", when digit "7" is added to the erroneous "9" and the required "79" is generated.

In general, expansion of the error classes, supported by the mutation-based error correction algorithm of the FORENSIC [16] tool will make process of error correction more accurate, more designs should be fixed, but process of correction slower, as all possible mutations should be applied to every node when correction of performed.

5.7 Chapter summary

The principles of the mutation-based error correction were described in the Chapter. In addition to error correction suggestion, mutation-based error correction algorithm is suitable for measuring the error localization accuracy. [45].

Mutation-based error correction algorithm uses information provided by the candidates for correction and applies various correction mutations to the nodes. It starts with the nodes with the highest rank and re-verifies the design according to the specification. If successful, then the correctional mutation had been found, an error is localized and corrected. If not, the process is continued until all the possible mutation are applied or the correction is found.

Mutation-based error correction algorithm is an error-matching approach, that were shown within contents of this Chapter using examples on the example Siemens Designs [6].

Mutation-Based error correction algorithm is model dependent, and it depends on how the design is presented in the model if error will be corrected or not. Some errors can not be corrected only because of specific representation of the design in the model, that were shown within contents of this Chapter using examples on the example Siemens Designs [6].

Number of mutations, applied during mutation-based algorithm's processing, is more accurate meter of the error localization accuracy than number of statements, required to be processed in order to reach the error location. The metric is more accurate in the sense that every statement can contain many different operators in original design, and, therefore, many different nodes in the corresponding model, and many different possible correctional mutations, and number of mutations, applied for error correction, should be more “realistic” meter.

Chapter 6. Experimental results

During the period 01.01.2010 – 01.01.2013 the DIAMOND FP7 [14] project was conducted by the Tallinn University of Technology and partners: University of Bremen, Graz University of Technology, Linköping University, IBM and Ericsson research groups. The goal of the project were to develop new algorithms and methods for error localization and error correction in the C designs, to implement them in a usable tool and to publish the experimental results that can be used further for developing industrial error localization and error correction tools. The name of the developed tool within contents of the project is FORENSIC tool [16]. Experimental results, obtained within contents of the project using the tool, are described in the current Chapter.

Three types of experiments are described in the current Chapter: model-based error localization, dynamic slicing and mutation-based error correction experiments all using the Siemens designs [6] as input designs, experiments are compared with similar ones, published in [12, 60, 1].

Mutation-based error correction experiments are compared with the corresponding experiments made by Debroy and Wong [12], and model-based error localization experiments are compared with the corresponding results by Abreu, Zoeteweij and van Gemund [1]. The details of the experimental data for model-based error localization of the FORENSIC tool [16] are in *Appendix 2.* The details of the dynamic slicing experiments are given in *Appendix 1*.

Most of the experimental results are published in [45].

6.1 Experiments with Siemens Designs

The Siemens benchmarks/designs are widely used for the evaluating

various approaches to error localization and error correction. For example in [12, 1, 48] the same benchmarks used for the experiments. Researchers at Siemens sought to study fault-detecting effectiveness of coverage criteria. Therefore, they created faulty versions of seven base programs by manually seeding those programs with faults, usually by modifying a single line of code in the program. In a few cases they modified between two and five lines of the code. Their goal was to introduce faults that were as realistic as possible, based on their experience with real programs. Ten people performed fault seeding, working "mostly without knowledge of each other's work" [48].

Siemens benchmarks/designs are variants of programs, versions, and test cases created originally by researchers at Siemens Corporate Research and provided by Mary Jean Harrold and Gregg Rothermel [6].

Designs are programs written in *C* with different functionalities as described below:

- tcas – logical/arithmetic calculations.
- print_tokens, print_tokens2 – text processing.
- replace – text processing.
- schedule, schedule2 – usage of C structs and unions.
- tot_info – arithmetic computations.

Descriptions of the designs are in Table 6.1.

Table 6.1. Overview of the Siemens benchmarks.

Design	Number of lines of code	Number of inputs	Number of designs with errors
print_tokens	569	4130	7
print_tokens2	515	4115	10
replace	558	5543	31
schedule	419	2650	9
schedule2	312	2710	10
tcas	186	1605	41
tot_info	413	1052	22

In order to get numeric data about model-based error localization and mutation-based error correction algorithm's executions, with and without dynamic slicing usage, inputs C designs for the FORENSIC tool [16] are required. The Siemens benchmarks contain set of C designs, that are used as inputs.

6.1.1 Organization of the test suites in Siemens benchmarks

For each of the base programs, researchers at Siemens created a test pool containing useful inputs for the design. To populate these input sets, they first created an initial set of black-box test cases “according to good testing practices, based on the tester’s understanding of the program’s functionality and knowledge of special values and boundary points that are easily observable in the code”, using category partition method and the Siemens Test Specification Language tool [3]. They then augmented this set with manually-created white-box test cases to ensure that each executable statement, edge, and definition-use pair in the base program or its control flow graph was exercised by at least 30 test cases. To obtain meaningful results with the seeded versions of the programs, researchers retained only faults that were “neither too easy nor too hard to detect”, which they defined as being detectable by at least three and at most 350 test cases in the input associated with each program [48].

As a result different versions of test suites were created for the Siemens designs by Siemens Corporate Research, and those are used as inputs for the FORENSIC tool. Inputs are located in the „testplans.alt” folder of each

design, scripts folder contains scripts that execute designs with corresponding arguments for test cases.

In the „testplans.alt" directory of each design there are the following:

- "testplans-cov": tests that were generated to achieve branch coverage in somewhat minimal fashion, continually adding tests to the suite as long as they add coverage.
- "testplans-rand": tests that were generated randomly, 1000 of them, with sizes ranging randomly from A/2 to A*2, where A is average size of branch coverage suites found in -cov directories.
- "testplans-rand-covsize": tests that were generated randomly, 1000 of them, with sizes equivalent to sizes of their correspondingly named tests in "testplans-cov".
- "testplans-bigcov": tests that are also generated for achieving coverage, but not in a minimal manner – a test that hits a particular edge was picked at random. 1000 testswere generated.
- "testplans-bigrand": tests that were selected randomly, to have same size as their corresponding "bigcov" suites.
- "universe" – total amount of tests for the program[6].

For the experiments, described in this chapter, I use "universe" tests, or part of the "universe" tests, when first 1000 or 2500 inputs are used, and inputs are not selected if they are reduced, but taken randomly. This makes sure that probability to detect error is kept the same when number of inputs is reduced, error correction, however, may fail because of insufficient inputs for applied mutation's check.

6.1.2 Error localization experiments

In [1] a number of rankings for the model-based error localization are proposed and the corresponding experiments are performed with Siemens designs. In the current dissertation similar experiments are performed, using the rankings that are defined in Chapter 4 and it is possible to compare error localization accuracy in [1] with the approach used in FORENSIC tool [16].

The experimental results differ, and difference is defined by the different metrics used: in [1] the metric of error localization accuracy is the percentage of statements to be inspected to reach error location, in the current dissertation we propose to use as a metric the percentage of mutations required to correct the error with the mutation-based error correction algorithm, refer to Subsection 5.4. Designs, where the error correction was not found using the mutation-based error correction procedure discarded from ranking

comparison experiments of the FORENSIC tool [16]. Simple and Consecutive rankings, introduced in Chapter 4, is used for FORENSIC tool's experiments too. Siemens designs [6] were used as input designs.

Average error localization accuracy, over all designs in the Siemens Benchmarks set [8] for various rankings is in [1] is in *Table 6.2*.

Table 6.2. Average error localization accuracy using different similarity coefficients in [1] – percentage of statements to be inspected to reach error.

	Ample	Jaccard	Ochiai	Tarantula
print tokens	2	7.3	1	10.5
print tokens2	16.4	17.5	13.9	20
replace	12.7	11.6	7.6	12.2
schedule	11.3	2.9	1.6	3
schedule2	34.7	31.1	25.1	31.3
tcas	9.8	8.8	7.9	8.8
tot info	13.2	9.6	7.1	11
Mean	14.3	12.69	9.17	13.83

The results of error localization accuracy, obtained with FORENSIC are in *Table 6.3*. Details of the results of performing model-based error localization in FORENSIC are given in Appendix 2.

Table 6.3. Average error localization accuracy using various rankings for error classes supported by the mutation based error correction algorithm in FORENSIC – percentage of mutations to be applied to correct the error in the design.

	Ample	Consecutive	Jaccard	Ochiai	Simple	Tarantula	Number of corrected designs
print tokens	12.34	8.5	9.05	9.05	20.22	9.05	2
print tokens2	4.29	1.34	10.88	2.65	25.61	12.9	7
replace	5.93	3.0	5.82	4.19	12.23	5.97	16
schedule	5.15	2.55	6.88	2.63	17.53	7.09	3
schedule2	50.04	37.49	45.22	41.92	15.33	45.22	4
tcas	8.4	5.99	9.51	6.72	8.04	9.86	31
tot info	29.16	8.03	19.67	9.81	16.62	21.57	18
Mean	16.47	9.56	15.29	11	16.51	15.95	
Weighted mean	14.2	6.94	12.81	8.2	13.3	13.58	
Corrected Standard deviation	16.14	12.51	13.13	13.69	5.81	12.95	

Graphical representations of corresponding ranking accuracies for *Table 6.2* and *Table 6.3* are given at *Figures 6.1* and *Figure 6.2* correspondingly.

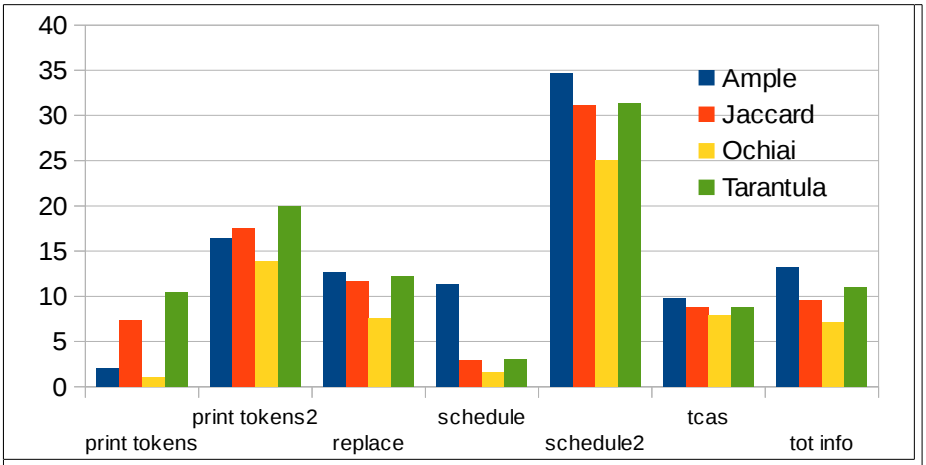

Figure 6.1. Graph representation of the average error localization accuracy using different similarity coefficients in [1] – percentage of the statements to be inspected to reach error.

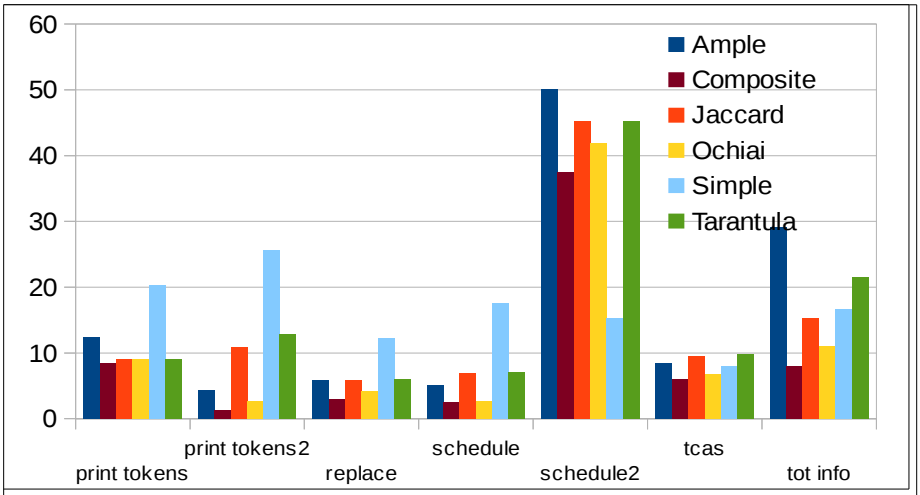

Figure 6.2. Graph representation of the average error localization accuracy using various rankings for error classes supported by the mutation based error correction algorithm in the FORENSIC tool – percentage of mutations to be applied to correct the error.

Corrected standard deviation *s* of mutations, required to correct the error, is calculated for different rankings for experiments with FORENSIC to estimate the error correction stability:

$$s=\sqrt{\frac{1}{n-1}\cdot\sum_{i=1}^{n}\left(x_i-\bar{x}\right)^2} \quad (6.1.8)$$

Where corrected standard deviation *s* for design is calculated using x_i – number of mutations required to correct error in design using particular ranking, $\bar{x}$ is mean over number of mutations, required to correct error in design, for all rankings, *n* is the number of rankings.

Corrected standard deviations for error localization experiments in *Table 6.4* are following:

Table 6.4. Corrected standard deviation of mutations required to correct error using various rankings in the FORENSIC tool.

Ranking	Standard Deviation - *s*
Ample	16.14
Consecutive	12.51
Jaccard	13.13
Ochiai	13.69
Simple	5.81
Tarantula	12.95

6.1.3 Interpretation of the error localization experiments

Following conclusions can be made from experimental data:

Simple ranking for the model-based error localization is more stable than other rankings, as it has lowest standard deviation of results equal to 5.*61*, that is 2.23 times less than second minor standard deviation, obtained using tarantula ranking for the error localization for the Siemens Benchmarks[6].

As it is said in [1], Ochiai ranking decreases the percentage of blocks of code to be inspected by *5%* comparing with worst case – the Ample tool

ranking, and by *3.5%* comparing with the next to Ochiai - Jaccard coefficient ranking, refer to Table 6.6, and using the FORENSIC [16] tool, the Ochiai ranking decreases the number of required mutations compared to the worst case Ample ranking by 10%, that is much better.

Consecutive ranking that is simple ranking plus some secondary ranking, for example ochiai, results the best results, it has 1.25% better error localization accuracy than next the Ochiai ranking. Its advantages – stability and error accuracy of both simple ranking and secondary Ochiai ranking.

It also shows the best results for the case when a part of the code is missing. The corresponding supported error class, missing increment and decrement by one exists only in 3 mutations of the Siemens designs[6]. For more description refer to Section 5.2. The results of applying different rankings on the mutated designs is in *Table 6.5*.

Table 6.5. Error Localization results when erroneous statement is missing code in erroneous design, number of mutations, required to apply, to correct error.

Design	Ample	Consecutive	Jaccard	Ochiai	Simple	Tarantula	Error class
replace/v1	7	7	7	7	795	7	-1 missing
replace/v23	1496	152	2187	1570	328	2214	+1 missing
replace/v26	66	66	66	66	3955	66	+1 missing
Mean	523	75	753.33	547.67	1692.67	762.33	
% of mutations to inspect	3.91	0.56	5.63	4.1	12.66	5.7	13373

6.1.4 Dynamic slicing experiments

Processed designs can be very different, and some of them can have activated nodes, that do not have any influence on the simulation output. It is possible to discard such nodes by applying the dynamic slicing algorithm. The corresponding experiments were performed with FORENSIC [16] tool. The summary of experimental results of applying the dynamic slicing algorithm in FORENSIC [16] tool is in *Table 6.6* and full details can be found in *Appendix 1*.

Table 6.6. Number of mutations, required to correct errors in the Siemens designs with model-based error localization with and without dynamic slicing use.

Design	Mean number of mutations, required to correct error if for error localization		Change, $\left\|\frac{A-B}{B}\cdot 100\,\%\right\|$
	Dynamic slicing is used (A)	Dynamic slicing is not used (B)	
print_tokens	1954.5	2106.0	7.75
print_tokens2	2141.14	2254.29	5.02
replace	1635.88	1683.06	2.8
schedule	564.0	715.33	21.16
schedule2	648.25	713.25	9.11
tcas	304.0	553.71	45.1
tot_info	1338.5	1363.94	1.87

6.1.5 Interpretation of the dynamic slicing experiments

The effect of the dynamic slicing algorithm on error localization accuracy, or on the number of correctional mutations required for a correction depends on the internal structure of the design. If the design has a number of initializations in the beginning of the code, then dynamic slicing algorithm discards initializations that are not initialized with particular simulation with input *s*.

According to the experiments, the dynamic slicing algorithm has a high influence on the *tcas, schedule, schedule2 and print_tokens* design. For tcas design the *dynamic slicing* algorithm will reduce the number of required mutations for correction by *45,1%* on average, for schedule – by *21,16%*, for schedule2 by *9,11%*, and for print_tokens by *7.19%*, but less for *replace, tot_info* and *print_tokens2* designs.

In the experimental evaluation of using dynamic slices for fault location [60] dynamic slicing experiments are presented using the same designs – Siemens designs [6], and it is possible to compare the results. The experimental data, obtained using FORENSIC [16] tool corresponds to the relevant slicing algorithm from [60], and the number of processed designs from each design group is presented for comparison in Table 6.7.

Table 6.7. Comparison of percentage of number of statements, required to reach error location. Using Relevant Slicing in [60] with FORENSIC dynamic slicing experiments, when number of mutations, need.

Design	Number of processed designs using FORENSIC tool	Change, if dynamic slicing in FORENSIC tool is used, % (A)	Number of processed designs in [60]	Change, if Relevant Slicing in [60] is used, % (B)	Difference of error localization accuracy change, B-A, %
print_tokens	2	7.75	5	32	24.25
print_tokens2	7	5.02	8	49	43.98
replace	16	2.8	19	42	39.2
schedule	3	21.16	6	47	25.84
schedule2	4	9.11	3	53	43.89
tcas	31	45.1	-	-	
tot_info	18	1.87	-	-	

From *Table 6.7.* it can be seen that the relevant slicing algorithm [60] allows to increase the error localization accuracy by *24,25%* for print_tokens design, by *43.98%* for *print_tokens2* design and by *43,89%* for *schedule2* design, comparing with FORENSIC tool [16] dynamic slicing implemnetation. It is obivious that in [60] relevant slicing algorithm is more accurate.

While the relevant slicing algorithm performs better than our implementation of the dynamic slicing, it is relatively straight forward to extend the FORENSIC error localization back-end with the appropriate algorithm. On the other hand, we wanted to demonstrate the effect of dynamic slicing on the error localization accuracy using simpler than in [60] implementation of dynamic slicing.

6.1.6 Error correction experiments

During error correction experiments following assumptions were made:

- We have one error assumption.
- Dynamic slicing algorithm is used for the model-based error localization.
- The simple ranking is used for model-based error localization.

Model-based error localization followed by model-based error correction involving simulation-based verification was performed for each mutated design in the Siemens Designs[6]. The summary of the experiments is given in *Table 6.8.*

Table 6.8. Error correction experiments.

Design	Percentage of Errors Corrected	Mean number of mutations required to correct error, Simple ranking	Mean Processing time, s*
print_tokens	2/7 = 28.57%	1954.5	330
print_tokens2	7/10 = 70.00%	2124.14	502
replace	16/32 = 50.00%	1635.88	342
schedule	3/9 = 33.33%	564	130
schedule2	4/10 = 40.00%	648.25	184
tcas	31/41 = 75.61%	304	50
tot_info	18/32 = 56.25%	1338.5	312

** Processing time depends on the number of the inputs. Given time is for 1000 inputs for each design.*

6.1.7 Interpretation of the error correction experiments

From error correction experiments following conclusions can be made:

- Siemens Designs[6] are good and suitable for experiments.
- Only if error belongs to implemented error classes or not states will error be error corrected or not. Supported error classes are defied in the Chapter 5.
- From detailed experimental results, attached to *Appendixes 1* and *2* can be concluded, that Model-Based Error Localization and Mutation-Based Error Correction algorithms is model-dependent, some of errors were not corrected

during experiments because of design's model representation, some were corrected involuntarily.

6.1.8 Comparison of the error correction experiments

In [12] mutation-based error correction experiments on the same Siemens designs is presented, and it is possible to compare the results.

The comparison of the experimental results from [12] and FORENSIC [16] tool's is in *Table 6.9*:

Table 6.9. Comparison of the number of errors corrected using an approach in [12] and FORENSIC's mutation-based error correction.

Design	Number of errors, corrected in [12] (A)	Number of errors, corrected using *FORENSIC* (B)	Total designs (C)	$\frac{B-A}{C}\cdot 100\,\%$, percentage of additionally corrected errors
print_tokens	0	2	7	28.57
print_tokens2	0	7	10	70.00
replace	3	16	32	40.63
schedule	0	3	9	33.33
schedule2	1	4	10	30.00
tcas	9	31	41	53.66
tot_info	8	18	23	43.48
Mean/Sum	21	81	132	45.45

From *Table 6.9.* it can be seen that mutation-based error correction algorithm of the FORENSIC tool [16] can suggest corrections for significantly more designs than the approach presented in [12], *70%* additionally corrected errors for *print_tokens2* design, *54%* additionally corrected for *tcas* design. It seems that [12] error correction algorithm does not support the mutations in number of error classes, that are supported by FORENSIC [16] tool. Additional mutations (error classes) can be added to the mutation-based error correction algorithm of FORENSIC for even better results.

In [12] the data exists about the number of mutations, required to correct the errors. Unfortunately, because of different error classes, supported by Debroy and Wong in [12], it is not straight forward to compare the number of mutations.

6.2 Chapter summary

The chapter presents the error localization and correction experiments with FORENSIC [16] tool.

Three types of experiments were described in the current Chapter: model-based error localization, dynamic slicing and mutation-based error correction experiments all using the Siemens designs [6].

For the model-based error localization experiments simple ranking for the model-based error localization is more stable than other rankings, as it has lowest standard deviation of results equal to 5.*61*, that is 2.23 times less than second minor standard deviation, obtained using tarantula ranking.

As it is said in [1], Ochiai ranking decreases the percentage of blocks of code to be inspected by *5%* comparing with worst case – the Ample tool ranking, and by *3.5%* comparing with the next to Ochiai - Jaccard coefficient ranking, refer to Table 6.6, and using the FORENSIC [16] tool, the Ochiai ranking decreases the number of required mutations compared to the worst case Ample ranking by 10%, that is much better.

For whole comparison data refer to *Table 6.2* and *Table 6.3*.

Consecutive ranking that is simple ranking plus some secondary ranking, for example ochiai, results the best results, it has 1.25% better error localization accuracy than next the Ochiai ranking. Its advantages – stability and error accuracy of both simple ranking and secondary Ochiai ranking.

According to the experiments, the dynamic slicing algorithm of the FORENSIC tool [16] has a high influence on the *tcas, schedule, schedule2 and print_tokens* design. For tcas design the *dynamic slicing* algorithm will reduce the number of required mutations for correction by *45,1%* on average, for schedule – by *21,16%*, for schedule2 by *9,11%*, and for print_tokens by *7.19%*, but less for *replace, tot_info* and *print_tokens2* designs.

However, it can be seen that the relevant slicing algorithm [60] allows to increase the error localization accuracy by *24,81%* for print_tokens design, by *43.98%* for *print_tokens2* design and by *43,89%* for *schedule2* design, comparing with FORENSIC tool [16] dynamic slicing implemnetation. It is obivious that in [60] relevant slicing algorithm is more accurate.

For whole comparison data refer to *Table 6.6* and *Table 6.7*.

Mutation-based error correction algorithm of the FORENSIC tool [16]

can suggest corrections for significantly more designs than the approach presented in [12], *70%* additionally corrected errors for *print_tokens2* design, *54%* additionally corrected for *tcas* design. For whole comparison data refer to *Table 6.9.* It seems that [12] error correction algorithm does not support the mutations in number of error classes, that are supported by FORENSIC [16] tool. Additional mutations (error classes) can be added to the mutation-based error correction algorithm of FORENSIC for even better results.

All experimental data is published in [45, 41].

Chapter 7. Conclusions and future work

Current chapter includes short description of the tool, developed with DIAMOND FP7 project, description of the novel algorithms and approaches, used with the tool, and summary of the experimental results, obtained with the tool. In the end of the chapter future extensions of the tool exists.

7.1 Description of the tool

The essence of the work described in this dissertation is to enhance the state of the art of the development tools targeted to design software that is further synthesized into hardware.

FORENSIC is an implementation of a such tool, experimental results and novel approaches, suggested in the dissertation, can be used for future academic research and can be applied in the software development industry.

It consists of the *GIMPLE f*ront-end (implemented at the University of Bremen), that parses a processed C design into the model representation (implemented at the Graz University of Technology), and number of back-ends, exactly a simulation-based back-end, that consists of model-based error localization with dynamic slicing and mutation-based error correction algorithms, (implemented at the Tallinn University of Technology by the author), and described in detail in the current dissertation. In addition tool consists of a symbolic and concolic back-end (implemented at the Graz University of Technology by Robert Könighofer and colleagues from Institute for Applied Information Processing and Communications), and wolfram back-end (implemented at the University of Bremen by Alexander Finder and colleagues from group of Computer Architecture).

The error localization and error correction algorithms, described within

contents od the dissertation, are applicable for error localization and error correction of designs, written on arbitrary programming languages, including C/C++/SystemC and Java.

7.2 Summary of the contributions and novel approaches

Novel approaches and contributions, presented in this dissertation are stated below:

Introduced two novel rankings for model-based error localization – Consecutive and Simple rankings.

Usage of the compiler functionality for the model simulation task, the specification animation and the implementation of the dynamic slicing algorithm.

Usage of the number of processed mutations instead of a number of statements required to reach the error as a measure of the error localization accuracy.

Implementation of the model-based error localization, mutation-based error correction and dynamic slicing algorithms in the corresponding tool.

Definition of the specification format for simulation-based verification.

Finally, based on the development and evaluation performed during writing the dissertation, can be claimed that the FORENSIC tool can be utilized in a useful way in the process of software development. The experimental results obtained with the tool make it possible to implement similar tools with a more user-friendly interface, based on the same verification principles.

7.3 Summary of the experimental results

For the model-based error localization experiments *simple* ranking for the model-based error localization is more stable than other rankings, as it has lowest standard deviation of results equal to 5.*61*, that is 2.23 times less than second minor standard deviation, obtained using tarantula ranking.

Ochiai ranking decreases the number of required mutations compared to the worst case *Ample* ranking by 10%, *Consecutive* ranking that is simple ranking plus some secondary ranking, for example ochiai, results the best results, it has 1.25% better error localization accuracy than next the *Ochiai* ranking. Its advantages – stability and error accuracy of both simple ranking and secondary Ochiai ranking.

For whole comparison data refer to *Table 6.2* and *Table 6.3*.

According to the experiments, the dynamic slicing algorithm of the

FORENSIC tool [16] has a high influence on the *tcas, schedule, schedule2 and print_tokens* design. For tcas design the *dynamic slicing* algorithm will reduce the number of required mutations for correction by *45,1%* on average, for schedule – by *21,16%*, for schedule2 by *9,11%*, and for print_tokens by *7.19%*, but less for *replace, tot_info* and *print_tokens2* designs.

For whole comparison data refer to *Table 6.6* and *Table 6.7.*

Mutation-based error correction algorithm of the FORENSIC tool [16] can suggest corrections for significantly more designs than the approach presented in [12], *70%* additionally corrected errors for *print_tokens2* design, *54%* additionally corrected for *tcas* design. For whole comparison data refer to *Table 6.9.* It seems that [12] error correction algorithm does not support the mutations in number of error classes, that are supported by FORENSIC [16] tool. Additional mutations (error classes) can be added to the mutation-based error correction algorithm of FORENSIC for even better results.

7.4 Future Extensions

In the future, the FORENSIC [16] tool can be extended with *C++/SystemC* support and the algorithms presented in the dissertation can be carried over for *C++/SystemC* error localization and error correction.

It is possible to implement an input generator for the tool while the reference outputs can be achieved by simulating the specification with the generated inputs.

The usability of the tool is further enhanced by extensive documentation available at the web page of the project [14].

References

1. Abreu Rui, Zoeteweij Peter and van Gemund Arjan J.C. 2006. An Evaluation of Similarity Coefficients for Software Fault Localization. Dependable Computing, 2006. PRDC '06. 12th Pacific Rim International Symposium on, 39-46.

2. Aviž ienis A., Laprie J.-C., Randell B., and Landwehr, C. E.. 2004. Basic concepts and taxonomy of depend able and secure computing. IEEE Trans. Dependable Sec. Computing, 11–33

3. Balcer M., Hasling W., and Ostrand T.. 1989. Automatic generation of test scripts from formal test specifications. In Proc. Of the 3rd Symp. on Softw. Testing, Analysis, and Verification, 210–218.

4. Borba Paulo, Goguen Joseph A. 1994. An Operational Semantics for FOOPS. Oxford University, Computing Laboratory, Programming Research Group

5. Brüning J., Gogolla M., and Forbrig P. 2010. Modeling and Formally Checking Workflow Properties Using UML and OCL. in Proc. BIR, 2010, 130–145

6. Bug Aristotle Analysis System -- Siemens Programs, HR Variants [online]. WWW:<http://pleuma.cc.gatech.edu/aristotle/Tools/subjects/> (01.01.2012)

7. Cleve H. and Zeller A. 2005. Locating causes of program failures.

Proc. of Int. Conf. on Software Engineering, 342-351, 2005.

8. Chen M. Y., Kiciman E., Fratkin E., Fox A. and Brewer E.. 2002. Pinpoint: Problem determination in large, dynamic internet services. International Conference on Dependable Systems and Networks, 595–604

9. Cohen B., Venkataramanan S., A Kumari. 2004. Using PSL/Sugar for Formal and Dynamic Verification: Guide to Property Specification Language for Assertion-Based Verification, 2nd Edition. VhdlCohen Publishing.

10. Dallmeier V., Lindig C., and Zeller A.. 2005. Lightweight defect localization for Java. In A. P. Black, editor, ECOOP 2005 : 19th European Conference, Glasgow, UK, July 25–29 and Proceedings, volume 3568 of LNCS., 528–550

11. Davis, B. 1982. The Economic of Automated Testing. McGraw-Hill, London, UK

12. Debroy Vidroha, and Wong W. Eric. 2010. Using Mutation to Automatically Suggest Fixes for Faulty Programs. Software Testing, Verification and Validation (ICST), 2010 Third International Conference on, 65–74

13. DeMillo R. A., Lipton R. J., and Sayward F. G. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. IEEE Computer, vol. 11, Issue 4, 34–41, 1978.

14. DIAMOND - Diagnosis, Error Modelling and Correction for Reliable Systems Design [online]. *WWW:*<http://fp7-diamond.eu/> (01.01.2012)

15. Douglas L. Perry, Harry D. Foster. 2005. Applied Formal Verification. The McGraw-Hill Companies, Inc.

16. *FORENSIC* tool web page [online]. *WWW:*< *http://www.informatik.uni-bremen.de/agra/eng/forensic.php*> (01.01.2012)

17. GIMPLE plug-in for GCC compiler [online]. *WWW:*< *http://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html* > (01.01.2012)

18. Goguen Joseph, Meseguer Jose. 1982. Rapid prototyping: in the OBJ executable specification language. Proceedings of the workshop on Rapid prototyping. 75-84

19. Guttag J. V., Horning J. J., Garland S. J., Jones K. D., Modet A., Wing J. M. 1992. Automatic Translation of VDM Specifications into Standard ML Programs. The Computer Journal (1992) 35 (6), 623-624.

20. Guttag J. V., Horning J. J., Garland S. J., Jones K. D., Modet A., Wing J. M.. 1993. Larch: Languages and tools for formal specification. Texts and monographs in computer science.

21. Jadish S. Gangolly. Lecture notes on Design & Analysis of Accounting Information Systems [online]. University at Albany, New York, 2000. WWW: <http://www.albany.edu/acc/courses/acc681.fall00/681book/681book.html> (01.01.2012)

22. Jain A. K. and Dubes R. C. 1988. Algorithms for clustering data. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 528–550

23. Jha N. K. and Gupta S. 2003. Testing of Digital Systems. Cambridge University Press

24. Jia Xiaoping. 1995. An approach to animating Z specifications. Computer Software and Applications Conference, Proceedings., Nineteenth Annual International, 108-113

25. Hayes I.J. 1992. Specification Case Studies. Second edition. London: Prentice-Hall.

26. Hofer Birgit, Wotawa Franz. 2015. Fault Localization in the Light of Faulty User Input. Software Quality, Reliability and Security (QRS), 2015 IEEE International Conference on, 282-291

27. Hofferek Georg, Könighofer Robert, Schmidt Jörn-Marc, Fey Görschwin, Finder Alexander, Sülflow Andre, Ingelsson Urban, Repinski Urmas, Raik Jaan, Scholefield Stephen. 2012. FoREnSiC - An Automatic Debugging Environment for C Programs. Haifa Verification Conference

(HVC'2012), 260-265.

28. Kas'janov, V. N. 1975. Distinguishing Hammocks in a Directed Graph. Soviet Math. Doklady, vol. 16, no. 5, 448-450.

29. Kleer J. de and Williams B. C. 1989. Diagnosis with behavioral modes. Proceedings of IJCAI-89, 324-1330

30. Korel Bogdan, Rilling Jurgen. 1997. Application of Dynamic Slicing in Program Debugging. AADEBUG - Proceedings of the Third International Workshop on Automatic Debugging, Linköping, Sweden, 43-58

31. Könighofer Robert and Bloem Roderick. 2011. Automated error localization and correction for imperative programs. Formal Methods in Computer-Aided Design (FMCAD), 2011, 91 – 100

32. Lach, J. ; Bingham, S. ; Elks, C. ; Lenhart, T. ; Nguyen, T. ; Salaun, P. ;. 2006. Accessible formal verification for safety-critical hardware design. Reliability and Maintainability Symposium, 2006, RAMS '06. Annual, 29 – 32

33. Lam William K. 2005. Hardware Design Verification. Simulation and Formal Method-Based Approaches. Prentice Hall. Profession Technical Reference.

34. Larsen Kim G., Pettersson Paul, Yi Wang. 1997. Uppaal in a nutshell. International Journal on Software Tools for Technology Transfer, Volume 1, Issue 1-2, 134-152

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

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

37. Marciniak J. J. 2001. Process Models in Software Engineering. Encyclopedia of Software Engineering, 2nd Edition.

38. Myers G. J. 2004. The Art of Software Testing, 2nd Edition, John Wiley & Sons

39. Passos Lúcio S.; Abreu Rui; Rossetti Rosaldo J. F. 2016. Empirical Evaluation of Similarity Coefficients for Multiagent Fault Localization. IEEE Transactions on Systems, Man, and Cybernetics: Systems, 1-16.

40. Pierce P. 1996. Software verification and validation. Northcon/96, 265-268

41. Quek, C. and Leitch, R.R. 1992. Using energy constraints to generate fault candidates in model based diagnosis. Intelligent Systems Engineering, First International Conference on (Conf. Publ. No. 360), 159 - 164

42. Reiter R. 1987. A theory of diagnosis from first principles. Artificial Intelligence, 32, 57-95.

43. Repinski Urmas. 2012. Model-based Verification with Error Localization and Error Correction for C Designs. Программные продукты и системы (international journal Software and Systems), № 100, 2012, 221-229.

44. Repinski, U., Hantson, H., Jenihhin, M., Raik, J., Ubar, R., Di Guglielmo, G., Pravadelli, G., Fummi, F. 2012. Combining dynamic slicing and mutation operators for ESL correction. Test Symposium (ETS), 2012 17th IEEE European, 1-6, 2012.

45. Repinski Urmas, Raik Jaan. 2012. Comparison of Model-Based Error Localization Algorithms for C Designs. Proc. of 10th East-West Design & Test Symposium, Kharkov, Ukraine, September 14-17, IEEE Computer Society Press, 1-4

46. Reps T., Ball T., Das M., and Larus J. 1997. The use of program profiling for software maintenance with applications to the year 2000 problem. In M. Jazayeri and H. Schauer, editors, Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97), volume 1301 of LNCS., 432 – 449

47. Rogin F, and Drechsler R. 2010. Debugging at the electronic system

level. Springer Verlag, 2010.

48. Rothermel, G. ; Harrold, M.J. ; Ostrin, J. ; Hong, C. 1988. An empirical study of the effects of minimization on the fault detection capabilities of test suites. Software Maintenance, 1998. Proceedings. International Conference on, 34–43.

49. Selic Bran. 1998. Using UML for modeling complex real-time systems. Lecture Notes in Computer Science Volume 1474, 250-260

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

51. Stephen, Andriole J. 1986. Software Validation, Verification, Testing, and Documentation. Princeton, NJ: Petrocelli Books.

52. Uppaal tool web page [online]. *WWW:*<*http://www.uppaal.org/* >

53. Weiser Mark. 1984. Program Slicing. IEEE Transactions on Software Engineering, vol. SE-10, no. 4, 352-357.

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

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

56. Woodcock Jim, Davies Jim. 1996. Using Z: specification, refinement, and proof. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1996

57. Working Draft of the CGMP Final Rule – Regulation, 1995, 2-3

58. You Yi-Sian; Huang Chin-Yu; Peng Kuan-Li; Hsu Chao-Jung. 2013. Evaluation and Analysis of Spectrum-Based Fault Localization with Modified Similarity Coefficients for Software Debugging. Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual, 180 – 189

59. Zhang Fubo and D'Hollander Erik H. 2004. Using Hammock Graphs to Structure Programs. IEEE Trans. Softw. Eng. 30, 4 (April 2004), 231-245

60. Zhang Xiangyu, He Haifeng, Gupta Neelam, Gupta Rajiv. 2005. Experimental evaluation of using dynamic slices for fault location. ADEBUG - Proceedings of the Third International Workshop on Automatic Debugging, Linköping, Sweden, 33-42

61. Zhu Yunshan. 2005. Applying formal techniques in simulation-based verification. ASIC, 2005. ASICON 2005. 6th International Conference On, 1036 – 1041

Appendix 1. Dynamic slicing experiments using FORENSIC

Simple ranking is used for the dynamic slicing experiments, but any other ranking can be used.

tcas design

Erroneous design	Error detected times	Number of mutations required to correct error, simple ranking.		Error Class
		Localization with dynamic slicing	Localization without dynamic slicing	
tcas/v1	132	727	1011	ROR
tcas/v2	69	114	384	Integer mutation
tcas/v3	23	707	990	LCR
tcas/v4	26	523	808	LCR
tcas/v5	10	NF		Missing Code
tcas/v6	12	87	372	ROR
tcas/v7	36	45	111	Integer mutation
tcas/v8	1	44	246	Integer mutation
tcas/v9	9	212	497	ROR
tcas/v10	14	87	372	ROR
tcas/v11	14	87	372	ROR
tcas/v12	59	438	723	LCR
tcas/v13	4	465	750	Integer mutation
tcas/v14	50	170	440	Integer mutation
tcas/v15	10	NF		Integer Mutation 300 → 350 + Logic change

Erroneous design	Error detected times	Number of mutations required to correct error, simple ranking.		Error Class
tcas/v16	70	37	37	Integer mutation
tcas/v17	35	39	105	Integer mutation
tcas/v18	29	45	179	Integer mutation
tcas/v19	19	48	250	Integer mutation
tcas/v20	18	391	676	ROR
tcas/v21	16	504	789	Function Replaced By Constant
tcas/v22	11	581	866	Function Replaced by Constant
tcas/v23	42	325	610	Function Replaced by Constant
tcas/v24	7	287	572	Function Replaced by Constant
tcas/v25	3	353	638	ROR
tcas/v26	11	NF		Missing Code
tcas/v27	10	NF		Missing Code
tcas/v28	76	16	286	Logic Change
tcas/v29	18	NF		Logic Change
tcas/v30	58	NF		Logic Change
tcas/v31	14	394	664	Code Added
tcas/v32	2	71	341	Code Added
tcas/v33	89	NF		Mutation In Array Declaration
tcas/v34	77	NF		Logic Change
tcas/v35	76	16	286	Logic Change
tcas/v36	126	1503	1827	Integer mutation
tcas/v37	93	NF		Constant to Variable

Erroneous design	Error detected times	Number of mutations required to correct error, simple ranking.		Error Class	
tcas/v38	91	NF		Mutation In Array Declaration	
tcas/v39	3	353	638	ROR	
tcas/v40	126	511	796	Logic Change	
tcas/v41	26	244	529	Logic Change	
Highest		1503	1827	Change = |(A-B)/B| *100, % Total Corrected - # of #	
Change, %/Mean	45.1	304	553.71		
Lowest		16	37	31	41
Sum		9424	17165		

*Table 6.3. Mutation-Based Error-Correction experiments for **tcas** design.*

For tcas design dynamic slicing algorithm will reduce number of the required mutations for correction by 45.1 % in mean. Number is defined by internal structure of design, tcas design has number of initialisations at the beginning of code. dynamic slicing allows to discard number of initialisations that are not required for the particular simulation.

schedule design

Erroneous design	Error detected times	Number of mutations required to correct error, simple ranking.		Error Class	
		Localization with dynamic slicing	Localization with no dynamic slicing		
schedule/v1	7	NF		Logic Change	
schedule/v2	210	NF		Logic Change	
schedule/v3	159	907	1146	Integer mutation to float (+1 → +1.1)	
schedule/v4	294	730	945	Integer mutation	
schedule/v5	41	NF		Missing Code	
schedule/v6	7	NF		Logic Change	
schedule/v7	25	NF		Logic Change	
schedule/v8	31	NF		Missing Code	
schedule/v9	23	55	55	ROR	
Highest		907	1146	Total Corrected - # of #	
Mean/Change, %/	21.16	564	715.33		
Lowest		55	55	3	9
Sum		1692	2146		

*Table 6.4. Mutation-Based Error-Correction experiments for **schedule** design.*

Logic change in schedule designs were too complicated to be corrected by FORENSIC algorithms, other faults are successfully corrected.

schedule2 design

Erroneous design	Error detected times	Number of mutations required for to correct error, simple ranking.		Error Class	
		Localization with dynamic slicing	Localization with no dynamic slicing		
schedule2/v1	65	NF		Missing Code	
schedule2/v2	31	NF		Missing Code	
schedule2/v3	34	NF		Missing Code	
schedule2/v4	3	40	64	Missing Code	
schedule2/v5	32	1546	1683	Code Added	
schedule2/v6	3	149	173	Float Mutation	
schedule2/v7	31	858	933	ROR	
schedule2/v8	47	NF		Missing Code	
schedule2/v9	0	E. Not Detected		Logic Change	
schedule2/v10	46	NF		Logic Change	
Highest		1546	1683	Total Corrected # of #	
Mean/Change, %/	9.11	648.25	713.25		
Lowest		40	75	4	10
Sum		2593	2853		

T*able 6.5. Mutation-Based Error-Correction experiments for* ***schedule2*** *design.*

Logic change in schedule2 designs were too complicated to be corrected by FORENSIC algorithms, missed code error is not supported, other errors are successfully corrected.

replace design

Erroneous design	Error detected times	Number of mutations required for to correct error, simple ranking.		Error Class
		Localization with dynamic slicing	Localization with no dynamic slicing	
replace/v1	9	795	808	-1 Removed
replace/v2	7	NF		Missing Code
replace/v3	8	NF		Missing Code
replace/v4	16	NF		Logic Change
replace/v5	74	1574	1601	ROR
replace/v6	8	1504	1531	ROR
replace/v7	11	263	276	Integer mutation
replace/v8	9	NF		Logic Change
replace/v9	26	NF		Missing Code
replace/v10	15	NF		Missing Code
replace/v11	26	NF		Integer mutation + ROR
replace/v12	105	NF		Constant mutation(/2)
replace/v13	17	2846	2911	Added Code
replace/v14	40	NF		Missing Code
replace/v15	60	671	708	+1 added/missing
replace/v16	26	359	372	Added Code
replace/v17	7	NF		Constant mutation ESCAPE →

				NEWLINE	
replace/v18	54	NF		Missing Code	
replace/v19	4	53	53	I/O error/Integer Mutation	
replace/v20	3	NF		Constant mutation ESCAPE → ENDSTR	
replace/v21	2	NF		2 Constants mutations	
replace/v22	4	NF		Logic Change	
replace/v23	4	328	341	+1 added/missing	
replace/v24	43	NF		Missing Code	
replace/v25	1	5002	5123	ROR	
replace/v26	29	3955	4085	+1 added/missing	
replace/v27	77	240	240	Logic Change	
replace/v28	14	1396	1464	Logic Change	
replace/v29	15	1875	1943	Logic Change	
replace/v30	40	357	370	Logic Change	
replace/v31	51	4956	5103	ROR	
replace/v32	0	Err Not Detected		LCR (Logical and bitwise and)	
Highest		5002	5123	Total Repaired # of #	
Mean/Change, %	2.8	1635.88	1683.06		
Lowest		53	53	16	32
Sum		26174	26929		

*Table 6.6. Mutation-Based Error-Correction experiments for **replace** design.*

** were not supported at the moment of the experiments*

tot_info design

Erroneous design	Error detected times	Number of mutations required for to correct error, simple ranking.		Error Class
		Localization with dynamic slicing	Localization with no dynamic slicing	
tot_info/v1	158	NF		Missing Code
tot_info/v2	10	1686	1723	ROR and Integer
tot_info/v3	3	NF		Constant Mutation (1000 → 990)
tot_info/v4	33	1791	1791	Float Rounded*2, l 243
tot_info/v5	29	4050	4098	ASOR (Composite – 1 (one) mutation)
tot_info/v6	45	60	121	Constant mutation l 54
tot_info/v7	123	1289	1300	ROR
tot_info/v8	199	1225	1225	AOR
tot_info/v9	37	4078	4113	ASOR
tot_info/v10	8	NF		Double replaced with Float
tot_info/v11	199	314	314	ASOR
tot_info/v12	33	353	353	Float

				Rounded*3 l 172	
tot_info/v13	128	1823	1834	ROR	
tot_info/v14	2	597	658	ROR	
tot_info/v15	199	413	413	Float Rounded	
tot_info/v16	169	3026	3098	Float Mutation	
tot_info/v17	44	1102	1102	AOR	
tot_info/v18	119	NF		ROR (x2)	
tot_info/v19	85	60	121	Constant Mutation 256 → 56, l 54	
tot_info/v20	25	NF		Missing Code	
tot_info/v21	115	601	662	Constant Mutation, l 74	
tot_info/v22	23	753	753	Float Mutation	
tot_info/v23	71	872	872	Integer Mutation	
Highest		4078	4113	Total Repaired # of #	
Mean/Change, %	1.87	1338.5	1363.94		
Lowest		60	121	18	23
Sum		24093	24551		

*Table 6.7. Mutation-Based Error-Correction experiments for **tot_info** design.*

print_tokens design

Erroneous design	Error detected times	Number of mutations required for to correct error, simple ranking.		Error Class	
		Localization with dynamic slicing	Localization with no dynamic slicing		
print_tokens/v1	1	NF		Added code (CASE)	
print_tokens/v2	12	1511	1587	Added code (CASE) l 219	
print_tokens/v3	2	NF		Missing Code	
print_tokens/v4	5	NF		Constant Mutation in tokens.h file*	
print_tokens/v5	28	NF		Missing Code	
print_tokens/v6	58	NF		Constant Mutation in tokens.h file*	
print_tokens/v7	8	2398	2625	Integer Mutation	
Highest		2398	2625	Total Repaired # of #	
Mean/Change, %	7.19	1954.5	2106		
Lowest		1511	1587	2	7
Sum		3909	4212		

*Table 6.7. T Mutation-Based Error-Correction experiments for **print_tokens** design.*

** include files are not processed.*

print_tokens2 design

Erroneous design	Error detected times	Number of mutations required for to correct error, simple ranking.		Error Class	
		Localization with dynamic slicing	Localization with no dynamic slicing		
print_tokens2/v1	91	NF		Missing Code	
print_tokens2/v2	96	NF		Missing Code	
print_tokens2/v3	0	Err Not Detected		Missing Code	
print_tokens2/v4	207	3273	3584	Integer Mutation	
print_tokens2/v5	124	1221	1247	Constant Mutation	
print_tokens2/v6	221	1051	1051	+1 Added	
print_tokens2/v7	146	2667	2818	Added Code	
print_tokens2/v8	104	2756	2907	Added Code	
print_tokens2/v9	22	3121	3274	Added Code	
print_tokens2/v10	124	899	899	Integer Removed	
Highest		3273	3584	Total Repaired # of #	
Mean/Change, %	5.02	2141.14	2254.29		
Lowest		1051	1051	7	10
Sum		8212	8700		

*Table 6.7. Mutation-Based Error-Correction experiments for **print_tokens2** design.*

Appendix 2. Ranking comparison for the model-based error localization of FORENSIC.

tcas design

Erroneous design	Error detected times	Number of mutations required for correction if error localization with dynamic slicing used with						
		Simple Ranking	Tarantula Ranking	Jaccard Ranking	Ochai Ranking	Ample tool Ranking	Consecutive Ranking	
tcas/v1	132	727	65	65	65	65	65	
tcas/v2	69	114	240	154	154	154	154	
tcas/v3	23	707	93	93	7	7	7	
tcas/v4	26	523	84	84	84	84	84	
tcas/v5	10	NF						
tcas/v6	12	87	590	590	522	522	522	
tcas/v7	36	45	45	45	45	45	45	
tcas/v8	1	44	44	44	44	44	44	
tcas/v9	9	212	796	796	647	647	647	
tcas/v10	14	87	249	249	181	95	95	
tcas/v11	14	87	622	622	449	449	347	
tcas/v12	59	438	2114	2022	1017	2386	931	
tcas/v13	4	465	1887	1887	1422	1446	909	
tcas/v14	50	170	97	97	97	97	97	
tcas/v15	10	NF						
tcas/v16	70	37	37	37	37	37	37	

tcas/v17	35	39	39	39	39	39	39	
tcas/v18	29	45	45	45	45	45	45	
tcas/v19	19	48	48	48	48	48	48	
tcas/v20	18	391	39	39	39	39	39	
tcas/v21	16	504	39	39	39	39	39	
tcas/v22	11	581	984	984	780	780	780	
tcas/v23	42	325	962	962	760	760	760	
tcas/v24	7	287	868	800	800	800	734	
tcas/v25	3	353	168	168	100	100	100	
tcas/v26	11	NF						
tcas/v27	10	NF						
tcas/v28	76	16	459	373	16	349	16	
tcas/v29	18	NF						
tcas/v30	58	NF						
tcas/v31	14	394	142	142	142	142	142	
tcas/v32	2	71	108	108	108	108	108	
tcas/v33	89	NF						
tcas/v34	77	NF						
tcas/v35	76	16	459	373	16	349	16	
tcas/v36	126	1503	59	59	59	59	59	
tcas/v37	93	NF						
tcas/v38	91	NF						
tcas/v39	3	353	168	168	100	100	100	
tcas/v40	126	511	5	5	5	5	5	
tcas/v41	26	244	5	5	5	5	5	
Highest		1503	2114	2022	1422	2386	931	
Mean		304	372.9	359.42	253.94	317.58	226.42	
Lowest		16	5	5	5	5	5	

Sum		9424	11560	11142	7872	9845	7019	31
Maximum Number of Mutations:								3781
% of mutations to inspect		8.04	9.86	9.51	6.72	8.4	5.99	
		Simple Ranking	Tarantula Ranking	Jaccard Ranking	Ochai Ranking	Ample tool Ranking	Consecutive Ranking	

*Table 6.3. Comparison of ranking algorithms for **tcas** design.*

schedule design

Erroneous design	Error detected times	Number of mutations required for correction if error localization with dynamic slicing used with						Error Class
		Simple Ranking	Tarantula Ranking	Jaccard Ranking	Ochai Ranking	Ample tool Ranking	Consecutive Ranking	
schedule/v1	7	NF						
schedule/v2	210	NF	+1 manipulation					
schedule/v3	159	907	127	107	53	87	53	
schedule/v4	294	730	370	370	9	221	9	
schedule/v5	41	NF	Add Code					
schedule/v6	7	NF	Constant to Another Constant (and v1)					
schedule/v7	25	NF	Disable Condition * 2					
schedule/v8	31	NF	Add Code					
schedule/v9	23	55	194	194	194	194	187	
Highest		907	370	370	194	221	187	
Mean		564	230.33	223.67	85.33	167.33	83	
Lowest		55	127	107	9	87	9	3
Sum		1692	691	671	256	502	249	
Maximum Number of Mutations:								3250
% of mutations to inspect		17.35	7.09	6.88	2.63	5.15	2.55	
		Simple Ranking	Tarantula Ranking	Jaccard Ranking	Ochai Ranking	Ample tool Ranking	Consecutive Ranking	

*Table 6.3. Comparison of ranking algorithms for **schedule** design.*

schedule2 design

Erroneous design	Error detected times	Number of mutations required for correction if error localization with dynamic slicing used with						
		Simple Ranking	Tarantula Ranking	Jaccard Ranking	Ochiai Ranking	Ample tool Ranking	Consecutive Ranking	
schedule2/v1	65	NF						
schedule2/v2	31	NF	Add Code (and v1)					
schedule2/v3	34	NF	Add Code					
schedule2/v4	3	40	1318	1318	1318	1318	1318	
schedule2/v5	32	1546	2381	2381	2262	2684	1967	
schedule2/v6	3	149	1042	1042	1042	1460	876	
schedule2/v7	31	858	2906	2906	2467	3001	2180	
schedule2/v8	47	NF	Add Code					
schedule2/v9	0	E. Not Detected	Function by another function with parameter 0. v10-Add condition					
schedule2/v10	46	NF						
Highest		1546	2906	2906	2467	3001	2180	
Mean		648.25	1911.75	1911.75	1772.25	2115.75	1585.25	
Lowest		64	612	612	612	845	534	4
Sum		2593	7647	7647	7089	8463	6341	
Maximum Number of Mutations:								4228
% of		15.33	45.22	45.22	41.92	50.04	37.49	

mutations to inspect								
		Simple Ranking	Tarantula Ranking	Jaccard Ranking	Ochai Ranking	Ample tool Ranking	Consecutive ranking	

*Table 6.3. Comparison of ranking algorithms for **schedule2** design.*

replace **design**

Erroneous design	Error detected times	Number of mutations required for correction if error localization with dynamic slicing used with						
		Simple Ranking	Tarantula Ranking	Jaccard Ranking	Ochai Ranking	Ample tool Ranking	Consecutive Ranking	
replace/v1	9	795	7	7	7	7	7	
replace/v2	7	NF						
replace/v3	8	NF						
replace/v4	16	NF						
replace/v5	74	1574	1643	1483	273	554	273	
replace/v6	8	1504	1011	1011	936	628	485	
replace/v7	11	263	87	87	87	87	87	
replace/v8	9	NF						
replace/v9	26	NF						
replace/v10	15	NF						
replace/v11	26	NF						
replace/v12	105	NF						

replace/v13	17	2846	11	11	11	11	11	
replace/v14	40	NF						
replace/v15	60	671	1949	1949	1949	2429	1949	
replace/v16	26	359	183	183	183	183	183	
replace/v17	7	NF						
replace/v18	54	NF						
replace/v19	4	53	4781	4781	3193	6521	2512	
replace/v20	3	NF						
replace/v21	2	NF						
replace/v22	4	NF						
replace/v23	4	328	2214	2187	1570	1496	152	
replace/v24	43	NF						
replace/v25	1	5002	59	59	59	59	59	
replace/v26	35	3955	66	66	66	66	66	
replace/v27	77	240	130	6	6	6	6	
replace/v28	14	1396	183	183	183	183	183	
replace/v29	15	1875	181	181	181	181	181	
replace/v30	40	357	181	181	181	181	181	
replace/v31	51	4956	88	88	88	88	88	
replace/v32	0	Err Not Detected						
Highest		5002	4781	4781	3193	6521	2512	
Mean		1635.88	798.38	778.94	560.81	792.5	401.44	
Lowest		53	6	6	6	6	6	16

Sum		26174	12774	12463	8973	12680	3965	
Maximum Number of Mutations:								13373
% of mutations to inspect		12.23	5.97	5.82	4.19	5.93	3	
		Simple Ranking	Tarantula Ranking	Jaccard Ranking	Ochai Ranking	Ample tool Ranking	Consecutive Ranking	

*Table 6.3. Comparison of ranking algorithms for **replace** design.*

tot_info design

Erroneous design	Error detected times	Number of mutations required for correction if error localization with dynamic slicing used with						
		Simple Ranking	Tarantula Ranking	Jaccard Ranking	Ochai Ranking	Ample tool Ranking	Consecutive Ranking	
tot_info/v1	158	NF						
tot_info/v2	10	1686	1675	1675	1418	4804	974	
tot_info/v3	3	NF						
tot_info/v4	33	1791	3207	3176	2899	3187	1843	
tot_info/v5	29	4050	1354	1354	855	888	1	
tot_info/v6	45	60	1396	1356	200	6349	200	
tot_info/v7	123	1289	1838	1798	745	1162	745	
tot_info/v8	199	1225	782	525	525	525	525	

tot_info/v9	37	4078	1337	1274	193	193	53	
tot_info/v10	8	NF						
tot_info/v11	199	314	241	241	241	241	241	
tot_info/v12	33	353	2021	1990	1275	1732	1275	
tot_info/v13	128	1823	2395	2297	1279	2403	1279	
tot_info/v14	2	597	135	135	135	135	109	
tot_info/v15	199	413	340	340	340	340	340	
tot_info/v16	169	3026	3975	2831	2568	3894	2568	
tot_info/v17	44	1102	1818	1747	442	1366	442	
tot_info/v18	119	NF						
tot_info/v19	85	60	984	226	126	6635	95	
tot_info/v20	25	NF						
tot_info/v21	115	601	5816	5608	389	6654	354	
tot_info/v22	23	753	442	442	380	380	380	
tot_info/v23	71	872	1517	1505	212	1387	212	
Highest		4078	5816	5608	2899	9062	2568	
Mean		1338.5	1737.39	1584.44	790.11	2348.61	646.44	
Lowest		60	135	126	126	135	1	18
Sum		24093	31273	28520	14222	42275	11636	
Maximum Number of Mutations:								8054
% of mutations to inspect		16.62	21.57	19.67	9.81	29.16	8.03	
		Simple Ranking	Tarantula Ranking	Jaccard Ranking	Ochai Ranking	Ample tool Ranking	Consecutive Ranking	

*Table 6.3. Comparison of ranking algorithms for **tot_info** design.*

**2 Intresting case, Valid Correction is '1.0' -> '6.0' mutation in 'D.5756=a+1.0;' (line 243, col 15), error is in line 229, constant removed.*
**3 Valid Correction is '5' -> '3' mutation in 'j<=5' (line 172, col 2), error in line 175*

print_tokens design

Erroneous design	Error detected times	Number of mutations required for correction if error localization with dynamic slicing used with						
		Simple Ranking	Tarantula Ranking	Jaccard Ranking	Ochai Ranking	Ample tool Ranking	Consecutive Ranking	
print_tokens/v1	1							
print_tokens/v2	12	1511	1671	1671	1671	2308	1566	
print_tokens/v3	2	NF	Add Code (and v1 add case)					
print_tokens/v4	5	NF	.h file not processed					
print_tokens/v5	28	NF	Add code					
print_tokens/v6	58	NF	.h file not processed (and v7 model-dependent)					
print_tokens/v7	8	2398	78	78	78	78	78	
Highest		2398	1671	1671	1671	2308	1566	
Mean		1954.5	874.5	874.5	874.5	1193	822	
Lowest		1511	78	78	78	78	78	2
Sum		3909	1749	1749	1749	2386	1644	
Maximum Number of Mutations:								9667
% of mutations		20.22	9.05	9.05	9.05	12.34	8.5	

to inspect								
		Simple Ranking	Tarantula Ranking	Jaccard Ranking	Ochai Ranking	Ample tool Ranking	Consecutive Ranking	
Rate		I	V	IV	II	III		

*Table 6.4. Comparison of ranking algorithms for **print_tokens** design.*
** include files are not processed.*
***Valid Correction is 'token_ind' -> '-token_ind' mutation in 'token_ind>9' (line 281, col 5)*

print_tokens2 design

Erroneous design	Error detected times	Number of mutations required for correction if error localization with dynamic slicing used with						
		Simple Ranking	Tarantula Ranking	Jaccard Ranking	Ochai Ranking	Ample tool Ranking	Consecutive Ranking	
print_tokens2/v1	91	NF						
print_tokens2/v2	96	NF	Add Code (and v1 add code)					
print_tokens2/v3	0	Err Not Det	Add Code					
print_tokens2/v4	207	3273	1294	683	60	60	60	
print_tokens2/v5	124	1221	3	3	3	3	3	
print_tokens2/v6	221	1051	59	19	19	19	19	
print_tokens2/v7	146	2667	657	386	6	6	6	
print_tokens2/v8	104	2756	2936	2936	33	1525	4	
print_tokens2/v9	22	3121	2024	2024	1110	580	372	
print_tokens2/v10	124	899	575	318	318	318	318	

Highest		3273	2936	2936	1110	1525	372	
Mean		2141.14	1078.29	909.86	221.29	358.71	111.71	
Lowest		696	3	3	3	3	3	7
Sum		14988	7548	6369	1549	2511	782	
Maximum Number of Mutations:								8360
% of mutations to inspect		25.61	12.9	10.88	2.65	4.29	1.34	
		Simple Ranking	Tarantula Ranking	Jaccard Ranking	Ochai Ranking	Ample tool Ranking	Consecutive Ranking	
Rate		V	IV	III	I	II		

*Table 6.5. Comparison of ranking algorithms for **print_tokens2** design.*

tcas design, various consecutive rankings

Erroneous design	Error detected times	Number of mutations required for correction if error localization with dynamic slicing used with									
		Simple Ranking	Tarantula Ranking	Jaccard Ranking	Ochai Ranking	Ample tool Ranking	Simple+Tarantula	Simple+Jaccard	Simple+Ochiai	Simple+Ample	
tcas/v1	132	726	279	279	279	279	279	279	279	279	
tcas/v2	69	114	240	154	154	154	154	154	154	154	
tcas/v3	23	705	93	93	7	7	7	7	7	7	
tcas/v4	26	523	1	1	1	1	1	1	1	1	
tcas/v5	10	NF									
tcas/v6	12	87	590	590	522	522	522	522	522	522	
tcas/v7	36	45	45	45	45	45	45	45	45	45	

tcas/v8	1	44	44	44	44	44	44	44	44	44	
tcas/v9	9	212	796	796	647	647	647	647	647	647	
tcas/v10	14	87	656	656	483	483	347	347	347	347	
tcas/v11	14	87	622	622	449	449	347	347	347	347	
tcas/v12	59	438	2114	2022	1017	2386	931	931	931	931	
tcas/v13	4	465	1887	1887	1422	1446	909	909	909	909	
tcas/v14	50	170	97	97	97	97	97	97	97	97	
tcas/v15	10	NF									
tcas/v16	70	37	37	37	37	37	37	37	37	37	
tcas/v17	35	39	39	39	39	39	39	39	39	39	
tcas/v18	29	45	45	45	45	45	45	45	45	45	
tcas/v19	19	48	48	48	48	48	48	48	48	48	
tcas/v20	18	391	977	977	758	758	758	758	758	758	
tcas/v21	16	504	1054	1054	939	939	871	871	871	871	
tcas/v22	11	581	984	984	780	780	780	780	780	780	
tcas/v23	42	325	962	962	760	760	760	760	760	760	
tcas/v24	7	287	868	800	800	800	734	734	734	734	
tcas/v25	3	353	168	168	100	100	100	100	100	100	
tcas/v26	11	NF									
tcas/v27	10	NF									
tcas/v28	76	16	459	373	16	349	16	16	16	16	
tcas/v29	18	NF									
tcas/v30	58	NF									
tcas/v31	14	394	314	314	314	314	314	314	314	314	
tcas/v32	2	71	164	164	164	164	164	164	164	164	

tcas/v33	89	NF									
tcas/v34	77	NF									
tcas/v35	76	16	459	373	16	349	16	16	16	16	
tcas/v36	126	1503	59	59	59	59	59	59	59	59	
tcas/v37	93	NF									
tcas/v38	91	NF									
tcas/v39	3	353	168	168	100	100	100	100	100	100	
tcas/v40	126	511	5	5	5	5	5	5	5	5	
tcas/v41	26	244	5	5	5	5	5	5	5	5	
Highest		1503	2114	2022	1422	2386	931	931	931	931	
Mean		303.9	460.61	447.13	327.48	393.9	296.16	296.16	296.16	296.16	
Lowest		16	5	5	5	5	5	5	5	5	31
Sum		9421	14279	13861	10152	12211	9181	9181	9181	9181	
Maximum Number of Mutations:											3781
% of mutations to inspect		8.04	12.18	11.83	8.66	10.42	7.83	7.83	7.83	7.83	
		Simple Ranking	Tarantula Ranking	Jaccard Ranking	Ochiai Ranking	Ample tool Ranking	Simple+Tarantula	Simple+Jaccard	Simple+Ochiai	Simple+Ample	

*Table 6.3. Comparison of ranking algorithms for **tcas** design, different combinations of consecutive ranking.*

Appendix 3. Curriculum Vitae

Curriculum vitae

1. Personal data

Name: Urmas Repinski

Date and place of birth: 12.03.1981, Kohtla-Järve, Estonia.

E-mail address: urrimus@hotmail.com

2. Education

Educational institution	Graduation year	Education (field of study/degree)
Jõhvi City Art School	1998	Art/Basic
Kohtla-Järve City Ahtme Gymnasium	1999	Secondary
University of Tartu	2004	Computer Science/BSc
Tallinn University of Technology	2010	Computer Science/MSc
Tallinn University of Technology	2014	Computer Science/PhD

3. Language competence/skills (fluent, average, basic skills)

Language	Level
russian	native
estonian	fluent
english	fluent

german	basic

4. Professional employment

Period	Organisation	Position
December 2003 – May 2006	Evotec Technologies GMBH	Software Developer
November 2006 – December 2007	Titanium Systems	Software Developer
January 2008 – January 2009	Food Factories in UK	General Worker
February 2009 – December 2009	Information Technology N	Software Developer
January 2010 – March 2012	Tallinn University of Technology	Software Developer
May 2012 – July 2012	Satron RD (TOP-Tronic)	Software Developer

5. Research activity, including honours and dissertation supervised

List of Publications:

Raik, Jaan; Repinski, Urmas; Hantson, Hanno; Jenihhin, Maksim; Di Guglielmo, Giuseppe; Pravadelli, Graziano; Fummi, Franco (2012). Combining Dynamic Slicing and Mutation Operators for ESL Correction. 17th IEEE European Test Symposium (1−6). IEEE.

Repinski, Urmas; Raik, Jaan (2012). Comparison of Model-Based Error Localization Algorithms for C Designs. Proc. of 10th East-West Design & Test Symposium, Kharkov, Ukraine, September 14-17, 2012 (1−4). IEEE Computer Society Press.

Hantson, H.; Repinski, U.; Raik, J.; Jenihhin, M.; Ubar, R. (2012). Diagnosis and Correction of Multiple Design Errors Using Critical Path Tracing and Mutation Analysis. In: 13th IEEE Latin-American Test Workshop Proceedings (27−32). IEEE Computer Society Press.

Raik, Jaan; Repinski, Urmas; Jenihhin, Maksim; Chepurov, Anton (2011). High-Level Decision Diagram Simulation for Diagnosis and Soft-Error Analysis. Design and Test Technology for Dependable Systems-on-Chip (294−309). IGI Publishing.

Raik, Jaan; Repinski, Urmas; Ubar, Raimund; Jenihhin, Maksim; Chepurov, Anton (2010). High-level design error diagnosis using backtrace on decision diagrams. 28th Norchip Conference 15-16 November 2010, Tampere, Finland, IEEE.

Урмас Репинский (2012), Верификация на основе симуляции с нахождением и исправлением ошибок для с-дизайнов, Програмные продукты и системы, № 100, 2012 год, стр. 229-237.

Urmas Repinski (2012). Model-based Verification with Error Localization and Error Correction for C Designs, Програмные продукты и системы (international journal Software and Systems), № 100, 2012, pp. 221-229.

Bloem, Roderick; Drechsler, Rolf; Fey, Goerschwin; Finder, Alexander; Hofferek, Georg; Koenighofer, Robert; Raik, Jaan; Repinski, Urmas; Suelflow, Andre (2012). FoREnSiC - An Automatic Debugging Environment for C Programs, Haifa Verification Conference (HVC'2012), pp 1-6, IBM Research Labs, Haifa, Israel: IBM.

Appendix 4. Elulookirjeldus

Elulookirjeldus

1. Isikuandmed

Ees- ja perekonnanimi: Urmas Repinski

Sünniaeg ja -koht: 12.03.1981, Kohtla-Järve, Estonia.

Kodakondsus: Estonia/EU

E-posti aadress: urrimus@hotmail.com

2. Hariduskäik

Õppeasutus (nimetus lõpetamise ajal)	Lõpetamise aeg	Haridus (eriala/kraad)
Jõhvi Linna Kunstikool	1998	Kunst/Põhi
Kohtla-Järve Linna Ahtme Gümnaasium	1999	Kesk
Tartu Ülikool	2004	Arvutiteadus/ BSc
Tallinna Tehnika Ülikool	2010	Arvutiteadus/ MSc
Tallinna Tehnika Ülikool	2014	Arvutiteadus/ PhD

3. Keelteoskus (alg-, kesk- või kõrgtase)

Keel	Tase
vene	emakeel
eesti	kõrg
inglise	kõrg

saksa	alg

4. Teenistuskäik

Töötamise aeg	Tööandja nimetus	Ametikoht
Detsember 2003 – Mai 2006	Evotec Technologies GMBH	Tarkvaraaren daja
November 2006 – Detsember 2007	Titanium Systems	Tarkvaraaren daja
Jaanuar 2008 – Jaanuar 2009	Toidutehased Suurbritaanias	Üldtöötaja
Veebruar 2009 – Detsember 2009	Information Technology N	Tarkvaraaren daja
Jaanuar 2010 – Märts 2012	Tallinna Tehnika Ülikool	Tarkvaraaren daja
Mai 2012 – Juuli 2012	Satron RD (TOP-Tronic)	Tarkvaraaren daja

5. Teadustegevus, sh tunnustused ja juhendatud lõputööd

Publikatsioonid:

Raik, Jaan; Repinski, Urmas; Hantson, Hanno; Jenihhin, Maksim; Di Guglielmo, Giuseppe; Pravadelli, Graziano; Fummi, Franco (2012). Combining Dynamic Slicing and Mutation Operators for ESL Correction. 17th IEEE European Test Symposium (1−6). IEEE.

Repinski, Urmas; Raik, Jaan (2012). Comparison of Model-Based Error Localization Algorithms for C Designs. Proc. of 10th East-West Design & Test Symposium, Kharkov, Ukraine, September 14-17, 2012 (1−4). IEEE Computer Society Press.

Hantson, H.; Repinski, U.; Raik, J.; Jenihhin, M.; Ubar, R.

(2012). Diagnosis and Correction of Multiple Design Errors Using Critical Path Tracing and Mutation Analysis. In: 13th IEEE Latin-American Test Workshop Proceedings (27−32). IEEE Computer Society Press.

Raik, Jaan; Repinski, Urmas; Jenihhin, Maksim; Chepurov, Anton (2011). High-Level Decision Diagram Simulation for Diagnosis and Soft-Error Analysis. Design and Test Technology for Dependable Systems-on-Chip (294−309). IGI Publishing.

Raik, Jaan; Repinski, Urmas; Ubar, Raimund; Jenihhin, Maksim; Chepurov, Anton (2010). High-level design error diagnosis using backtrace on decision diagrams. 28th Norchip Conference 15-16 November 2010, Tampere, Finland, IEEE.

Урмас Репинский (2012), Верификация на основе симуляции с нахождением и исправлением ошибок для с-дизайнов, Програмные продукты и системы, № 100, 2012 год, стр. 229-237.

Urmas Repinski (2012). Model-based Verification with Error Localization and Error Correction for C Designs, Програмные продукты и системы (international journal Software and Systems), № 100, 2012, pp. 221-229.

Bloem, Roderick; Drechsler, Rolf; Fey, Goerschwin; Finder, Alexander; Hofferek, Georg; Koenighofer, Robert; Raik, Jaan; Repinski, Urmas; Suelflow, Andre (2012). FoREnSiC - An Automatic Debugging Environment for C Programs, Haifa Verification Conference (HVC'2012), pp 1-6, IBM Research Labs, Haifa, Israel: IBM.