UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Faculty for Computer Science, Electrical Engineering and Mathematics

MODEL-BASED IDEAL TESTING OF SEQUENTIAL SYSTEMS

ONUR KILINCCEKER

Dissertation submitted in partial
fulfillment of the requirements for the degree of
Doktor-Ingenieur (Dr.-Ing.)

July 2024

EIM-E/376

"*Baba* means father.
Baba means nobody gets left behind or forgotten." [1]

Dedicated to the loving memory of Efrahim Kılınççeker.
1953 – 2012

1 Original quote: "Ohana means family. Family means nobody gets left behind or forgotten." from "Lilo & Stitch" [Film] directed by Chris Sanders and Dean DeBlois (2002) from Walt Disney Pictures. https://www.disneyplus.com/movies/lilo-stitch/1KQztXx3gPGi

SUMMARY OF THE DISSERTATION:

The crucial problem of testing as a validation technique is that it usually shows the presence of faults but not their absence. In 1975, Goodenough and Gerhart (GG) showed that adequately structured tests could demonstrate the absence of errors in a program. The phrase properly entails precisely defined criteria for selecting test cases. However, GG focus on the implementation (source code) of a program rather than its specification as a model, thus considerably limiting the scope of their approach to program codes only.

The present approach proposes to extend GG's concept to model-based analysis and testing sequential systems in toto (hardware and software), based on their formal specification, using the concepts of holistic testing and mutation testing, enriched with knowledge and results from the theory of finite state machines and regular expressions. Holistic testing, in this context, comprises positive testing to show that the system under test functions the way the user desires and negative testing to show that the system under test does not do anything the user would not desire.

On the other hand, mutation testing can be used to modify the system under test to generate its faulty versions to model undesirable situations to be checked against the corresponding implementation when the mutant system under test fails. Thus, the proposed model-based ideal testing (MBIT) enables us to show both the presence and the absence of the faults in the scope of the specification of the system under test.

The approach is limited to sequential systems because of the theoretical features of the techniques used. These techniques are demonstrated and evaluated analytically and experimentally in various case studies: traffic light controllers, graphical user interfaces, and sequential circuits. The results will be critically discussed concerning the advantages and shortcomings of the proposed approach.

ZUSAMMENFASSUNG DER DISSERTATION:

Das entscheidende Problem des Testens als Validierungstechnik besteht darin, dass es normalerweise das Vorhandensein von Fehlern zeigt, aber nicht deren Abwesenheit. 1975 zeigten Goodenough und Gerhart (GG), dass angemessen strukturierte Tests die Fehlerfreiheit eines Programms nachweisen können. Ihr Ansatz beinhaltet präzise Kriterien für die Auswahl von Testfällen. GG konzentrieren sich jedoch eher auf die Implementierung (Quellcode) eines Programms als auf seine Spezifikation als Modell, wodurch der Umfang ihres Ansatzes erheblich auf Programmcodes beschränkt wird.

Der vorliegende Ansatz schlägt vor, das Konzept von GG auf die modellbasierte Analyse und das Testen sequentieller Systeme in toto (Hardware und Software) zu erweitern, basierend auf ihrer formalen Spezifikation, unter Verwendung der Konzepte des ganzheitlichen Testens und des Mutationstests, angereichert mit Erkenntnissen und Ergebnissen aus der Theorie der endlichen Automaten und regulären Ausdrücke.

Ganzheitliches Testen umfasst in diesem Zusammenhang positives Testen, um zu zeigen, dass das getestete System so funktioniert, wie es der Benutzer wünscht, und negatives Testen, um zu zeigen, das getestete System nichts tut, was der Benutzer nicht wünscht.

Techniken von Mutationstests werden verwendet, um den das getestete System so zu modifizieren, dass seine fehlerhaften Versionen generiert werden, um unerwünschte Situationen zu modellieren. Somit ermöglicht uns das vorgeschlagene modellbasierte ideale Testen (MBIT), sowohl das Vorhandensein als auch das Abwesenheit der Fehler im Rahmen der Spezifikation des getesteten Systems zu zeigen.

Der Ansatz ist aufgrund der theoretischen Merkmale der verwendeten Techniken auf sequentielle Systeme beschränkt und wurde anhand verschiedener Fallstudien analytisch und experimentell demonstriert und bewertet: Verkehrsampel, grafische Benutzerschnittstellen und sequentielle Schaltkreise. Ihre Ergebnisse werden hinsichtlich der Vor- und Nachteile des vorgeschlagenen Ansatzes kritisch diskutiert.

ACKNOWLEDGEMENTS

Many thanks to everybody who already helped me to prepare this thesis!

First of all, I cannot thank enough Fevzi Belli, whom I respect immensely, for enabling me to start this thesis and for her unwavering support. I cannot repay my debt to him because of the time we spent together, his contribution to me, and the time he spared. Also, my dear friend and teacher, Moharram Challenger's contributions to me deserve my deepest gratitude. Thanks to him, I had the opportunity to work with Ercüment, Alper, Gizem, Evrim, Burak, and many more students. I also thank them for their contribution to our work. I also thank Geylani Gardaş and Orhan Dağdeviren from the International Computer Institute. I would also like to thank my esteemed friend and colleague, Dr. Sinem Getir, whom I had the opportunity to work with at the International Computer Institute and currently working as a postdoctoral researcher at the University of York.

I would like to thank and express my gratitude to Bekir Taner Dinçer, from Mugla University, Department of Computer Engineering, where I spent a long period of my doctorate, and to my close friend Enis Karaarslan, with whom we spent a lot of valuable work and time.

I also thank Tolga Ayav and Tuğkan Tuğlular, professors of the Computer Engineering Department of İzmir Institute of Technology, where I usually spent the summer of my doctorate. They hosted me in their departments and supported me and their former students, Savaş Takan, Uras Tos and Ekincan Ufuktepe, whom I know thanks to them.

Lastly, thank you so much to my dear love (wife) Banucicek Kandemir Kilincceker for her endless and invaluable support throughout this process. To my precious Mother (Pervin Kilincceker), whose support and prayers I have always felt throughout the process, with whom I spent a long time together, whether we were together or not. The support of my dear brothers Önder Kilincceker and Osman Can Kilincceker is vital, and I cannot pay my debt to them.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

ATPG	Automatic Test Pattern Generation
CT	Context Table
CRE	Contextual Regular Expression
ERE	Extended Regular Expression
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
GUI	Graphical User Interface
GG	Goodenough and Gerhart
HDL	Hardware Description Language
HT	Holistic Testing
MBIT	Model-Based Ideal Testing
MBMT	Model-Based Mutation Testing
MT	Mutation Testing
NT	Negative Testing
OVM	Open Verification Methodology
PT	Positive Testing
RE	Regular Expression
SD	Sequence Detector
SE	SEquential Systems
ST	Syntax Tree
SUT	System Under Test
TLC	Traffic Light Controller
TLS	Traffic Light System
UVM	Universal Verification Methodology
VLSI	Very-Large-Scale Integration

Part I

FOUNDATIONS AND RELATED WORK

CHAPTER 1

INTRODUCTION

In 1970, Dijkstra stated that program testing could be used to show the presence of bugs, but not to show their absence [1]. To find all program bugs, an exhaustive test of all possible test cases/scenarios is required, which is very costly (computationally and memory-wise) and sometimes even not feasible. However, in 1975, Goodenough and Gerhart proved a fundamental theorem showing that properly structured tests can demonstrate the presence and absence of faults in a program [2]. This theorem focuses on satisfying two testing requirements, namely the *reliability* requirement that implies consistency of the test results produced, while the second, *validity* requirement, implies competence of the test results. According to this theorem, a test method holding these requirements is called *ideal test* that enables to show presence and absence of faults.

Furthermore, Goodenough and Gerhart indicate that exhaustive testing with termination criterion could satisfy reliability and validity requirements [2]. The main difficulties preventing the application of the ideal test can be 1) selecting the entire domain in exhaustive testing and 2) independence of the goodness of a test set from individual programs [3]. However, Chow [4] recommended using specifications rather than program codes to achieve the ideal test because finding a test data selection strategy that is both valid and reliable is not solvable in general [2, 5] for any program. Therefore, these difficulties can be handled using a higher-level of abstraction, such as system specification or model, instead of the actual program.

A clear and comprehensive definition of the sequential system (SE) is not given in the literature. This concept has different meanings according to its application areas (for example, circuit theory [6]). The sequential systems discussed in the thesis are similar to the sequential circuit definition given in circuit theory [6]. However, this definition may cause the scope of this concept to be restricted to circuits. Therefore, to give a clear definition, the sequential system defined in this thesis is a deterministic system that can be represented by a finite state machine. In addition, in this system, which consists of consecutive events, an event can affect the results of the event that follows it, and a transition can be made from one event to more than one event. To this end, behavioral-level hardware description language

(HDL) and graphical user interface (GUI) programs are considered sequential systems within the scope of this thesis.

Nowadays, the Hardware Description Language (HDL) is one of the widely used methods for designing digital hardware systems (such as embedded systems). HDL design paves the way for specifying the hardware using a software program. Therefore, it is possible to utilize software-testing techniques such as the ideal test on HDL designs.

In this thesis, an approach is proposed for validating both the presence and the absence of a set of faults in the model of a hardware design, that is, the specification of the HDL program, using the well-known software testing methods *Holistic* testing [7, 8] and *Mutation* testing [9–11].

The proposed approach introduces the necessary steps to apply the ideal testing for the (pre-silicon) validation of the HDL programs and addresses the following common fault types in HDL [12, 13]: single output bit stuck-at 0/1, case bit stuck-at 0/1, condition stuck-at True/False, their combinations. However, the approach does not address conventional gate-level faults such as stuck-at and bridging.

The current approach differs from the manufacturing testing (for example, Automatic Test Pattern Generation (ATPG)) that targets these manufacturing faults. The similarity between the names of the fault models may lead to misunderstanding. The single output bit stuck-at 0/1 for pre-silicon validation refers to the output bit(s) at the behavioral level HDL either stuck at 0 or 1. The stuck-at-fault model for manufacturing testing refers to a functional fault on a Boolean Logic and the corresponding logic that becomes stuck-at either 1 or 0.

Graphical User Interface (GUI) testing is an evaluation process for the correctness of the GUI that is considered a sequential system in this thesis. GUI testing is a very significant section of software engineering and a crucial part of the software development life cycle. It must be a section of the development process from the beginning of application development. In general, the correctness and usability of software applications are essential and may constitute a significant reason to choose one software over another. To attract users, software developers must consider the user experience and GUI of their applications and their correctness. Flaws or faults in GUIs will result in dissatisfaction among the customers. Because of this, catching flaws and hidden faults in an application GUI is critical before deploying the software.

GUI testing is a process of testing the visual elements and their design to limit the probable problems. Component type, size, color, and font are just a few examples of those elements that one can test in an application. More importantly, the business logic of an application can be tested with GUI testing via automation. GUI testing can detect faults in an application with the help of automation tools. Manual

GUI testing is a prolonged and costly process. With test automation, a significant reduction in test time and cost could be achieved.

As with other software testing methods, GUI testing approaches have been suggested to indicate the presence of an fault. However, functional faults are mostly caused by GUI components.

Definition 1.1. *Functional fault is an event-based fault in which the system achieves the final event without providing the expected output.*

An example of the functional fault category is called the "Action" fault in the literature [14], which is frequently encountered in these components and can be given when a GUI user presses a button and there is no action or a faulty action. When the user presses a save button for saving reservations in a hotel management system, but the button does not respond as expected and does not save the reservations, that defines *no action*. Instead of detecting the presence of such faults, showing their absence will make it easier for the tester and prevent the GUI user from experiencing such a fault. This thesis suggests a method that shows both the presence and absence of the fault in this respect (see details on Chapter 7.2 on page 77, paragraph 4).

Along with introducing a toolchain for automated GUI testing, this thesis introduces a methodology for GUI testing by addressing functional faults. It uses Holistic Testing (HT) [7, 8] and Mutation Testing (MT) [9, 10] to achieve ideal testing of the specification (model) of a GUI instead of its program code.

The proposed approach is called Model-based Ideal Testing (MBIT). This methodology is rather general and can be adapted to other application domains. In this thesis, other approaches are proposed on applying MBIT to the validation of hardware design and GUI testing as they are considered sequential systems. The thesis adopted the HT because it is an integrated and complementary view that uses negative testing (NT) aside from positive testing (PT). The HT acquires the legal (expected) test inputs using the fault-free model, which is applied to the GUI under test for the PT.

In positive testing, the system is validated against legal (correct, regular) inputs that are expected data generated from the original (supposedly fault-free) model, which is the conventional way of testing. In negative testing, the system is validated against illegal (faulty, irregular) inputs that are unexpected data generated from a faulty (mutant) model.

Conventionally, the HT offers an integrated perspective as a joint test of expected and unexpected functions. For example, for a hotel reservation system, it is a function that the user is expected to be able to reverse a hotel room using the system by providing requested information successfully. An unexpected function on a hotel reservation website could be a feature that randomly changes the selected dates for a reservation without notifying the user. The HT integrates this bi-directional perspective into its test methods. The present method applies the necessary steps for HT to a model-based testing approach. It uses different and specific models that contain expected and unexpected functions. While the conventional HT uses models that use expected functions to test unexpected functions in certain studies, this thesis uses a separate model for each unexpected function. In this way, it is possible to obtain test suites specific to each unexpected function. While advocating testing all certain unexpected functions with a single set of tests, this thesis argues that different sets of tests are required for each different function.

Mutation testing, a well-known method for test evaluation [11, 15], is a fault-oriented validation method which uses mutants obtained by injecting faults using mutation operators into the system and/or its model. Tests can then detect (kill) mutants, and the effectiveness of a given test set can be determined by the mutation score, that is, the percentage of the killed mutants [16]. Mutation testing has been extended to the model level, leading to model-based mutation testing [8, 17, 18].

This thesis employs the code-based mutation test (CBMT) for assessment and the model-based mutation test (MBMT) to produce model-based mutants. In contrast to traditional MBMT, mutant models are used in this thesis for test generation. Furthermore, conventional CBMT methods produce mutants at random using suitable mutation operators. There are a lot of duplicate mutations that have little to do with real faults. Furthermore, certain mutations may be identical to the system being tested. One of the most difficult aspects of mutation testing involves excluding equivalent mutations. Although related scenarios arise in MBMT methods, randomly generated mutant models in MBMT also lead to non-determinism. As a result, defining analogous and non-deterministic mutant models in MBMT is a difficult task. This thesis takes a more comprehensive and novel approach to the problems that exist in CBMT and MBMT by using model-based mutants and code-based mutants that are unique to the system's faults.

Fault, error, and failure terms can cause misunderstandings. A fault is simply a static change in the system under test, an error is an incorrect internal state caused by some fault, and failure is an external incorrect behavior of the system [19–21]. This thesis generally uses the term fault as a root event and sometimes uses errors caused by these faults.

This thesis focuses on the following research questions (HDL-RQs) for behavioral level HDL programs:

HDL-RQ 1. Is it possible to apply the code-based ideal testing approach (proposed by Goodenough and Gerhart [2]) to model-based testing and exemplary to HDL programs that will be viewed as hardware specifications?

- HDL-RQ 1.1 What kinds of HDL systems can be addressed? Sequential, combinational, cyber-physical systems, embedded systems? What are the borders?
- HDL-RQ 1.2 What kinds of HDL faults can be targeted?
- HDL-RQ 1.3 What are the outcomes of applying the ideal testing?

HDL-RQ 2. What are the costs of applying ideal testing in terms of time and size of test suites (a set of test sequences), and is this approach scalable?

- HDL-RQ 2.1 How does this cost affect the scalability of applying ideal testing?
- HDL-RQ 2.2 What are the complexities of the algorithms used?

A final discussion of these research questions is included in Section 7.4.1.3, also indicating to what extent they have satisfactorily been answered.

This thesis provides the following contributions, considering the current state of the literature for MBIT of behavioral level HDL programs:

1. Unlike conventional usages of the HT and MT, they are adapted to achieve a novel validation approach proposed to target the HDL faults at the behavioral HDL design.
   - The HT is adapted to provide different test suites for each different HDL fault.
   - The MT is tailored by acquiring mutants for test generation.
   - A methodology is provided to test the presence and absence of real faults in the HDL designs.
2. This thesis showed formally that the proposed approach satisfies the requirements of the ideal testing.
3. An experimental evaluation for the current methodology is provided.
   - Three HDL case studies are experimented on to evaluate the methodology.
   - Five different test generation approaches and tools are adapted and used in the experimental study.
4. Tool support is created and made available.
   - The MBIT is partially automated for HDL validation.

- The toolchain including examples and details are provided in a bundle [1].

The proposed method is far more general than only considering the validation of the HDL program to show both the presence and absence of HDL faults. The MBIT proposed by this work can be easily adaptable and applicable to any software and systems domain that employs model-based testing. This generality of the MBIT comes from the methods (the HT and MT) adapted to achieve ideal testing. Also, any model-based test generation method can be easily adapted to the current methodology as this thesis employs several of these methods for evaluation. Finally, this thesis selects a coverage-guided test generation method that is based on an analysis of the model of the HDL.

For GUI testing, the experimental and theoretical studies carried out within the scope of this study are designed to answer the research questions (GUI-RQs) given below:

1. GUI-RQ 1. Is it practically and theoretically possible to offer an ideal testing [2] approach for GUI testing?
    - GUI-RQ 1.1 What types of systems can be tested in this way?
    - GUI-RQ 1.2 What types of faults can be targeted with the proposed approach?
2. GUI-RQ 2. What is the cost of applying this approach to GUI testing?
3. GUI-RQ 3. How is scalability affected?

Considering the experimental and theoretical studies carried out in this work, the above-mentioned research questions are examined in detail in Section 7.4.2.2.

In this thesis, MBIT is adapted and extended to GUI testing for targeting GUI-related functional faults and evaluating MBIT on case studies, including comparison with three different approaches. The thesis uses two selection criteria for the algorithmic correctness of the models. To this end, the thesis provides the following contributions:

1. Different than conventional usages of the HT and MT, they are adapted to achieve the ideal test suites for GUI testing for the presence and absence of faults.
    - The HT is adapted by offering different test suites for each different faults
    - The MT is customized by acquiring mutants for test generation.

1 MBIT4HW, `https://github.com/kilincceker/MBIT4HW`

    - A methodology is provided to target functional faults for GUI testing, including an informal proof for being MBIT.

2. An experimental evaluation of the current methodology is presented.
    - Two GUI case studies are used to evaluate MBIT.
    - Three different test generation approaches are utilized for comparison.

3. A tool support is developed and provided
    - The MBIT is partially automated for GUI testing.
    - The toolchain, including examples and details, are provided in a bundle [2].

A model represents abstracted functionalities of a system under test. However, the correctness of the model that represents the system needs to be addressed and assumed within its construct. Therefore, an assumption given below needs to be presented to address this. Otherwise, a wrong model may threaten the validity of the proposed methodology.

Assumption 1.1. *Model correctness in MBIT is a requirement to ensure the model used for test generation complies with the system modeled.*

This assumption is also addressed in Section 6.2.3, and some possible mitigation methods will be presented in Section 7.4.2.3.

1.1 OVERVIEW OF PUBLICATIONS

Various articles have been published in peer-reviewed workshops, conferences, and journals within the scope of this thesis. Published articles are grouped under the titles of foundations, test generation, and model-based ideal testing (MBIT). Details and related articles about these titles are given in Figure 1.1.1

Foundations consist of modeling, coverage criteria, and scalability sub-categories. The modeling category includes algorithms required for modeling sequential systems in general, transformations between models, and complexity analysis of these algorithms. Coverage Criteria are structural metrics for the utilized models, which are essential for the next category, test generation, and enable the generated test sequences to be produced more effectively and faster. Various publications have been made for a regular expression and contextual regular expression models in this context. Scalability is a challenging problem that model-based testing methods often encounter. Thus, to mitigate

2 MBIT4SW, `https://kilincceker.github.io/MBIT4SW/`

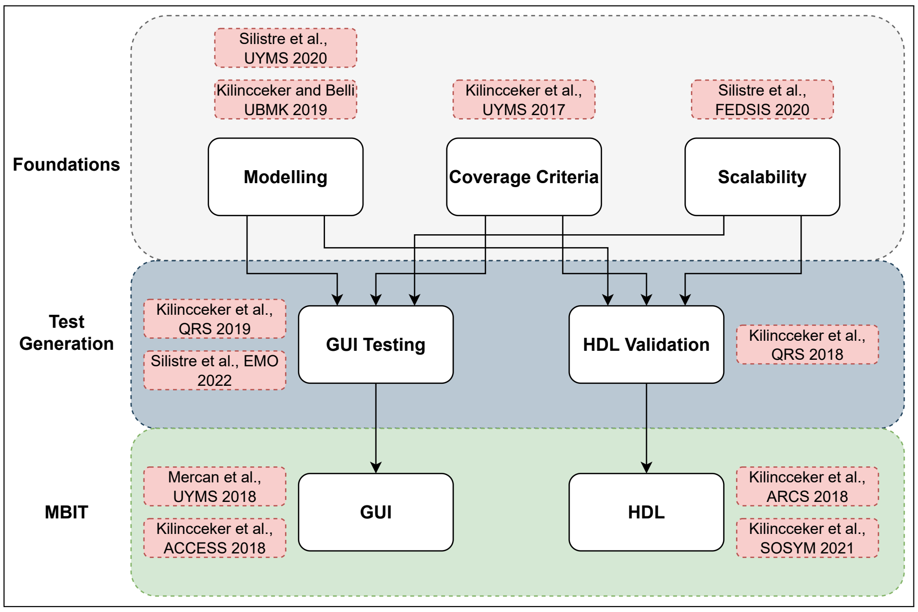

Foundations
Test Generation
MBIT
Silistre et al., UYMS 2020
Kilincceker and Belli UBMK 2019
Kilincceker et al., UYMS 2017
Silistre et al., FEDSIS 2020
Modelling
Coverage Criteria
Scalability
Kilincceker et al., QRS 2019
Silistre et al., EMO 2022
GUI Testing
HDL Validation
Kilincceker et al., QRS 2018
Mercan et al., UYMS 2018
Kilincceker et al., ACCESS 2018
GUI
HDL
Kilincceker et al., ARCS 2018
Kilincceker et al., SOSYM 2021

Figure 1.1.1: Overview of the Publications Related to This Thesis

this problem, an approach has been proposed that automatically identifies sub-models within models and thus enables more effective test generation.

Test generation algorithms differ according to the model used and coverage criteria to assess adequacy as a termination criterion. In this thesis, a test generation approach is proposed for regular expression and contextual regular expression models. This category is divided into two groups HDL validation and GUI Testing. The test suites generated using random, and optimization algorithms for GUI Testing were executed on GUI systems during the test execution step. The results obtained with the help of different algorithms were compared with the proposed method. A similar approach has been taken for the HDL validation.

Different articles have been published for MBIT, which is the most significant contribution to this thesis. GUI systems (web application and mobile application) and HDL systems are presented for this category. All of the approaches put forward in the foundations and test generation categories for GUI and HDL systems have been used and implemented in this category. The studies given in this category form the basis of the thesis. Some ideas and figures have appeared previously in the following publications:

- O. Kilinccceker, E. Turk, F. Belli and M. Challenger, (2021). "Model-based ideal testing of hardware description language (HDL) programs," Software and Systems Modeling, vol. 21, pp. 1-32.
- O. Kilincceker, A. Silistre, F. Belli and M. Challenger, "Model-Based Ideal Testing of GUI Programs–Approach and Case Studies," IEEE Access, 2021, vol. 9, pp. 68966-68984.

- A. Silistre, O. Kilincceker, F. Belli, M. Challenger and G. Kardas, (2022). "Grafiksel Kullanıcı Arayüzü Testi İçin Bir Uçtan Uca Model Tabanlı Yaklaşım (End-to-End Model-based Testing for Graphical User Interface)," EMO Bilimsel Dergi, vol. 12(1), pp. 7-19.
- O. Kilincceker and F. Belli, "An Approach to Extending Sequential Systems for Achieving Fault Tolerance," 2024 (under preparation).
- A. Silistre, O. Kilincceker, F. Belli, M. Challenger, and G. Kardas, "Models in Graphical User Interface Testing: Study Design," 2020 Turkish National Software Engineering Symposium (UYMS), 2020, pp. 1-6.
- A. Silistre, O. Kilincceker, F. Belli, M. Challenger, and G. Kardas, "Community Detection in Model-based Testing to Address Scalability: Study Design," 15th Conference on Computer Science and Information Systems (FedCSIS), 2020, pp. 657-660.
- O. Kilincceker and F. Belli, "Towards Uniform Modeling and Holistic Testing of Hardware and Software," 2019 1st International Informatics and Software Engineering Conference (UBMYK), 2019, pp. 1-6.
- O. Kilincceker, A. Silistre, M. Challenger and F. Belli, "Random Test Generation from Regular Expressions for Graphical User Interface (GUI) Testing," 2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C), 2019, pp. 170-176.
- O. Kilinccceker, E. Turk, M. Challenger, and F. Belli, "Regular Expression Based Test Sequence Generation for HDL Program Validation," 2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C), 2018, pp. 585-592.
- O. Kilincceker, E. Turk, M. Challenger, and F. Belli, "Applying the Ideal Testing Framework to HDL Programs," ARCS Workshop 2018; 31st International Conference on Architecture of Computing Systems, 2018, pp. 1-6.
- G., Mercan, E., Akgündüz, O., Kilincceker, M. Challenger and F. Belli, "Android uygulaması testi için ideal test ön çalışması (A Preliminary Study on Ideal Testing of Mobile Applications)," 2018 Turkish National Software Engineering Symposium (UYMS), 2018, pp. 36-42.
- O., Kilincceker, and F., Belli, (2017). Grafiksel Kullanici Arayuzleri icin Duzenli Ifade Bazli Test Kapsama Kriterleri (Coverage

Criteria For Testing Graphical User Interfaces Based On Regular Expressions). 2017 Turkish National Software Engineering Symposium (UYMS), 2017, pp. 332-343.

1.2 STRUCTURE OF THESIS

This thesis consists of three main parts. Part I summarizes the foundations, related work, and background information. Then, Part II introduces and evaluates new approaches within various case studies. Part III presents further perspectives, conclusions, and future directions. Details on these general main parts are presented below.

In Part I, Chapter 1 summarizes the motivations of the thesis, including research questions. Then, it presents an overview of the publications within the scope of the thesis and provides information on the thesis's structure. Chapter 2 summarizes related work on validation of written in hardware description language, graphical user interface testing, holistic testing, and ideal testing. In Chapter 3, the background information is presented concerning notions used, coverage criteria, and analysis method on utilized models.

In Part II, namely approaches, two test generation approaches based on regular expression and contextual regular expression are presented in Chapter 4 and Chapter 5, respectively. Then, Chapter 6 introduces model-based ideal testing approaches for hardware description language and graphical user interface programs. Case studies, results based on experimental evaluation, and tool support are summarized in Chapter 7.

Finally, in Part III, Chapter 8 introduces an approach for fault tolerance of sequential systems based on extending their models. Chapter 9 concludes this thesis by presenting its important aspects. Chapter 10 finalizes this thesis with possible follow-on projects and future directions.

CHAPTER 2

RELATED WORK

This chapter presents related work and background information for this thesis. Holistic testing (HT), code-based mutation testing, model-based mutation testing, graphical user interface testing, and ideal testing are given in the following sections.

2.1 VALIDATION OF PROGRAMS WRITTEN IN HDL

The purpose of conducting functional testing of software is to find variations or differences between the program and the specification that may cause errors. The specification is an external and accurate description of the software program's behavior. There are test sequences generated from the external specification and executed in the program that are designed to expose the errors that are caused by the discrepancy between the program and specification.

Testing hardware, on the other side, can be done at various abstraction levels, mainly at *structural level* (gate level), *register transfer level* (RTL) or *behavioral level*. At the structural level, the process of generating test sequences is called *ATPG*, which strictly uses a netlist (schematic) representation of the circuit under test [22]. Manufacturing defects, represented by fault models, such as stuck-at, open, bridging, and delay, can be targeted less expensively at higher levels of abstraction, for example, at the behavioral level or at RTL.

Validation of VLSI is carried out in pre-silicon or post-silicon stages of the development flow [23, 24]. The development starts with a specification provided by the customer and then a designer implements a behavioral level design that is specifically used for pre-silicon validation or verification using simulation. After fixing design errors, the designer converts the verified specification to an RTL design which can be used for pre-silicon validation against the specification. RTL can be also used for functional test generation. In post-silicon validation, a silicon die, a chip not packaged yet, is validated against the specification to detect errors missed during the pre-silicon validation stage [23].

Pre-silicon validation is an activity performed on a simulation or emulation of a design to detect errors or bugs in that design. One approach used to realize the design validation is to generate test sequences from the specification of the system and execute them on a simulation or an emulation of the design to detect design errors or bugs. Some errors can escape from the pre-silicon validation process, for which the post-silicon validation is performed to increase product quality. However, detecting and fixing errors in post-silicon validation are expensive and time-consuming. Therefore, detecting as many errors as possible in pre-silicon validation is desirable.

Test generation or testing at the behavioral level for hardware can be done in a similar manner to software testing and requires well-selected fault models to target physical and manufacturing defects [25]. Jervan et al. presented fault models and test generation at the behavioral level [26]. Moreover, it is reported by Lajolo et al. that 'stuck-at fault' models at the gate and behavioral levels are correlated with each other [27]. The study concluded that the test generation performed at the behavioral level is faster and simpler than at the gate level [12].

Validation of HDL programs to detect design faults has been popular since the 1990s. Stumptner and Wotawa propose a model-based approach for the validation of HDL programs to address design faults [28]. The method is called model-based diagnoses (MBD) and attempts to identify behavioral changes in the HDL program. In MBD, a discrepancy between the implementation waveform and the specification waveform addresses the existence of a fault. The method parses specifications implemented in an HDL language and compares execution results obtained from running the implementation to find differences. The MBD is also used to localize faults [29] through the verification of HDL programs. Rather than using the waveform of the specification and implementation by Stumptner and Wotawa [28], Bloem and Wotawa employ a state-transition diagram of the specification and utilize a model-checker to find a counter-example that addresses the design faults [29]. Then, they employ the MBD using the information from the counter-example to localize the detected faults. The thesis employs a similar model used by Bloem and Wotawa [29] for test generation and mutant generation to address specific HDL faults. Different from Stumptner and Wotawa [28] and Bloem and Wotawa [29], this thesis focuses on proposing an ideal testing methodology that addresses both the presence and absence of HDL faults at the behavioral level.

Jervan et al. analyze existing coverage metrics at the behavioral level with the correlation of gate-level stuck-at faults [26]. The analysis results show that the combination of bit and condition coverage can be effectively utilized to evaluate the quality of a test set at the behavioral level. Furthermore, the proposed method by Jervan et al. use the Random Mutation Hill Climber (RMHC) algorithm for test sequence generation [26].

Shin et al. propose a model-based testing approach and a coverage criterion for programmable logic controller (PLC) programs based on its modeling language called Function Block Diagram (FBD) [30]. They use mutation testing to generate mutants of the model to measure the effectiveness of the approach and the coverage criterion. The evaluation is carried out on six FBD models for three test criteria by comparison with random testing. The generated test suites are executed on the models rather than on the actual programs, for which the authors define a future work to use model-driven development techniques to convert the models into executable C programs.

Mens et al. present a validation and test approach for executable state charts for which they developed Sismic, an open-source research prototype tool [31]. They employ a semi-formal natural language to develop executable test scripts for functional testing of specific scenarios using the state charts. They also propose two run-time verification techniques for the state charts, which are based on applying the design by contract (DbC) and specifying behavioral properties. They conduct a controlled experiment to evaluate their approaches with thirteen participants.

In the verification of an HDL design, there is a detailed methodology called Open Verification Methodology (OVM) that is "*the first truly open, interoperable, and proven verification methodology*" [32]. The Universal Verification Methodology (UVM) is a standardized HDL verification methodology derived from the OVM. Unlike previous methodologies from different vendors, UVM and OVM class libraries are used to automate the verification procedures for SystemVerilog HDL.

Adir et al. propose a unified methodology for pre and post-silicon validation that aims to detect design faults that escape detection by pre-silicon validation [33]. The methodology applies the coverage-driven verification (CDV) methodology to the post-silicon validation problem domain. They implement the methodology in a tool called Threadmill [33], which is used for pre-and post-silicon validation of IBM's POWER7 processor. Kannavara presents a unified pre-silicon validation method, including security validation [34]. The paper uses the existing tools used in pre-silicon validation and defines the unified method, which is generally based on developing common collateral between project development and validation teams. Moreover, other methods are proposed by August [35], and Lotfy et al. [36] for pre-silicon validation of mixed-signal circuits and a System-Verilog behavioral model of the phase-locked loops (PLLs).

Pre-silicon validation of a VLSI circuit with an HDL program is similar to functional testing of software. Each attempts to find discrepancies between the product and specification that may be caused by errors or bugs. To this end, this thesis exploits the methods used in software testing for pre-silicon validation of VLSI circuit design to address design faults for both positive and negative testing.

To sum up, this thesis proposes a pre-silicon validation methodology at the behavioral level that is supported by a toolchain to automatize the approach for application of the ideal testing to the given HDL program, targeting faults such as single output bit stuck-at 0/1, case bit stuck-at 0/1, condition stuck-at True/False, their combinations.

2.2 GRAPHICAL USER INTERFACE TESTING

The process of testing the GUI of a software application, that is, one that has a GUI front-end and there are available events("enter a text", "click on a button", "select an item from a dropdown") that can be applied on GUI widgets (for example, "text-field", "button", "dropdown") to perform actions in the system, is called GUI testing. The GUI testing process can be carried out effectively through a well-selected model, that is, Finite State Machine (FSM) [37], Event Flow Graph (EFG) [38, 39], Event Sequence Graph (ESG) [7]. The FSM, EFG, and ESG are graph-based models. Optimization and traversal algorithms need to be applied to them to produce test sequences.

Shehady and Siewiorek implemented a formal way to describe a GUI called a Variable Finite State Machine (VFSM) [37]. The VFSM is then transformed into an FSM to be used in test generation by using a well-known W-Method, which was originally introduced by Chow [4]. The W-Method requires a completely defined FSM, so there could be many NULL transitions within the model. Although the VFSM requires fewer states than the FSM, the proposed algorithm runs on many states of the FSM.

ESG is proposed by Belli et al. [7, 42] to be used in modeling the GUI. A testing method is also proposed for use in this novel model. The ESG describes the events at the vertexes and the relationship of the events at the edges of the given GUI. Testing methods merge the PT and NT to obtain a holistic viewpoint [7]. The system is tested against illegal inputs (incorrect behavior) in the NT. In the PT, it is tested against legal inputs (correct behavior) in compliance with user expectations. The suggested approach is thus generic, describing all correct and incorrect behaviors.

Memon et al. presented a test generation algorithm based on artificial intelligence-based planning using the EFG model [38]. They also propose generating a hierarchical model from the given GUI structure. The EFG model is constructed, and then the planning algorithm is implemented, which involves specifying a collection of operators, an initial state, and a target state. Then, by considering GUI events and interactions, the algorithm produces test sequences between the initial and target states. They are also using GUI model decomposition to deal with the issue of scalability.

Memon recast the existing idea of event-based GUI testing via model-based techniques called event-space exploration strategies (ESES)

Table 2.2.1: Comparison of GUI testing methods

	Model	Coverage Criteria	Pros	Cons
[37]	Variable Finite State Machine (VFSM)	N-switch set cover	Covers nearly all predefined faults	Results excessive number of redundant test cases
[40]	Finite State Machine (FSM)	Complete interaction sequences (CIS)	N/A	N/A
[7]	Event Sequence Graph (ESG)	Complete interaction sequences (CIS)	Testing GUI with the PT and NT	N/A
[38]	GUI Tasks	N/A	Intuitively and easily scalable for larger GUIs	Tasks are chosen by test designer that may yield inadequate coverage
[39]	Event Flow Graph (EFG)	N/A	Quickly generate test cases without scalability problem	N/A
[41]	Event Interaction Graph (EIG)	Genetic algorithm–CIT coverage	Increase the feasible coverage of these suites	N/A

*N/A refers to "Not Available"

[39]. He decreases the cost and effort of event-flow techniques and automates the procedure to enable extensive experiments and simplify the model creation step.

Xie and Memon presented a new concept called Minimal Effective Event Context (MEEC) and used this in an empirical way for fault detection [41]. Generally, GUIs are implemented as a collective of widgets with their event handlers and responses to event handlers. Generating long test cases becomes expensive. The purpose of modeling MEEC is to create an abstract model of GUIs and then generate the shortest "potentially" problematic event sequences for test case generation.

Huang et al. developed a method to repair GUI test suites, which suffer from in-feasibility because graphs are generally used to acquire test cases, and they are created from all possible sequences of events [43]. There is a possibility that an event inside these kinds of test sequences may not be available for execution and terminate early. They used a genetic algorithm to fix these problematic test suites and increase test coverage.

Belli et al. examined current software reliability models and seeked to obtain an experimental understanding of this issue [44]. In the provided work, they indicate that selecting an appropriate modeling technique for GUI testing affects the quality of the assessment process and, hence, the software.

Banerjee et al. researched GUI testing articles and studies and matched them with a systematic mapping technique [45]. They defined selection criteria for studies about GUI testing from the pool of 230 articles written between 1991 and 2011. They classified studies, provided an overview of existing approaches, and spotted areas that require more study and research.

Belli et al. presented a study about reviewing and summarizing existing works on model-based GUI testing [46]. They also provide the PT and NT examples from realistic GUI projects. They gave examples from conventional and modern techniques for model-based GUI testing. They also covered test-case construction and optimization of the process.

Alegroth and Feldt provided a case study, including a comprehensive qualitative study for visual GUI testing (VGT) in industrial practice [47]. The study is conducted on a well-known music streaming application, Spotify. They attempt to answer three research questions about problems, challenges, and limitations for adaptation of automated VGT on the company using the Sikuli [48] test automation tool and Graphwalker [49] model-based testing tool at the industrial level. They also explain why the VGT was abandoned in the company due to organizational changes based on experiences. Finally, they present an automated GUI testing solution for Spotify.

Besides automated methods based on test automation tools, some works utilize machine learning methods, especially deep reinforcement learning [50, 51] for automatically traversing the GUI. Eskonen et al. presented an image-based deep reinforcement learning method for exploring GUI structure [50]. The introduced method mainly focuses on learning GUI behaviors by feeding screenshots of the GUI to the neural network and letting the learning method explore the GUI events. They also compare the exploration efficiency of the algorithm with Q-learning and random exploration methods. However, they do not provide an appropriate experimental evaluation of how the method effectively catches faults. Similar to Eskonen et al. [50], Adamo et al. present a reinforcement learning method for automated GUI testing based on exploration [51]. They utilize a Q-learning algorithm that outperforms the random exploration approach based on experimental evaluation concerning only code coverage efficiency. They also do not provide any information regarding fault coverage.

Table 2.2.1 presents a comparison among the studies related to GUI testing, focusing on each approach's advantages and disadvantages. It also gives the models used in each study and the coverage criteria for the study. In this table, the N-switch set cover is a generalization of a switch cover and refers to the covering switch statement in the software program graph, introduced by Chow [4].

In this thesis, the GUI under test is modelled as given by Silistre et al. [52] and partially by Shehady and Siewiorek [37]. Then, the FSM model is used for mutant generation. However, the FSM models are converted to the RE model for test generation that is different from Silistre et al. [52], and Shehady and Siewiorek [37]. This modeling approach is more similar to the method proposed by Belli [7] in which the author does not offer a test generation approach in contrast to this thesis. In the methods proposed by Memon [39] and Xie and Memon [41], the authors used a very different modeling methodology than the thesis by using various node types for different GUI events. The advantages and disadvantages of these different models for GUI testing are elaborated by Silistre et al. [52]. The main bottlenecks of the exploration-based solutions proposed by Eskonen et al. [50] and Adamo et al. [51] are unnecessary test inputs and automatic exploration of GUI events without knowing the correct input(s) of this event to trigger errors. However, the exploration-based methods are valuable for automatic extraction of the GUI model (called GUI ripping) accompanied by static analysis methods to eliminate low-quality and redundant test suites. Therefore, this thesis focuses on the model-based testing approaches due to their robustness and determinism.

2.3 HOLISTIC TESTING

Belli presented a holistic strategy for modeling and testing, which considers the behavior of the system under both desirable (legal, expected) and undesirable (illegal, unexpected) situations, forming *positive testing* and *negative testing*, respectively [7]. Belli also proposed finite-state automata and regular expression, having equivalent expressive power and test generation capability, for modeling and testing graphical user interfaces using *positive testing* and *negative testing* [7]. The holistic strategy is then applied to graphical user interfaces by Belli [7], web service composition by Belli et al. [53], web application by Belli et al. [54], interactive systems by Belli et al. [55], hardware designs by Kilincceker et al. [56], and android applications by Mercan et al. [57] for both modeling and testing. Kilincceker and Belli introduced uniform modeling and testing for hardware and software design using a holistic strategy [58].

Fraser and Wotawa introduced a test case generation method based on the property relevance of test cases to determine property violations [59]. The proposed method uses model-checkers that employ the Kripke structure as a model formalism to generate these test cases. To assess adequacy, they also propose new coverage criteria for the test generation method by using structural coverage and property relevance. This thesis defines positive and negative test cases, including their test execution results in contrast to the approach by Fraser and Wotawa [59]. The negative test cases are used to show property violations if they pass on the system. The positive test cases, however, are used to show property relevance if it fails on the system. The empirical study concludes that the proposed method is encouraging and provides several advantages, such as the effectiveness of the test cases based on the proposed coverage criteria. However, this method results in a large number of test cases that have a higher execution time. This work differs from the thesis concerning the type of model used, that is, this work uses the Kripke structure as a model, and the current study uses the FSM and RE.

The approach introduced in this thesis assumes that positive testing addresses the presence of faults, whereas negative testing targets the absence of faults.

- Positive Testing: Test sequences generated from the fault-free (original model) are applied to the faulty/mutant program.
- Negative Testing: Test sequences generated from the faulty (mutant model) are applied to fault-free/original programs.

A holistic testing strategy is a major component of the approach introduced in this thesis. The thesis refers to a supposedly fault-free system as an original HDL program, a supposedly fault-free system model as an original model, a faulty system as a mutant HDL program, and a faulty model as a mutant model.

2.4 MUTATION TESTING

DeMillo et al. [9] and Hamlet [10] proposed the MT in their seminal paper. The MT is a fault-oriented technique that uses a given software program's mutation. A mutation contains a simple fault caused by making small changes in the original software program. A generated test data is executed on each mutant, and the results are compared with the result of the original program's test execution results. If the result of the test data differs from the result of the original test data, then the corresponding mutant becomes dead; otherwise, it is still alive because the test data result does not make any difference. Therefore, the two cases could occur. The test data does not contain enough sensitivity to distinguish between the mutant and the corresponding programs, so the mutant is live; thus, there is no test data to detect the fault. Or it is an equivalent mutant; thus, it behaves as equivalent to the corresponding program. A mutant is equivalent to the corresponding program only when there exists no test case (not just any test data) that distinguishes the two. Mutation testing can be used to assess the effectiveness of a given test set using a testing criterion, namely mutation score [60]. Depending on the programming languages, there are several types of mutation operators for generating the mutants [16]. A test set, which is measured for effectiveness, is then executed on the generated syntactic mutants to calculate the mutation score. Once the result of the execution of the test set on the mutant program differs from the result of the execution on the original program, the mutant becomes killed. If any mutant does not differ from the original program, those mutants stay undetected and *live*. Thus, the mutation score is the ratio of the number of *killed* mutants over the total number of mutants.

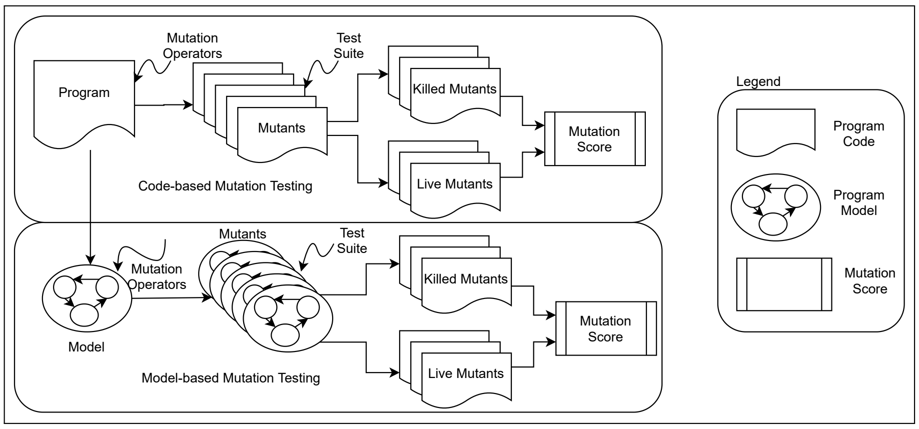


Figure 2.4.1: Code-based and Model-based Mutation Testing

Mutation testing, see Figure 2.4.1, can be applied to the software model: Offutt [11]. This model can be an FSM [17], Event Sequence Graph [8], or Function block diagram [30]. This thesis uses the FSM for mutation testing. Mutation testing offers an adequate artifact to qualify the test suite, in which test cases are executed on the mutants.

This thesis uses the mutation operators to insert, omit, or replace transition(s)/state(s) in the FSM mutation. The combinations of those operators are utilized to materialize faults in the model domain.

DeMillo et al. emphasized the power of the coupling effect that states the test data that distinguish only simple faults could also be sensitive to cover more complex faults [9]. The MT method is a powerful and elegant method that is applied to both software and hardware testing [56, 61]. The only consideration is the cost of this method, which increases very quickly due to the program's size and directly affects the number of mutants. A comprehensive literature review in the form of a "mini-handbook"-style road-map for the MT is given by Papadakis et al. [62].

King and Offutt presented an MT framework with the 22 mutation operators for the Fortran 77 version of the Mothra system, a software testing environment [63]. The Mothra achieves the highest mutation (adequacy) score for the set of test cases executed on mutant and original programs. The Mothra system generates 970 mutants for a 27-line program. These results are computationally and spatially expensive due to the excessive number of mutants. Therefore, the Mothra handles this problem by utilizing incremental compilation.

Wong and Mathur presented an empirical study to reduce unacceptable computational expenses due to the number of mutants [64]. One of the proposed solutions is randomly selected from a subset of all mutants (x). Earlier investigation shows that a random selection of 10 to 100 of all mutants makes dramatic reductions in requiring efforts while keeping the MT's effectiveness. They increase x by 5 up to 40 to examine the cost and power of the MT. Another offered solution is constrained mutation that requires selecting a few specific types of mutants and neglecting the others. They state that proper selection of a small set of mutant types significantly lessens the MT's complexity and still keeps nearly the same fault detection ability of the MT.

Ma et al. introduced the MuJaVa tool for the MT, including the GUI of the Java programming language for both method and class-level mutation with related levels of mutation operators [65]. The method-level mutation operators change the expressions by replacing, deleting, and inserting operators. The class-level mutation operators are responsible for object-oriented attributes: inheritance, polymorphism, and dynamic binding. The MuJava contains the mutant generator, including an engine to detect equivalent mutants, the mutant executor, and the mutant viewer components. However, it is reported that the MuJava is still very slow for a large set of mutants.

Jia and Harman presented a comprehensive analysis and survey for the MT [60]. It is also mentioned that the new trend in the MT will be the semantic effects of mutants rather than syntactic effects.

Fabbri et al. provided an MT technique to validate state chart-based specifications [66]. The technique uses a set of mutation operators: the finite state machine, extended finite state machine, and state charts-feature-based operators [66]. The set contains 37 mutation operators. They also utilize an abstraction strategy, namely the Hierarchical Incremental Testing Strategy (HITS), to make the technique more feasible for conducting a modular and incremental testing activity. However, they also state that tool support becomes mandatory for testing large-size statecharts.

Belli and Beyazit compared the event-based and state-based approaches for MBMT [67]. The event-based approach uses the event sequence graphs (ESG), whereas the state-based approach uses the finite state machine (FSM) [67]. The comparison criteria are mutation operators, coverage criterion, and test generation method. The mutation operators are sequence insertion, sequence omission, event insertion, and event omission for the ESG model. In contrast, transition insertion, transition omission, state insertion, and state omission are used for the FSM model. The coverage criterion is event pair coverage for ESG and transition coverage for FSM. However, the test generation method for specific coverage criteria roughly requires solving a well-known problem, namely the Chinese Postman Problem (CPP). They report that the FSM-based test sequences comprise more redundancy and cover 40 to 100 more failures. However, the cost becomes roughly 52 to 122 times higher. The ESG covers 29 to 50 fewer failures while it costs roughly 30 to 55 less due to event sequences clustering. Experiments conclude that the FSM-based test results are more effective for covering more failures because of the redundancy.

Belli et al. proposed an MBMT method providing novel mutation operators, namely omission and insertion operators, evaluated fault detection ability of the test set acquired using the mutated model and surveyed the literature on the MBMT [8]. They validate the effectiveness of three examples, which are industrial and commercial systems. Experiments show that the insertion operator is more efficient than the omission operator because it reveals more faults.

Kilincceker et al. proposed a hybrid MT approach that combines code-based mutation testing and MBMT to validate the hardware design [56]. They used code-based mutation for test execution. They selected the regular expression (RE) model for test generation due to its algebraic and declarative power. They also theoretically and experimentally proved that the proposed method satisfies the conditions of Goodenough and Gerhart's ideal testing [2].

To summarize the available studies in the scope of code and model-based mutation testing, this thesis presents a comparison table, Table

2.4.1, in which the mutation operators and effectiveness of each approach have been elaborated.

Table 2.4.1: Comparison of Mutation Testing Methods

	Method	Code and Model	Mutation Operator	Effectiveness
Code-Based Mutation	[9]	Fortran-like programs	Logical Expression replaces	Simple change
	[9]	VHDL programs	10 mutation operators	Not Specified
	[63]	Fortran Programs	22 mutation operators	Automation environment
	[65]	Java programs	40 operators	Not Specified
Model-Based Mutation	[66]	Statecharts	19 operators	Automation environment
	[67]	Event Sequence Graph (ESG)	3 operators	Comparison of models
	[8]	Event Sequence Graph (ESG)	3 operators	MBMT approach
Hybrid	[56]	Finite State machine (FSM)	Semantic mutants operators	MBIT approach

The Mutation testing concept has originally been applied to program codes. During the last decade, this concept has also been applied at the design level to models, leading to model-based mutation testing; Belli et al. [8], and Aichernig et al. [18] (See Table. 2.4.1). For a systematic utilization, appropriate mutation operators have been suggested, for example, for mutating finite-state machines Fabbri et al. [17] and event sequence graphs Belli et al. [8].

Fraser et al. provided a survey on testing based on model-checkers that is normally employed to find counterexamples violating checked properties [68]. The survey also presents details on mutation-based test generation that is first introduced by Ammann and Black [69] based on model-checkers.

Aichernig et al. introduced an automation tool for Model-based mutation testing (MBMT) [18]. The tool called MoMuT:UML (pronounced "MoMuT for UML") automatically generates test suites from the UML model of a system and then executes them on generated mutants to check their conformance relation.

Fellner et al. presented a novel test generation method for model-based mutation testing by using a model checking method [70]. They use hyper properties to generate test cases to address strong mutation analysis. Their approach utilizes a logic called HyperLTL for hyperproperties and offers a solution to generate test cases from non-deterministic models. The solution provided for non-determinism is not feasible in practice, thus, they changed the solution to the well-known method by converting a non-deterministic model to a deterministic one. The authors automate the mutant generation for Verilog models via the utilization of some mutant operators. Normally, the mutants are used to measure the mutation scores by executing generated test cases on these mutants. Even though the details are provided for the mutant generation, the authors do not provide enough detail for test execution that is automatically carried out by a model-based mutation testing tool called MoMuT.

Fellner et al. offered a test generation approach based on a heuristic-guided branching search algorithm for model-based mutation testing [71]. The approach employs asynchronous parallel processing to explore non-deterministic models. They integrate the approach to the MoMuT tool to cope with the huge state space of the models. The algorithm can achieve over 2,300 concurrent objects. They use 10 different industrial scale case studies defined in UML, Event-B, and a textual domain-specific language (DSL) modeling language. The MoMut tool can automatically convert these models to action systems. However, the mutation score for some of the case studies is low due to the random search algorithm utilized.

Prasetya and Klomp proposed a model-based testing approach to cope with non-deterministic models [72]. The non-determinism in the models causes the execution of multiple paths that lead to the impossibility of the calculation of test coverage. The authors employ a probabilistic approach to calculate aggregate coverage for non-deterministic cases produced by Labelled Transition System (LTS), Markov Decision Process (MDP), and Markov Chain. They model the system under test using LTS and then extended it to MDP that is a probabilistic version of the LTS. This work provides two experiments that are a non-deterministic model not linked to any system and a model of the backoff mechanism of the IEEE 802.11 WLAN protocol. Their approach suffers from a combinatoric explosion of the possible combinations of words for the usage of the coverage metrics proposed.

Besides using model-based testing for specific programs, there is a special type of MBT called checking experiment or finite state machine testing that requires special assumptions on the model. The assumptions need a specification (Mealy machine) required for them to be reduced, deterministic, and completely specified. Petrenko introduced an approach for this type of MBT that utilizes Mealy machines with symbolic inputs and outputs to remove these assumptions [73]. Petrenko also addressed detecting assignment/output faults and transition faults [73].

Mutation testing of hardware generally addresses design faults, but it is experimentally shown that they can also detect manufacturing faults; Nguyen et al. [61], Robach et al. [74]. Mutation testing is also used as a functional qualification system for HDL design verification; Hampton et al. [75], Rahkonen [15]. The proposed method by Hampton et al. [75] has been implemented and is available as a commercial tool called Certitude [1]. Certitude is used to measure the quality of a verification method using mutation testing.

In this thesis, the code-based mutation testing approach is adapted from DeMillo et al. [9], King and Offutt [63], Ma et al. [65], and Fabbri et al. [66] to obtain code-based mutants from the original program by using mutation operators. The authors by Fabbri et al. [66], Belli and Beyazit [67] and Belli et al. [8] offered model-based mutation testing that this thesis utilizes to construct model-based mutants from the

1 Certitude Tool, https://www.synopsys.com/verification/simulation/certitude.html

original (fault-free) model by using model mutation operators presented by Kilincceker and Belli [76] for the FSM model. It uses the similar idea proposed by Kilincceker et al. [56] as being a hybrid approach applying code-based and model-based mutation testing methods simultaneously.

2.5 IDEAL TESTING

Goodenough and Gerhart define the ideal test based on its principal conditions [2]. The theorem states that in the case of test data satisfying these conditions, namely reliability and validity, this test data enables testing in the absence of faults. They also provided proof of this fundamental theorem. Howden introduced a testing method for the analysis of paths, namely P-Testing, and evaluated the ideal test in terms of reliability condition [5]. The reliability of P-testing is checked against different types of common faults. However, Howden stated that P-testing is reliable or almost reliable for subset faults, not covering all faults. Bouge extended the ideal test's current conditions by offering additional features, namely bias and acceptability [77]. Bouge also detailed the relationship between program testing and program proving for bias and acceptability conditions. Langmaack presented sufficient and readable proof for compiler verification, considering the ideal test for verification and software testing [78]. The main inspiration of the thesis is based on the seminal work of Goodenough and Gerhart [2].

To briefly describe the *ideal test*, the following informal definitions given by Naik and Tripathy [3] are used. The formalization of an ideal test starts from the definition of a program as follows:

Definition 2.1. *Program* β *is a function that maps domain (D) to range (R).* $\beta : D \rightarrow R(\beta \subseteq DxR)$ *with a set P of properties that are main attributes of the program* β*, such as concurrency, sequence, specific computation, program state, timing behavior.*

The domain and range, including selected attributes of a program, can be limited to a model to eliminate irrelevant details such as timing and concurrency. Considering these details requires utilizing a model at higher expressive power that measures how well modeling can convey the details and relationships inherent in a particular problem, which is out of the scope of the thesis. This limited program can be represented by a model for which a formal definition is given as follows.

Definition 2.2. *Let* β *be a program as defined in Definition 2.1. Model* μ_{P_m} *is a function that maps* D_{P_m} *to* R_{P_m} *of the program* β *for a set* P_m *of specific properties.* $\mu_{P_m} : D_{P_m} \rightarrow R_{P_m}$*, where* $\mu_{P_m} \subseteq \beta$, $D_{P_m} \subseteq D$, $R_{P_m} \subseteq R$, $P_m \subseteq P$.

Model μ_{P_m} as being subset of P_m represents the same behavior as the program P_m. However, the behavior of μ_{P_m} is limited to its domain D_{P_m} to range R_{P_m} meaning that μ_{P_m} contains only some pairs of inputs and outputs. Therefore, the nonexistent pairs in P_m also do not exist in μ_{P_m}.

Definition 2.3. *Let β be a program as defined in Definition 2.1. A test $\beta(t)$ is a predicate, which is assigned to an execution of a program β resulting successful if it is passed or unsuccessful.*

The passing or failing values on a program β constitute a test case that is obtained from a model executed on the program. In execution, a test predicate is measured by checking whether the test output and execution output match or not.

Definition 2.4. *Test case (t) is a pair of input (i) and expected output (o). t = {(i, o)} where i $\in D_{P_m}$ and o $\in R_{P_m}$. For example, t_1 = { 0100, 110011}.*

Definition 2.5. *Let (t) be a test case as defined in Definition 2.4. The test suite (T) is a set of test cases (t)). For example, T = { $t_1, t_2, t_3 \dots$ }.*

Definition 2.6. *Test sequence (t_s) is an ordering of test cases, which starts and ends with a specific test case. For example, $t_s = (t_3 t_1 t_2 t_5 t_4\ t_5 t_6)$.*

The test suite may cover the entire input and output domain of the model, as using exhaustive testing, which results in excessive test effort. To decrease this effort in a more practical way, a test suite can be selected from the entire domain by using some criteria. In this way, more efficient test sequences satisfying test selection criteria are collected to assess test adequacy.

Definition 2.7. *Let (t) be a test case as defined in Definition 2.4. Test selection criterion (σ) is a rule to choose specific test cases for certain reasons, such as fault detection. Satisfy(t, σ) is a predicate to define fulfillment of t with respect to σ. Satisfy(t, σ) = true iff t satisfies σ, otherwise, Satisfy(t, σ) = false.*

Definitions 2.1-2.7 are to support and clarify the background of an ideal test. Test predicate β(t) is either successful or unsuccessful depending on execution results. However, it is necessary to define another predicate to check its acceptability, as follows:

Definition 2.8. *Let $\beta(t)$ be a test as defined in Definition 2.3. OK(d) is a predicate referring to the acceptability of the result of β(t). OK(d) = true iff β(t) is an acceptable output o. OK(d) = false iff β(t) is an unacceptable output o.*

The predicate OK(d) checks the acceptability of only a test case in a test suite. If all test cases in a test suite T are acceptable, T becomes successful by means of the following definition.

Definition 2.9. *Successful(T) is a predicate that defines the success of a set of* $t_s \in T$*. T is a successful test iff* $\forall\ t_s \in T \mid OK(t_s)$*. Successful (T) = true iff* $\forall\ t \in T \mid OK(t)$*. Otherwise, T is an unsuccessful test iff* $\forall\ t \in T \mid \neg OK(t_s)$*. Thus, Fail(T) = true iff* $\forall\ t \in T \mid \neg OK(t)$*.*

The predicates OK(d), Successful(T), Fail(T), Satisfy(t, σ) are used to define reliable and valid criteria to achieve an *ideal test*.

Definition 2.10. *Reliable Criterion is related to the consistency of a selected test suite with respect to σ, represented by Reliable(σ).* $\forall\ \sigma \mid$ *Reliable(σ) iff* $\forall\ t_s \in T \wedge$ *Satisfy*$(t_s, \sigma) \mid$ *Successful(T)* $\vee$ *Fail(T).*

Definition 2.11. *Valid Criterion is related to the ability of a test suite to produce meaningful results, represented by Valid(σ).* $\forall\ \sigma \mid$ *Valid(σ) iff* $\exists\ t \in T \wedge$ *Satisfy(t, σ)* $\mid \neg OK(t)$ *then* $\forall t \in T \wedge$ *Satisfy(t, σ)* $\mid \neg OK(t)$*.*

The following definition presents an *ideal test* concerning the reliability and validity of test selection criteria.

Definition 2.12. *A test β(t) is an ideal test iff* $\forall\ t_s \in T \wedge$ *Satisfy*$(t_s, \sigma) \mid$ *Reliable(σ)* $\wedge$ *Valid(σ).*

CHAPTER 3

BACKGROUND

In this chapter, the terminology used in this thesis, the coverage criteria and an analysis method are presented. The notations and models used in the thesis, including Finite State Machine (FSM), Regular Expression (RE), Extended Regular Expression (ERE), Syntax Tree (ST), and Contextual RE (CRE) are elaborated in Section 3.1. The coverage criteria for the defined models are presented in Section 3.2. Lastly, the regular expression analysis method, PQ-Analysis, is also presented with complementary algorithms and their complexity analysis.

3.1 USED NOTIONS

The FSM models behavioral level HDL and graphical user interface programs as considered sequential systems. The following formal definition is presented for the FSM notion, which will follow other notions.

Definition 3.1. *Finite State Machine: An FSM [79] is defined by 5-tuples* $\langle$ *Q,* $\sum$*,* δ*,* q_0*, F* $\rangle$ *where the tuples are;*

- *Q: A finite set of states;*
- $\sum$*: A finite set of input symbols (alphabet);*
- δ*: A state transition function, mostly represented by a table;*
- q_0*: An initial (starting) state is an element of Q;*
- *F: A finite set of final states is a subset of Q.*

Example 3.1. *An example FSM M is given by ({so,s1},{a,b,c,d},δ,{so},{s1}), where δ = { δ(so, a) = so, δ(so,b) = s1, δ(s1, c) = s1, δ(s1, d) = so } is a transition function. Figure 3.1.1 represents the M graphically.*

The following chapters will use the RE model converted from the FSM model to generate test suites. The formal definition of the RE model is below.

Definition 3.2. *Regular Expression: An RE [79] contains the symbols (a,b,c, ...) connected by operators that are defined as follows;*

- *Sequence (usually no operator to represent), for example, "ab" refers to "b" follows "a",*

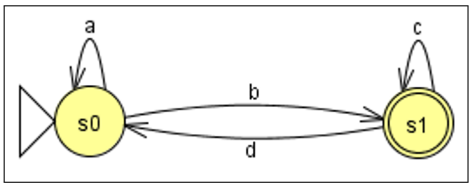


Figure 3.1.1: An example FSM M

- *Selection (represented by "+" operator), for example, "a+b" refers to "a" or "b",*
- *Iteration (mostly represented by "*" (star) operator), for example,* "a^*" *refers to "a" repeats zero or more times.. Also,* "a^+" *refers to one or more times.*

Example 3.2. *An example of a regular expression is given below;*

$$[(ab(c+d)^*)] \tag{1}$$

The example RE (in (1)) means symbol "a" is followed by symbol "b" that is followed by zero or more iterations of symbol "c" or "d".

The ERE model will also be used to generate test suites as one of the model-based test generation notions. The following definition presents it formally.

Definition 3.3. *Extended Regular Expression (ERE) [80]: The regular expression given above is extended using the following range operator applied to Kleene's Star operator*

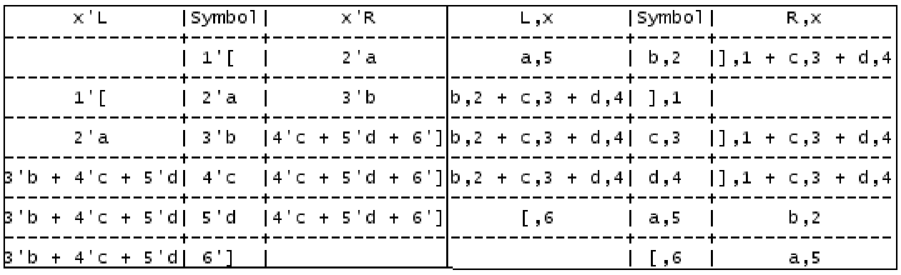
Range - is represented by "n~m" instead of "*" operator. For example, "$a^{n\sim m}$" means that "a" can occur a minimum of "n" and a maximum of (m-n+1) times.

Example 3.3. *The regular expression given in Example 3.2 can be extended as follows:*

$$[(ab(c+d)^{1\sim 2})] \tag{2}$$

The CRE model is obtained after the regular expression analysis that will be mentioned in Section 3.3 and will be used to generate test suites in Section 6.1. It can be defined as follows:

Definition 3.4. *Contextual Regular Expression (CRE) [81]: It is a tabular representation of a regular expression and contains backward and forward contextual information of it, which is obtained after contextual analysis (see Section 3.3 for details) of a RE. Figure 3.1.2 shows the ERE of Example 3.3.*



x'L	Symbol	x'R
	1'[	2'a
1'[	2'a	3'b
2'a	3'b	4'c + 5'd + 6']
3'b + 4'c + 5'd	4'c	4'c + 5'd + 6']
3'b + 4'c + 5'd	5'd	4'c + 5'd + 6']
3'b + 4'c + 5'd	6']	

L,x	Symbol	R,x
a,5	b,2	],1 + c,3 + d,4
b,2 + c,3 + d,4	],1	
b,2 + c,3 + d,4	c,3	],1 + c,3 + d,4
b,2 + c,3 + d,4	d,4	],1 + c,3 + d,4
[,6	a,5	b,2
	[,6	a,5

Figure 3.1.2: Contextual Regular Expression

The syntax tree will be used to represent a RE model and to generate test suites using the tree traversal algorithm, which will be introduced in the following chapter.

Definition 3.5. *Syntax Tree (ST): A syntax tree is basically a tree to represent a regular expression. It can be either right-to-left or left-to-right associative. Root and internal nodes store operators, and leaves store symbols of the regular expression. The ST representation, left-to-right associative, of RE in Example 3.2 is shown in Figure 3.1.3.*

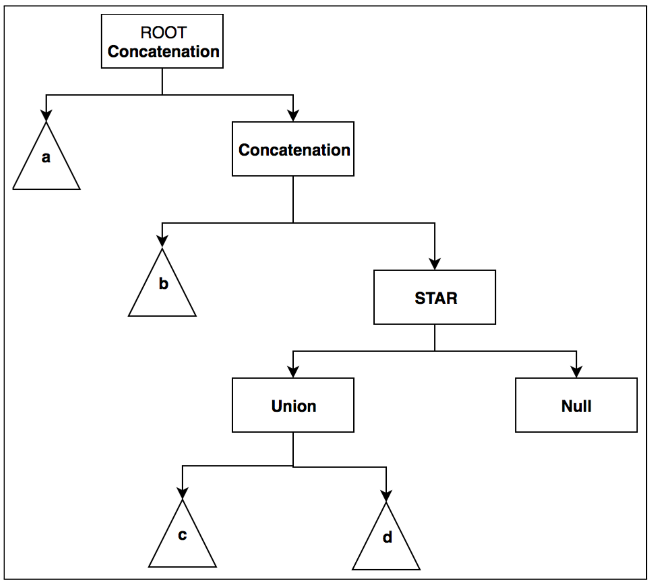


Figure 3.1.3: The ST Representation, Left-to-right Associative, of RE in the Example

Using the approach proposed in this study, the set of test sequences generated from the ST model constitutes a test suite. Depending on the selection of consecutive Kleene star operators, the length of any test sequence can be infinite. A solution requires using the range operator based on the ERE model to tackle the problem, and the range operator can be defined using any specific values.

3.2 COVERAGE CRITERIA

There are several coverage criteria to assess the adequacy of the test. They define how the system is tested thoroughly and whether the generated test sequences are good enough. From a software testing perspective, "thorough", "good enough", and "adequate" refer to the same meaning [82]. Coverage criteria are used to achieve these quality measurements. A test set is adequate when it satisfies particular coverage criteria.

The bit coverage defined for HDL program validation at the behavioral level is used in this thesis.

Definition 3.6. *Bit coverage [13]: It is satisfied if, for each bit of a variable, signal, and port in the behavioral design, there is at least a test sequence that exercises bit stuck-at 0/1 assignments of these bits.*

The alphabet and operator coverage criteria are defined for test sequence generation based on RE in the literature [83].

Definition 3.7. *Alphabet coverage [83]: It is satisfied if, for each symbol 'a' in the alphabet, there is at least a test sequence, including symbol 'a', in the test suite. For instance, the test suite abcd satisfies this criterion for the RE given in Example 3.2*

Definition 3.8. *Operator coverage [83]: It is satisfied if, for each union operator, the test suite includes a test case containing the first operand and another test case containing the second operand of the union operator.*

For each Kleene's star operator, the test suite includes a test sequence containing no iteration, precisely one iteration, and more than one iteration of the operand of Kleen's star operator. For instance, the test suite ab, abc, abd, abcc, abcd, abdd satisfies this criterion for the RE given in Example 3.2

Definition 3.9. *Context Coverage Criterium: It is defined as the generated test suite that includes all cases by the Context Table. This is formally;*

$$\forall t_n \in T \Rightarrow t_n = c \in C_i^j \tag{3}$$

Definition 3.10. *Left Context Coverage Criterium: It includes all states of the generated test suite by the left indexed symbols of the Context Table. This is formally;*

$$\forall t_n \in T \Rightarrow t_n = c \in C_i^j \|_{left} \tag{4}$$

The Right Context Coverage Criterium is defined similarly to the Left Context Coverage Criterium.

Definition 3.11. *Forward Context Coverage Criterium: It includes all states of the generated test suite by the Forward Context Table. Thus, formally,*

$$\forall t_n \in T \Rightarrow t_n = c \in C^j \tag{5}$$

The Backward Context Coverage Criterium is defined similarly to the Forward Context Coverage Criterium.

Definition 3.12. *Compatibility Coverage Criterium: It is defined as the full compliance of all cases of the produced test suite with the indices given in the Compatibility Table. This is formally;*

$$\forall t_n \in T \Rightarrow t_n = u \in U \tag{6}$$

The coverage criteria are determined by the tables obtained as a result of the analysis of a RE. As mentioned, state information is lost in conversions from FSM to RE, and the indexing process recreates the lost state information. In this context, with the coverage criteria put forward, state coverage and transition coverage are frequently used in the literature.

The coverage criteria can sometimes be confused with metrics for evaluating the performance of test generation algorithms. They are used to measure the adequacy of the test. For evaluation performance and comparisons between test generation algorithms, the commonly used metrics are test suite size, test generation time, test execution time for efficiency, fault coverage, and mutation score for effectiveness. The test suite size is about the length of the generated test set, and the test generation time is a measurement of the total time for test generation, similar to the test execution time. On the other hand, the fault coverage is the ratio between the number of detected faults and the total number of faults injected in the experiment. The mutation score is the ratio between the number of killed mutants over the total number of mutants, as mentioned in Section 2.4.

3.3 FINITE STATE MACHINE AND REGULAR EXPRESSION ANALYSIS

The regular expression analysis is carried out by PQ-Analysis method and it is proposed by Eggers and Belli [81, 84] indexes the provided RE to obtain missing state information during conversion from the FSM. Reader can refer to Section 3.1 for definitions of RE and FSM.

Moreover, the context of the RE elements can cause ambiguity due to the same symbol appearing in different positions, which is copied after PQ-Analysis using indexing of the symbols. The primary purpose is to extract information regarding the analyzed system's fault tolerance capability using indexing and context tables (CTs). In the thesis, it is utilized to increase the ability of test sequences acquired from CTs. The PQ-Analysis is adopted as a base of test generation approach using tables resulting from the PQ-Analysis. Test generation from these tables results in more efficient test suites than from the others based on different models, such as FSM, due to the usage of redundancy provided by the PQ-Analysis.

Eggers and Belli [81, 84] propose an approach to detecting, localizing, and correcting faults based on regular expression, originally to be deployed in compiler construction. Belli extended this approach to a general theory of fault tolerance and its applications to sequential systems, for example, faults in system-user interactions [7] and hardware [85]. The idea is to analyze the checking self-detection and self-correction capabilities of a system under test. If this system does not possess these capabilities, the approach suggests extending the system by inserting a minimized amount of functional redundancy [81].

The approach is syntax-based and assumes that the system under test is strictly sequential and, thus, can be specified by a regular expression (RE). The symbols of this RE can be interpreted differently, for example, as events, such as user inputs and system outputs. A set of hypotheses is defined for insertion (I), replacing (R), and deletion (D) of the symbol(s) for context-based handling of faults, for example, "a symbol between two symbols is to be inserted/deleted/replaced to correct a fault detected". For an unambiguous self-correction of a fault, the pairwise, mutual exclusion of the application of hypotheses is necessary; that is, two hypotheses, P and Q out of I,R,D are supposed not simultaneously to be applied to a fault. In the course of several master's and Ph.D. theses on this subject, the approach has been coined "PQ-Analysis" by the students.

This thesis uses the PQ-Analysis as the basis of the test generation approach. Test sequences generated from the context table (see Figure 3.1.1) resulted from PQ-Analysis in the current methodology.

A given regular expression T can be expanded to generate sequences (strings or words) of symbols that build up a regular (type 3) language L(T). The core idea of PQ-Analysis stems from extracting the context relations of the symbols from T, storing them in lookup tables, and then using these tables to check whether a word belongs to the language L(T) or not.

The approach comprises the seven steps that are depicted in Figure 3.3.1. The details of these steps are given below.

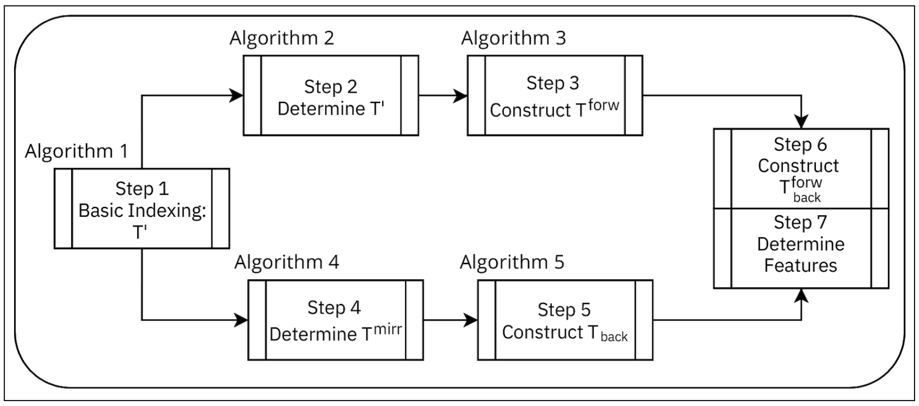


Figure 3.3.1: Steps of the PQ-Analysis and the Corresponding Algorithms

- Step 1: Index the given RE T for tracing the contextual positions of symbols leading to the indexed T′.
- Step 2: Transfer T′ to an equivalent FSM E^{forw} [86, 87].
- Step 3 (forward indexing): Scan T through E^{forw}, that is, input the symbols of T to E^{forw}, note (trace) the transferred states as superscripts on T.
- Step 4: Reverse (mirror) T′ leading to T^{mirr} and transfer T^{mirr} to an equivalent FSM E^{back}.
- Step 5: Scan T^{mirr} through E^{back}, that is, input the symbols of T^{mirr} to E^{back}, note (trace) the transferred states as subscripts on T^{mirr}.
- Step 6 (backward indexing): Reverse (mirror) T^{mirr}_{forw} and subscribe to the indices and construct of T^{forw}_{back} (Coding) by combining T^{forw} and T_{back}.
- Step 7 (coding): Simultaneously forward and backward indexing of T to determine its characteristic relations (tables).

To make the concept clear, the above-mentioned steps are demonstrated in a simple example. Note that State Transition Table of E^{forw} and State Transition Diagram of E^{forw} in Figure 3.3.2 and Compatibility and Context Tables in Figure 3.3.3 are borrowed from Belli [81].

Example 3.4. *Given the following regular expression:*

$$T = [(ba(b+c^*)^*(a+b)^*] \tag{7}$$

The related PQ-Analysis tables can be provided by following Steps 1-7.

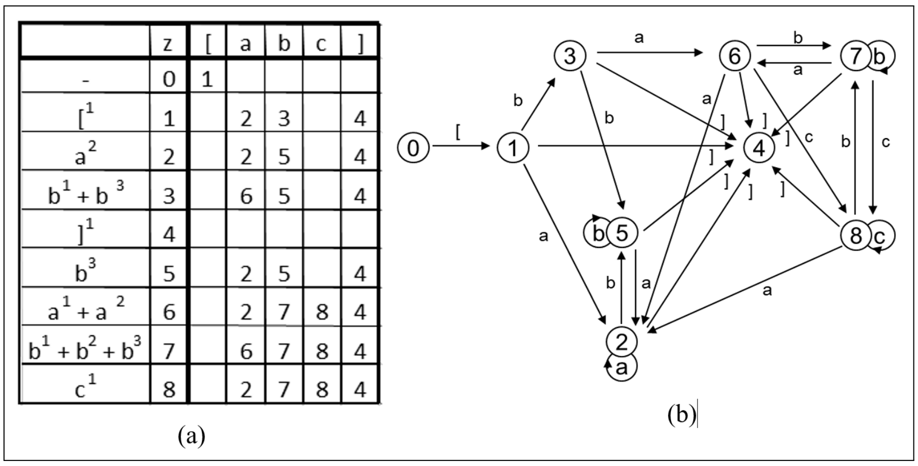
Step 1: Index the given T:

$$T' = [^1(b^1a^1(b^2+c^1)^*)^*(a^2+b^3)^*]^1 \tag{8}$$

Step 2: Transfer T′ to an equivalent FSM E^{forw} Determination of corresponding states of an equivalent automaton, notated by

(a)i=:j (input a transfers the state i into state j) initial state = 0, ([)0 = ($[^1$)0 =: 1 ≡ $[^1$, (a)1 = (a^2)1 =: 2 ≡ aa^2, (b)1 = (b^1 + b^3)1 =: 3 ≡ b^1 + b^3, (])1 = ($]^1$)1 =: 4 ≡ $]^1$, (a)2 = (a^2)2 =: 2 ≡ a^2, (b)2 = (b^3)2 =: 5 ≡ b^3, (])1 = ($]^1$)2 =: 4 ≡ $]^1$, (a)3 = ... etc. Final State = 4.

State Transition Table of E^{forw} and State Transition Diagram of E^{forw} is given in Figure 3.3.2.



	z	[	a	b	c	]
-	0	1				
$[^1$	1		2	3		4
a^2	2		2	5		4
b^1+b^3	3		6	5		4
$]^1$	4					
b^3	5		2	5		4
a^1+a^2	6		2	7	8	4
$b^1+b^2+b^3$	7		6	7	8	4
c^1	8		2	7	8	4

(a)

Figure 3.3.2: (a) State Transition Table and (b) Diagram for Eforw [81]

Step 3: Scan T through E^{forw} - Forwards scanning the expression:

$$T^{forw} = [^1(b^{3+7}a^6(b^7+c^8)^*)^*(a^{2+6}+b^{3+5+7})^*]^4 \tag{9}$$

Step 4: Reverse (mirror) The "mirrored" (reverse) expression

$$T^{mirr} =]^1(a^2+b^3)^*((c^1+b^2)^*a^1b^1)^*[^1 \tag{10}$$

Construction of the corresponding automata E_{back} in analogy to constructing of E^{forw}.

Step 5: Scan T^{mirr} through E_{back} backwards scanning of the expression E_{back}

$$T^{mirr+forw} =]^1(a^2+b^3)^*((c^4+b^{3+7})^*a^{2+6}b^{3+8})^*[^5 \tag{11}$$

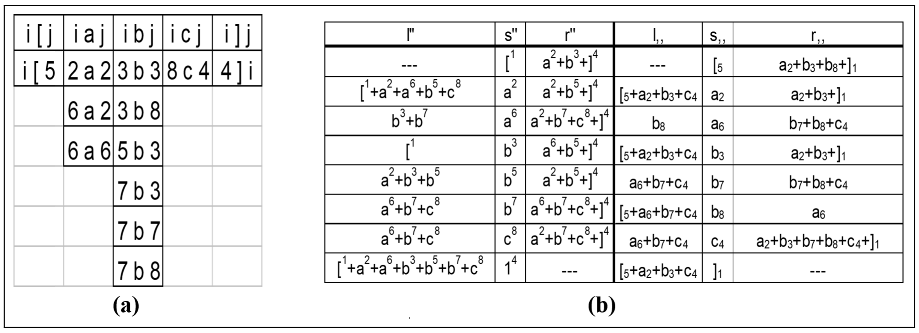
Step 6: Reverse (mirror) $T^{mirr+forw}$ repeated reversing ("Mirroring") of the expression, lower indexing

$$T^{mirr+forw+mirr} = [_5(b_{3+8}a_{2+6}(b_{3+7}+c_4)^*)^*(a_2+b_3)^*]_1 =: T_{back} \tag{12}$$

Coding, that is, concurrent forward and backward scanning

$$T^{forw}_{back} = [^1_5(b^{3+7}_{3+8}a^6_{2+6}(b^7_{3+7}+c^8_4)^*)^*(a^{2+6}_2+b^{3+5+7}_3)^*]^4_1 \tag{13}$$

Step 7: Determine the characteristic relations (tables) of T given in this section with Figure 3.3.3.



i[j	iaj	ibj	icj	i]j
i[5	2a2	3b3	8c4	4]i
	6a2	3b8		
	6a6	5b3		
		7b3		
		7b7		
		7b8		

(a)

l''	s''	r''	l,,	s,,	r,,
---	$[^1$	$a^2+b^3+]^4$	---	$[_5$	$a_2+b_3+b_8+]_1$
$[^1+a^2+a^6+b^5+c^8$	a^2	$a^2+b^5+]^4$	$[_5+a_2+b_3+c_4$	a_2	$a_2+b_3+]_1$
b^3+b^7	a^6	$a^2+b^7+c^8+]^4$	b_8	a_6	$b_7+b_8+c_4$
$[^1$	b^3	$a^6+b^5+]^4$	$[_5+a_2+b_3+c_4$	b_3	$a_2+b_3+]_1$
$a^2+b^3+b^5$	b^5	$a^2+b^5+]^4$	$a_6+b_7+c_4$	b_7	$b_7+b_8+c_4$
$a^6+b^7+c^8$	b^7	$a^6+b^7+c^8+]^4$	$[_5+a_6+b_7+c_4$	b_8	a_6
$a^6+b^7+c^8$	c^8	$a^2+b^7+c^8+]^4$	$a_6+b_7+c_4$	c_4	$a_2+b_3+b_7+b_8+c_4+]_1$
$[^1+a^2+a^6+b^3+b^5+b^7+c^8$	1^4	---	$[_5+a_2+b_3+c_4$	$]_1$	---

(b)

Figure 3.3.3: (a) Compatibility and (b) Context Tables [81]

The context table contains two sub-tables (Figure 3.3.3-(b)). The left side is for forward indexing, and the right side is for backward indexing (see Step 3 and Step 6, respectively). For each sub-table (forward and backward), the right context of each symbol (s) is given on the right column (function r), and the left context is on the left column (function l).

Another important table is the compatibility table (Figure 3.3.3-(a)). It consists of all pairs (i, j) of a forward index i and a backward index j existing in a coded symbol s^i_j of T^{forw}_{back} given in (13). This is described by the notation C^i_j of s^i_j. The latter means that states i and j are compatible via the symbol s.

The context table will be used for test generation in Chapter 5, and later, it will also be used together with the compatibility table for extending the sequential system, that is, determining the redundancy for achieving fault tolerance in Chapter 8.

Algorithms

The algorithms mentioned in Figure 3.3.1 are given below with their pseudo-codes. Also, their computational complexity analysis is provided.

Listing 1: Algorithm 1: Indexing the T

```
Input: T, an RE
Output: T′, a basic indexed RE
T′ := T
For each sym in T′: symcount [sym] = 0 \\symcount is symbol count
For each sym in T′
        currsym = sym \\currsym is current symbol
        if (symcount[currsym] = 0) symcount[currsym] = 1
        else symcount[currsym] = symcount [currsym)] + 1
        currsym in T′ := $currsym^{symcount(currsym)}$
        sym := currsym
End for
```

Algorithm 1, shown in Listing 1, runs in |T(sym)| time, which equals to the number of symbols in T because the "for" loop in the algorithm processes the 'number of the symbol' times. Therefore, the complexity of Algorithm 1 is linear, solely depending on the number of symbols in the RE T.

Listing 2: Algorithm 2: States and transitions determination of FSM E^{forw} that accepts T′

```
Input: T′, basic indexed T
Output: $ST_i$(sym), state transition table of T′; i is a state (row) and
    sym is a corresponding column in the table
currsym = $[^1$ in T′
1: = currsym
currstate = 1 \\currstate is current state
For each sym in T′
$F_{currstate}$(sym) = the set of sequent symbols after sym
        If $F_{currstate}(sym) \neq \emptyset$
                For each sym in $F_{currstate}$
                        new_state = $F_{currstate}$ (sym)
                        $ST_{currsym}$(sym)=new_state
                End for
        End if
End for
```

Listing 3: Algorithm 3: Constructing T^{forw}

```
Input: T′ and $ST_i$(sym)
Output: $T^{forw}$, forward indexed T
A = sym set of ST
For each sym in T′
        if sym ∈ A for each set of i(I)
                sym in $T^{forw}$ = $sym^I$
        End if
End for
```

The complexity of Algorithm 2, shown in Listing 2, equals to square power of the complexity of Algorithm 1 because the number of symbols in T and T′ are equal. Hence, algorithm 2 runs in quadratic time in the worst-case scenario depending on the number of symbols in RE T′.

Listing 4: Algorithm 4: States and transitions determination of FSM E_{back} that accepts T^{mirr}

```
Input: $T^{mirr}$ , basic backward indexed T
Output: $ST_i^{mirr}$(sym), state transition table of $T^{mirr}$; i is state (row)
    and sym is corresponding column in table
```

In the worst case, the complexity of Algorithm 3, shown in Listing 3, is $|(|I^{'}|.|T^{'}$ (sym)$|)|$ where I′ is the set of i for each symbol representing the indices of the symbols. The sum of the cardinality of the set i equals to I′ and it increases in a linear manner depending on the indexes.

Algorithm 4, shown in Listing 4, is the same as Algorithm 2, having the same complexity and it is $|\ [|\ T^{mirr}(sym)\ |]^2\ |$

The complexity of Algorithm 5, shown in Listing 5, is where $|\ |I^{mirr}|$ $\times$ $|T^{mirr}$(sym)$|\ |$ where I^{mirr} is the family of set i for each symbol representing the indexes of the symbols of T^{mirr}. The total complexity of the PQ-Analysis for Algorithms 1 to 5 is of quadratic order. The context table is used to generate test sequences from the given RE for negative and positive testing.

Listing 5: Algorithm 5: Constructing T^{back}

```
Input: $T^{mirr}$ and $ST_i^{mirr}$(sym)
Output: $T^{back}$,backward indexed T
B = sym set of $ST_i^{mirr}$(sym)
For each sym in $T^{mirr}$
    if sym $\in$ B for each set of i(I)
        sym in $T^{back}$ = $sym^I$
    End if
End for
```

Part II

APPROACHES

CHAPTER 4

TEST GENERATION BASED ON REGULAR EXPRESSION APPLIED TO HDL PROGRAMS

This chapter presents an approach for test generation based on an HDL program's regular expression (RE). The following sections present motivation and the proposed approach for regular expression.

4.1 MOTIVATION FOR HDL TESTING

According to the well-known Moore's Law, the number of transistors or components on a Very Large-Scale Integration (VLSI) chip doubles approximately every 18 months [88]. Therefore, the testing and validation of hardware become increasingly critical and overwhelming. As a result, the emergence of new methods to handle this problem efficiently becomes a necessity.

The validation of hardware in the early stages of the design flow not only reduces the cost, due to low complexity considering the abstraction level but also provides reusability of test sequences at lower design levels. Generally, test sequences or test patterns are generated for two purposes: testing to target design faults and structural testing to address manufacturing faults. The first one can be applied at the early stage of the design flow, that is, behavioral design, and the generated test sequences can be reused at lower levels, for example, register transfer level or gate level. The second one, structural testing, is applied at the gate level and requires extensive analysis. Thus, it demands a high cost for test pattern generation. On the other hand, structural testing provides appropriate and effective coverage of targeted manufacturing faults that are commonly stuck-at 0/1. In these faults, signals or pins are assumed to be stuck at logical '1' or '0'. It is reported by Lajolo et al. that there is a direct correlation between design faults at the behavioral level and manufacturing faults at the gate level [27]. Therefore, the test sequence generation approach discussed in this study can be useful for targeting design and manufacturing faults.

A method for test sequence generation is proposed to validate a given HDL program (as a Finite State Machine (FSM)) and target design faults at the behavioral level. The FSM model is automatically extracted from the HDL program by scanning the code to find the state and transition patterns. Then, this FSM is converted into a RE, which offers more abstraction, compactness, and conformity to algebraic operations based on Kleene Algebra [89]. This RE is represented by a Syntax Tree (ST). Operators and symbols of RE are represented in the external and internal nodes of the ST, respectively and, note that symbols are interpreted as events in this thesis. As a result, the procedure of the test sequence generation is carried on by traversing this ST.

4.2 THE APPROACH BASED ON REGULAR EXPRESSION

The proposed approach includes the following steps: extraction of FSM from the given HDL program, conversion of FSM to RE, ST construction from RE, and traversal of ST to generate test sequences satisfying the alphabet and operator coverage criteria. Figure 4.2.1 depicts these steps.

An HDL program is analyzed to generate an FSM in the FSM extraction step. The HDL programs, which satisfy specific patterns, can be considered. The patterns relate to implementing the HDL program where states and transitions are defined to construct the FSM model. The patterns represent these states, transitions, and their relations. A relation between states represents a transition that is triggered by an event that labels this transition. The FSM Extraction program reads the HDL program to determine states and transitions.

In the FSM to RE conversion step, the well-known Brzozowski algorithm [90] is used, which requires constructing an equation system from the state transition table of FSM. This equation system defines the transition relations of states. The system is represented by a square matrix on which an elimination algorithm, for example, Gaussian elimination algorithm, is run to generate the RE. Arden's rule [91] is applied to this equation system to eliminate redundant transitions and shorten the system. The application of this elimination continues until one column and one row for the equation are obtained, which represents the expected RE.

Based on the proposed approach, the FSM to RE conversion can be done using PQ-Analysis tool [1]. However, this conversion may result in a longer RE. Therefore, the JFLAP tool [92] can be used to shorten this RE as depicted in Figure 4.2.1. This approach provides a compacted RE model to reduce the computation in the further steps.

Note that in some systems, the HDL program is not already available. So, the model extraction step of the proposed approach is not useful in these cases. However, the hardware specification of the system, such as state machine diagram, activity diagram, and sequence diagram, may be available and provide the behavior model for the system. As these specifications imply dynamics of the system, the

1 PQ-Analysis tool, https://github.com/kilincceker/MBIT4HW

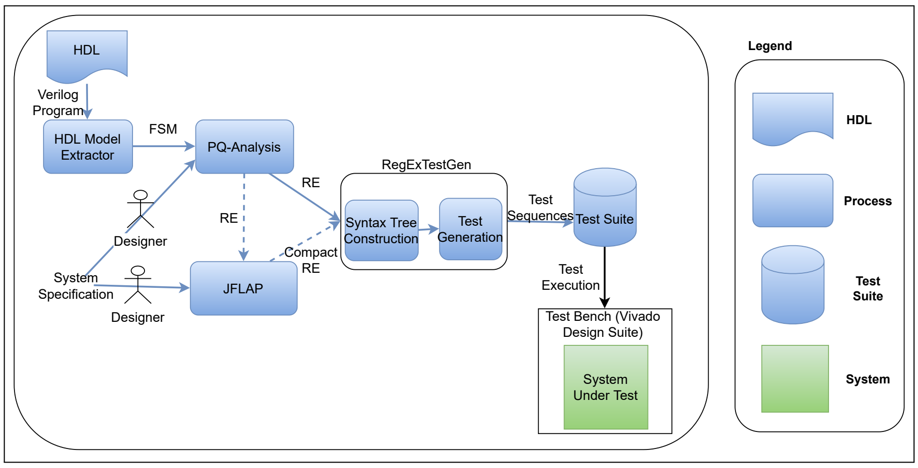


Figure 4.2.1: General Overview For The Approach Based on RE

designer can create the FSM for the system using tools such as PQ-Analysis or JFLAP [92].

As a result, the proposed approach supports both creating the FSM from the system specification and extracting it from the given HDL program.

For the ST construction, a tool called dk.brics.automaton [2] is used. This tool provides an open-source implementation of an antichain algorithm called forward subset. In this study, this tool has been extended in a way that it is to get rid of infinite sequences generated by Kleene's Star operator.

Based on the proposed approach, the test sequence generation from RE requires traversing the ST. For example, the following test sequences can be generated from the given RE in Example 3.2 in Section 3.1 by traversing the corresponding ST.

$$ab, abc, abd, abcc, abdd, abcd, ... \tag{14}$$

According to Kleene's Star operator, the length of the test sequences can be infinite (see (15)). This may cause the generation of infinite sequences, and it can be resolved by the utilization of the ERE model. For example, the ERE given in Example 3.3 in Section 3.1 can be used for (14). In this example, the iteration of Kleene's star is bounded to 1-2 by utilizing a range operator. Therefore, the following test sequences can be generated;

$$ab, abc, abd, abcc, abdd, abcd \tag{15}$$

The algorithm given in Listing 6 defines the procedure of test sequence generation in this section. The input of the algorithm is the

2 dk.brics.automaton toolchain, http://www.brics.dk/automaton/

ST of the RE model. The output is the resulting set of test sequences by traversing the ST.

Listing 6: Test sequence generation based on ST of RE Input: S S is syntax tree of given RE Output: ti ∈ T, i=1,. . . ,n T is the resulting test suite

```
node = ST.root
Set RegExTestGeneration (SyntaxTree node) {
CASE (node.data) OF
      (symbol) return {node.data}
      (union) return RegExTestGeneration(node.left) U
                              RegExTestGeneration (node.right)
      (Star)  return RegExTestGeneration (node.left) U Epsilon
      (Plus)       return RegExTestGeneration (node.left)
      (Concatenation) return Concatenation(
                                       RegExTestGeneration(node.left),
                              RegExTestGeneration (node.right))
      ('n~m') {Set s  = RegExTestGeneration (node.left)
                                        for  (i=0 to  n)
                                             Set  p = Concatenation (p, s)
                                             return p}
EndCASE
}
Set Concatenation (Set s1, Set s2){
Set result  = empty
for       each item1 in s1
            for each item2 in s2
                        result  = result  U item1.item2
}
```

This algorithm starts with the root node of the ST and traverses the left and right nodes based on the given procedure. The following values can be encountered in each node during traversal of the ST;

- A Symbol: The symbol is added to the current test sequence because it is a leaf node.
- A Disjunction (Union) operator: The test sequences from the left and right nodes of the union operator are combined and returned as a result.
- A Kleene Star operator: As is already discussed, 0 and 1 iterations are considered for star operator to satisfy the operator and alphabet coverage criteria. Therefore, test sequences including empty words and the string set from the left node of the current node are combined as the result of this operator.
- A Kleene Plus operator: The same procedure of the star operator is applied for the plus operator, excluding empty words, as it only contains 1 iteration.

- A Concatenation operator: This operator concatenates each string of the left node with each string of the right node.
- A Range operator: In this operator, the lower bound of the rage is considered to cover the criteria, and extra iterations are avoided. Therefore, the concatenation of the operand for n times is the result of this operator.

After executing one of the above-mentioned conditions for each node, based on the node's data, the traversal continues with the predecessor operator node using the test sequence set already generated.

A tool called RETestGen is developed by implementing the proposed test generation algorithm. The RETestGen takes the RE and constructs the ST from which test sequences are generated. This tool will be utilized in case studies for HDL in Section 7.1, and results in comparison with other similar tools will be presented in Section 7.3.1.

CHAPTER 5

TEST GENERATION BASED ON CONTEXTUAL REGULAR EXPRESSION APPLIED GRAPHICAL USER INTERFACE TESTING

This chapter introduces a method for test generation based on the contextual regular expression (CRE) for a GUI program. The following sections present the motivation and the proposed approach based on contextual regular expression.

5.1 MOTIVATION FOR GRAPHICAL USER INTERFACE TESTING

It is very important to catch unnoticed bugs and flaws in GUI and the project before their designs go into production. GUI testing is the process of testing the visual elements of an application, and it's designed to prevent the problems mentioned above before any user can perceive them. Those elements that influence the attractiveness can be color, font, size, etc., of the visual elements on the screen, and business policy can be checked with the help of automated UI testing. Any undesirable event, such as a design flaw, that will occur during the automated GUI testing will reveal the problems at the core of the application. Manual GUI testing is a cumbersome process of checking the application before it goes to the market. This process tends to be sloppy because of the difficulties of manual testing.

New techniques and processes have emerged to improve testing aspects of the software development life cycle. GUI testing is no longer considered a job to be done by a tester manually. Nowadays, software projects are becoming larger and larger, and thus, the required labor for testing every part of their GUI by hand increases excessively so that it almost becomes infeasible.

Random tests are, in general, not an effective way to validate products. However, random testing plays a crucial role in testing activity [93] due to its simplicity, which explains its popularity to be used as a yardstick to compare a novel, suggested method with existing testing techniques and to evaluate its efficiency.

An approach to random generation of test sequences for GUI testing is proposed to address sequencing and functional faults in the following section. In case of sequencing faults, GUI cannot reach the final event, which might cause a system crash. In case of functional faults, the GUI cannot provide the desired functionality even if it reaches the final event.

Modeling the GUI by a finite state machine is suggested, and it is automatically converted to a regular expression (RE) using a tool. The RE has the same expressive power as the corresponding FSM as both can be represented by a type-3 grammar, generating the same regular language. The tester can also model the GUI directly with a RE if he/she is familiar with working with RE.

There are several reasons why working with RE is preferred to working with FSM. First, RE form algebra (event algebra, see [89]) allows algebraic operations that are considerably easier and more efficient to handle than graph-based operations on FSM. Secondly, a RE model is mostly less spacious than a graph model. Last but not least, analyzing contextual relations can be carried out more easily and efficiently by RE than with graph models.

Therefore, a tool to make up the missing context information, such as the position of a symbol within its neighbors that cannot directly be determined in the original model, that is, FSM, will analyze the RE. Reader can refer to Section 3.1 for the details of the contextual regular expression as a result of this analysis and represented by the context table. As a next step, the context table representing will be traversed to generate the test sequences. To do this, a symbol in the table is repeatedly selected, starting from the initial symbol, in a random manner until reaching a special, finalizing symbol for constructing a test sequence. The selected symbol will be excluded from the further iterations and included in the list of covered symbols. Thus, the approach uses a symbol coverage criterion to assess the adequacy of the test generation. Once the required symbol coverage ratio is achieved, the test generation terminates to exclude redundant test sequences. For example, setting symbol coverage to 100% requires covering all of the different symbols in the table. Once a predefined coverage ratio is achieved, test generation terminates and excludes redundant test sequences that are incomplete.

The approach comprises test preparation and testing steps. In the test preparation step, the tester models the GUI by an FSM using the tool JFLAP and also converts the FSM into a corresponding RE. A tool analyzes the RE to construct the context table that contains contextual information about the symbols contained in the RE. In the testing step, the developed tool applies to the context table to generate test sequences.

A mutation testing technique is used to validate the approach (in Section 7.2). This technique entails the generation of mutants of the GUI as its faulty versions [8, 9] to model functional and sequencing faults. Mutants are obtained by applying mutation operators to the source code of the GUI. These mutation operators form slight changes in the code, such as manipulating an assignment. The test sequences generated from the RE and its context table will then be applied to these mutants. If they reveal a fault, the mutant is said to be “killed”; that is, the test sequence was successful. Otherwise, a behavioral equivalent mutant of the GUI is needed. A test automation tool will be used to run the tests.

5.2 THE APPROACH BASED ON CONTEXTUAL REGULAR EXPRESSION

The proposed approach comprises two steps: Test preparation and testing. In the test preparation step, the GUI behavior is modeled using an FSM using the JFLAP tool that also converts the FSM to RE and finally analyzes the RE to construct a context table.

A random test generation approach is employed in the testing step in this thesis. Figure 5.2.1 depicts the concept of the proposed approach.

Test Preparation

A GUI program is the input of this step. A tester models the GUI manually to obtain an appropriate finite state machine (FSM) representation. The tester can also draw the FSM model from GUI using the JFLAP tool that is then utilized to convert FSM into the RE model. RE contains missing context information that is necessary for random test generation. Context refers to the location of a symbol in RE and its relations with other symbols. For example, “[(xy (t+z) *)]” is a RE (based on Definition 3.2 in Section 3.1). The right side of the symbol “x” is only “y” and the left side is the symbol “[”. Thus, the right side of the “x” symbol in the context table contains the symbol “y,” and the left side is “[”. The terms right context and left context of a symbol are used. The context table contains all symbols of the RE and their right and left context. The definition of the context table, including an example, is already given in Section 3.3.

Moreover, PQ-Analysis uses indexing the RE to remove ambiguities in the RE that might cause missing context-based faults. Consider the same symbol in different locations in the RE; covering a symbol means addressing only one symbol in one location and missing others. Thus, PQ-Analysis uses indexing to overcome these problems.

Testing

The output of the PQ-Analysis tool [81, 94] is the input of the testing step. The approach randomly traverses the context table from the initial to the end symbol.

A symbol coverage criterion is used to assess the adequacy of the test generation. The symbol coverage criterion is a predefined ratio to

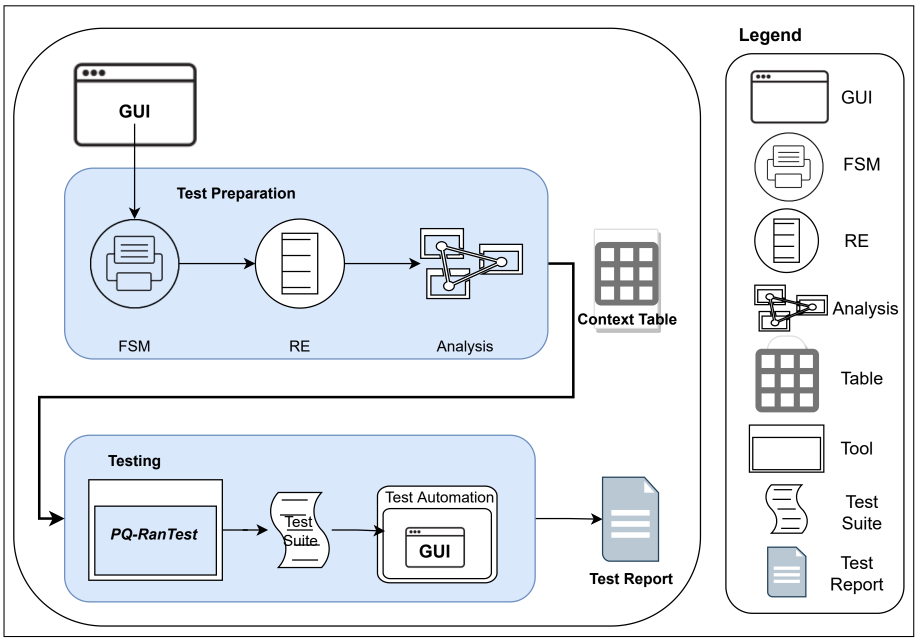


Figure 5.2.1: General Overview For the Approach Based on Contextual RE

terminate test generation when it is achieved. For example, 80 coverage ratios refer to 80% of all the different symbols in the context table required to be contained in the already generated test sequences for test generation termination.

When the predefined symbol coverage ratio is achieved, the test generation process is terminated by s. Then, the generated test sequences are saved together into a test suite. Finally, a test automation tool, such as Selenium, automatically executes this test suite on the GUI. The test report from the testing step finishes the procedure.

A tool called PQRTestGen is developed by implementing the proposed test generation algorithm based on contextual RE to automate the steps defined in Figure 5.2.1. The tool takes the analysis output and generates the test sequences using the context table from the analysis result. This tool will be utilized in case studies for GUI programs in Section 7.2, and results in comparison with other similar tools will be presented in Section 7.3.2.

CHAPTER 6

MODEL-BASED IDEAL TESTING (MBIT)

In this chapter, a model-based ideal testing approach (MBIT) is proposed to show the presence and absence of faults in the sequential systems. These sequential systems are HDL or GUI programs in the thesis, and the following sections present the MBIT approach for them.

6.1 MODEL-BASED IDEAL TESTING FOR HDL PROGRAMS

This section provides the general idea on HDL programs of the proposed approach and shows that it results in MBIT suites. In addition, this section elaborates on the main steps to be undertaken.

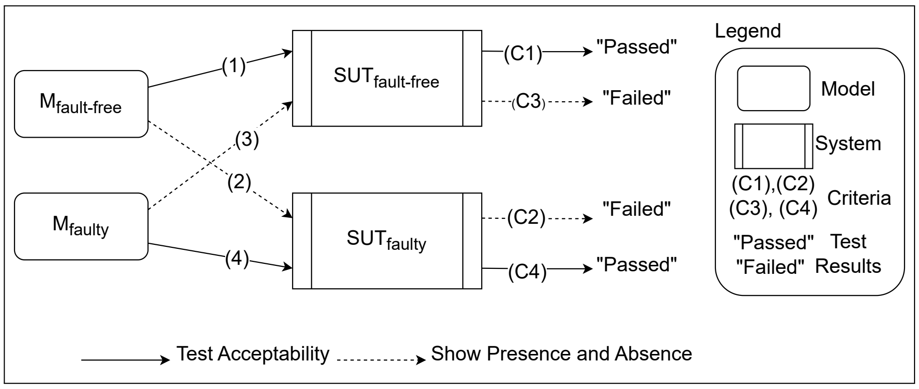


Figure 6.1.1: Test Composition For MBIT

6.1.1 *Model-based Ideal Testing (MBIT) and Its Proof*

The proposed approach consists of two main steps: test preparation and test composition. These steps contain some fundamental concepts and definitions, which are explained as follows:

Test Preparation

- It is assumed that a model of the HDL program is available. If not, a model will be extracted from the reference HDL program that is called the original (supposedly fault-free) program. Further, it is assumed that the generated/provided FSM model is deterministic.

- The model is mutated using mutation operators to develop faulty models.
- Supposedly fault-free and faulty models are converted to regular expressions and to their corresponding context tables that are used to construct all combinations of legal and illegal sequences of transitions. This is more powerful than transition coverage, which can be achieved by using FSMs.
- Using a test generation method, adequate test cases for positive and negative testing are generated.

Test Composition

The set of legal test sequences is executed on (supposedly) fault-free design under test (HDL programs) for positive testing. (See Figure 6.1.1)

Then, the "passed" test sequences are composed into a test suite TS_{ff-ff}, called a test suite generated from a (supposedly) fault-free model ($M_{fault-free}$) and executed on the (supposedly) fault-free SUT ($SUT_{fault-free}$).

Criterion 6.1. *(C_1): $\forall t_i$ ($M_{fault-free}$) | $SUT_{fault-free}$ (t_i) $\wedge$ OK(t_i) (Definition 2.8).*

The model extracted from the original program is expected to be correct. It is also assumed that the starting point is this supposedly correct program as a reference that represents the (formal) specification (normally) provided by a customer. The correctness of the model using criterion C_1 is checked by executing a generated test from the original model ($M_{fault-free}$) on the original program ($SUT_{fault-free}$).

The set of legal test sequences is executed on faulty systems for positive testing.

Then, the "failed" test sequences are composed into a test suite (TS_{ff-f}), called a test suite generated from a (supposedly) fault-free model ($M_{fault-free}$) and executed on the original SUT (SUT_{faulty}).

Criterion 6.2. *(C_2): $\forall t_j$ ($M_{fault-free}$) | SUT_{faulty}(t_j) $\wedge \neg$ OK(t_j) using Definition 2.8.*

The set of illegal test sequences for each mutant model is executed on faulty systems.

Then, the "failed" test sequences are composed into a test suite (TS_{f-ff}), called a test set generated from a faulty model (M_{faulty}) and executed on the (supposedly) fault-free SUT ($SUT_{fault-free}$).

Criterion 6.3. *(C_3): $\forall t_m$ (M_{faulty}) | $SUT_{fault-free}$(t_m) $\wedge \neg$OK(t_m) using Definition 2.8.*

The set of illegal test sequences for each mutant model is executed on the (supposedly) fault-free system. (See Figure 6.1.1)

Then, the "passed" test sequences are composed into a test suite (TS_{f-f}), called a test set generated from a faulty model (M_{faulty}) and executed on faulty SUT (SUT_{faulty}).

Criterion 6.4. *(C_4): $\forall t_k$ (M_{faulty}) | $SUT_{faulty}(t_k) \wedge$ OK(t_k) using Definition 2.8.*

Definition 6.1. *Model-based Ideal Test (MBIT) suite: T is an MBIT suite iff $\forall$t(M) $\in$ T $\wedge$ Satisfy(t, (σ)) | (Reliable(σ) $\wedge$ Valid(σ). (See the definition of Reliable and Valid in Section 2.5 in Definitions 2.10 and 2.11)*

Test sets were composed using $C_1, C_2, C_3, andC_4$ are TS_1, TS_2, TS_3, and TS_4, respectively. A test suite T equals to a set of these test sets T:=$TS_{ff-ff}, TS_{ff-f}, TS_{f-ff}, TS_{f-f}$ with respect to C:=$C_1, C_2, C_3, andC_4$.

To show that these test sets constitute an ideal test, the three requirements of an ideal test, namely acceptability, reliability/consistency, and validity/effectiveness, are examined in three lemmas and their proofs as follows;

(i) The test sequences generated from (supposedly) fault-free ($M_{fault-free}$) and faulty ((M_{faulty})) models either pass or fail on the (supposedly) fault-free system ($SUT_{fault-free}$) and faulty system (SUT_{faulty}) (Acceptability).

Lemma 6.1. *: $\forall t_i \in TS_{ff-ff}$, $\forall t_j \in TS_{ff-f}$, $\forall t_k \in TS_{f-ff}$, $\forall t_m \in TS_{f-f}$ $\subseteq$ T $\Rightarrow$ Successful(T) $\vee$ Fail(T)?*

Proof:

$\forall t_i \in$ T | (OK(t_i) $\wedge$ $SUT_{fault-free}(t_i)$) $\vee$ $\forall t_j \in$ T | ($\neg$OK(t_j) $\wedge$ $SUT_{faulty}(t_j)$)

$\forall t_k \in$ T | (OK(t_k) $\wedge$ $SUT_{fault-free}(t_k)$) $\vee$ $\forall t_m \in$ T | ($\neg$OK(t_m $\wedge$ $SUT_{faulty}(t_m)$)

Thus, test sequences t_i, t_j, t_k, and t_m, are either Successful(T) or Fail(T) as defined in Definition 2.9 (Section 2.5).

(ii) All test sequences satisfying corresponding selection criteria and generated from (supposedly) fault-free and faulty models either pass altogether or fail altogether on the faulty and (supposedly) fault-free systems (Reliability/Consistency).

Lemma 6.2. *: $\forall t_i \in TS_{ff-ff}, \forall t_j \in TS_{ff-f}, \forall t_k \in TS_{f-ff}, \forall t_m \in TS_{f-f} \subseteq D \mid (Satisfy(t_{i \vee j \vee k \vee m}, C)) \Rightarrow$ Reliable (C)?*

Proof:

Positive Testing:

$\forall t_i \in$ T, Satisfy(t_i, C_1)$\Rightarrow$Successful(t_i) thus Reliable(C_1) as defined in Definition 2.10.

Any ti belonging to T acquired from criterion C_1 is Successful t_i. Thus, C_1 is reliable as defined in Definition 2.10.

$\forall t_j \in T$, Satisfy(t_j, C_2)$\Rightarrow$Fail(t_j) thus Reliable(C_2) as defined in Definition 2.10.

Any t_j belonging to T acquired from criterion C_2 is Fail t_j. Thus, C_2 is reliable as defined in Definition 2.10.

Negative Testing:

$\forall t_k \in T$, Satisfy(t_k, C_3)$\Rightarrow$Fail(t_k) thus Reliable(C_3) as defined in Definition 2.10.

Any t_k belonging to T acquired from criterion C_3 is Fail t_k. Thus, C_3 is reliable as defined in Definition 2.10.

$\forall$ $t_m \in T$, Satisfy(t_m, C_4) $\Rightarrow$ Fail(t_m) thus Reliable(C_4) as defined in Definition 2.10.

Any t_l belonging to T acquired from criterion C_4 is Successful t_l. Thus, C_4 is reliable as defined in Definition 2.10.

(iii) There are some criteria from which the test sequences satisfying these criteria and generated from (supposedly) fault-free and faulty models reveal the faults or testify their absence (Validity/Effectiveness).

Lemma 6.3. *:* $\forall t_i \in TS_{ff-ff}, \forall t_j \in TS_{ff-f}, \forall t_k \in TS_{f-ff}, \forall t_m \in TS_{f-f} \subseteq D \mid (Satisfy(t_{i \vee j \vee k \vee m}, C)) \Rightarrow$ *Valid (C)?*

Proof:

C_1: $\forall t_i \in T \mid$ Satisfy(t_i, C_1)$\wedge$ $SUT_{fault-free}$ (t_i)$\Rightarrow$OK(t_i) thus $\neg$Valid(C_1) as defined in Definition 2.11.

C_2: $\forall t_j \in T \mid$ Satisfy(t_j, C_2)$\wedge$ SUT_{faulty} (t_j)$\Rightarrow$ $\neg$OK(t_i) thus Valid(C_2) as defined in Definition 2.11.

C_3: $\forall$ $t_k \in T \mid$ Satisfy(t_k, C_3) $\wedge$ $SUT_{fault-free}$ (t_k)$\Rightarrow$ $\neg$OK(t_k) thus Valid(C_3) as defined in Definition 2.11.

C_4: $\forall$ $t_m \in T \mid$ Satisfy(t_m, C_4) $\wedge$ SUT_{faulty} (t_m)$\Rightarrow$OK(t_m) thus $\neg$Valid(C_4) as defined in Definition 2.11.

Theorem 6.1. *The test suite* T_{ideal}*, which is constructed using criteria* C_2 *and* C_3*, forms an MBIT suite as defined in Definition 2.12 (Section 2.5).*

Proof: It is shown in Lemma 6.1, Lemma 6.2, and Lemma 6.3 that T_{ideal} selected by using reliable and valid criteria C_2 and C_3 constitutes MBIT suites.

6.1.2 *Application*

As discussed in Section 6.1, the test selection criteria C_1, C_2, C_3, and C_4 can be used for creating ideal test suites using the (supposedly) fault-free and faulty models of an HDL program. C_1 can be used to check the acceptability of the original program. As the specification or the model is assumed to be correct, the test sequences generated by applying C_1 can be used to check if the given original program complies with this model. If any of the test sequences fail, the program does not comply with the model, and the procedure of applying the ideal test is terminated. Otherwise, the test sequence is acceptable, and the procedure can continue. To this end, the criterion C_1 is considered as the pre-condition of the proposed approach.

Criterion C_4 can be used to check if each mutant model and corresponding mutant HDL program are consistent and if the test sequences generated from the mutant model are acceptable. For this purpose, all of the generated test sequences from a mutant model should be successful when it is executed on the corresponding mutant HDL program. Otherwise, the mutant model and related mutant HDL program are not consistent, and the generated test sequences are not acceptable. The tester can either re-do the injection or terminate the whole procedure. As the mutant model and its mutant HDL program represent a fault, the accepted test sequences will be used in this approach.

Also, according to the MBIT approach discussed in Section 6.1.1, the C_2 and C_3 criteria can be used to select the test sequences that are used to check the presence and absence of the pre-defined faults. To this end, the criterion C_2 selects the test sequences generated from the original model (the result of C_1) and fails on the mutant HDL program. This test suite (which is a subset of C_1) includes the test sequences that can detect the very specific fault (represented by the faulty HDL program) and is called a positive test suite for that fault.

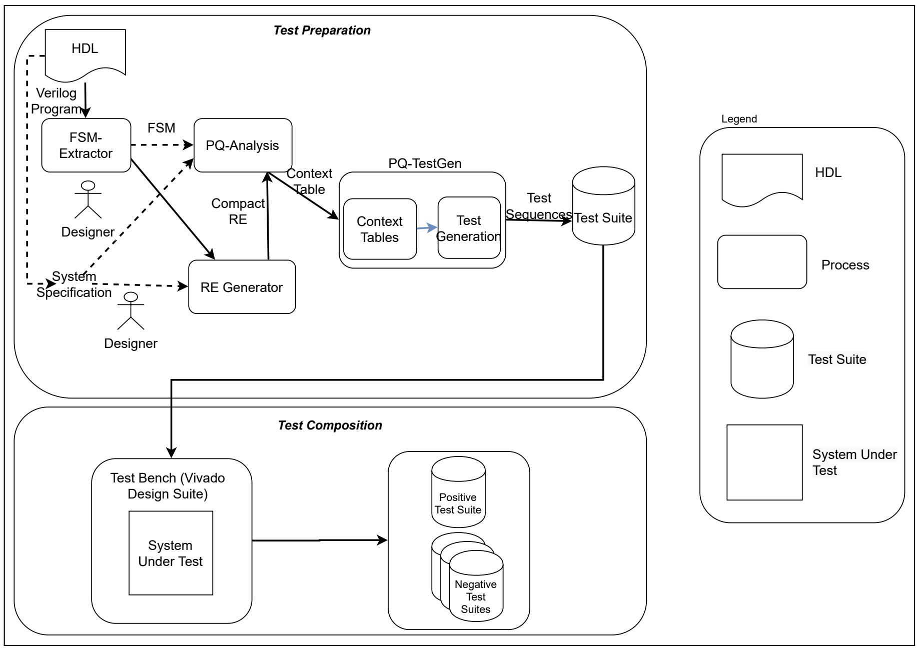


Figure 6.1.2: General Overview of MBIT For HDL

On the other hand, the criterion C_3 selects the test sequences generated from a mutant model (the result of C_4) and fails on the original HDL program. As these test sequences represent the fault (which is related to the mutant model), they can be used to check the absence of that fault in any new HDL program. These test sequences (which are a subset of C_4) constitute a negative test suite for that fault.

To summarize the use of criteria for MBIT, it can be stated C_1 and C_4 are used as the pre-condition of the ideal testing and generate the initial test sequences. Also, criteria C_2 and C_3 are used to select the positive and negative test suites for MBIT to check the presence and absence of faults.

Figure 6.1.2 shows a general overview of the proposed approach, including test preparation and test composition steps. In this figure, the straight lines represent the path preferred in this thesis, while dashed lines are other possible options. The curved rectangle covering the HDL program represents source code that can be input to the FSM-Extractor tool. Regular rectangles display utilized processes and tools within the thesis.

The test preparation step starts with the extraction of an FSM model automatically from the HDL program by means of the FSM extractor tool, for which more details are provided in Section 7.5. The FSM extraction is useful when the HDL program is available. However, the test engineer can obtain the FSM model manually from the system specification. Then, the generation of the mutants of the FSM is done using the insertion, omission, and replacement mutation operators [58]. The resulting FSM models can be given to a RE Generator tool to convert the models to RE. Once a compact RE has been obtained for the FSM, the PQ-Analysis tool is used to construct a context table with the algorithms explained in Section 3.3 for PQ-Analysis. The proposed approach proceeds through a straight line given in Figure 6.1.2. The result of the PQ-Analysis tool is given to the PQTestGen tool for test generation, which is the final part of the test preparation step.

The testing step contains the execution of test sequences on the corresponding HDL programs for positive and negative testing and then the selection of test sequences that fail or pass. The selection step results in ideal test suites that satisfy the requirements of the *ideal test*. Therefore, the proposed methodology holds these requirements. The details of the test preparation and test composition are defined in the following sections.

6.1.3 *Test Preparation*

The test preparation step comprises model construction, mutation, conversion, and test generation sub-steps, see Figure 6.1.3. The following sections give the details of these sub-steps. HDL programs are considered as a design under test.

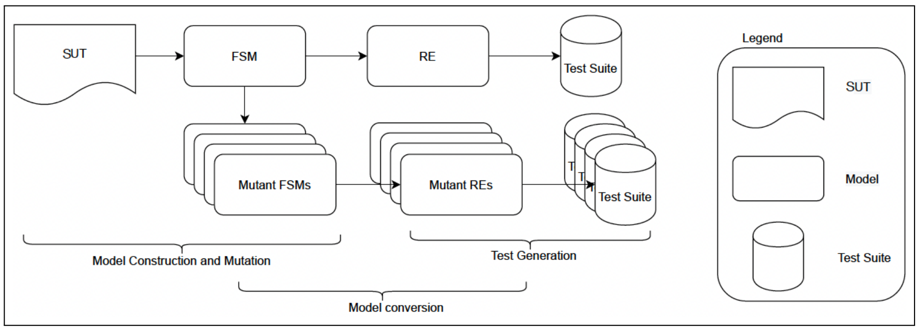


Figure 6.1.3: Test Preparation For HDL

Model Construction

In model construction, the FSM-Extractor tool parses the HDL program and extracts the states and transitions from the "case" statements in the code. This gives us the FSM model of the HDL program. Also, the test engineer can create the FSM model manually using the specification. In any case, the proposed methodology requires a behavioral model of the system to be prepared.

Therefore, the proposed approach is applicable for any sequential circuit given at the behavioral level by means of a Verilog HDL program, embedded system, and/or cyber-physical system. However, the borders are limited to the systems that an FSM-like model can model. An FSM model cannot model the combinational circuits and is therefore excluded from the scope of the thesis. This partially addresses HDL-RQ1 of the thesis.

Model Mutation

The mutation operators are applied to the original FSM to acquire the mutants. Note that applying the omission (O) and insertion (I) operators consecutively (for the state (S) and/or transition (T) replace) can realize the replace (R) operator. Moreover, the state omission operator requires the transition omission (TO) to delete the dangling transition(s), which will be created after removing a state at one end of the transition. This step uses mutation testing, as it is defined in Section 2.4.

The sequence of the mutation operators must be selected carefully because inattentive selection can result in non-determinism that cannot be used for the proposed approach.

It is possible to obtain mutant regular expressions from the original one using specific RE-mutation operators [95]. This thesis uses FSM mutation operators rather than RE mutation operators. However, utilization of RE mutation operators decreases the conversion cost because only the original FSM is converted to RE, and mutant RE models can be obtained from the original RE model. This advantage is only possible for transition mutation operators. The state mutation operators require extensive analysis, as the states are unavailable in the REs. Consequently, this thesis selects only FSM mutation operators for a mutant generation.

Listing 7: Pseudocode of the test generation algorithm

```
Open PQ Result File;
while ((read line by line PQ Result File) {
       if (find "Right−Context − Table:" line) {
           get next line;
            matrix++;
        while (read next line and line is not equal null) {
                 if (matrix equal 0 (forward analysis)) {
                        add line to forward array}
                 else if (matrix equal 1 (backward analysis)) {
                        add line to backward array}}}}
    close file
    create adj matrix for forward right
     for (each element i in forward right array)
          for (each element j in forward right array)
                 copy adjmatris_right [i][j] = forward array right[i][
                     j]
                  t[i][j] = new TestSuite();
    Set all nodes to "not visited";
    q = new Queue();
    q.enqueue(initial node);
    while ( q is not equal empty ) do
    {
        x = q.dequeue();
        if ( x has not been visited ){
        visited[x] = true; // Visit node x
        if (x is initial symbol){ //This defines initial case
            for (all reachable symbols y from x){
                for (each symbol i in y)
                    t[i][0]=x   //x is openning symbol "["
                    t[i][1]=y}}//y is the symbol reachable from
                         "["
        for (each edge (x, y)) //using all edges
            t.addSymbol(y)  //new symbol is added to t[i][j]
                with proper index
            if ( y has not been visited )
                q.enqueue(y);}}        // Use the edge (x,y)
        while(all last symbols in t[i][j] is not equal "]" ){ // t[i
            ][j] is a test suite
            if (last symbol in t[i][j] is "]"){
                opt(t[i][j−1]) = t[i][j]}
            elseif (t[i][j] is equal to opt[i][j−1]){
                t[i][j].addsymbols(opt(t[i][j−1]))
                opt(t[i][j−1]) = t[i][j]}}
```

Model Conversion

In the model conversion step, the original and mutant FSM models provided in the previous steps can be converted to the corresponding REs by means of different algorithms, such as state elimination [90] or the Brzozowski method [96]. The proposed methodology uses JFLAP [92] for converting FSM to RE due to the fact that it produces more compact RE models.

Test Generation

The input of the test generation step is the converted RE given in the previous step. Then, the original and mutant REs are given to the PQ-Analysis tool for indexing and constructing the context tables.

To generate test sequences, the PQTestGen tool uses the result of the PQ-Analysis tool. The PQTestGen tool implements the breadth-first search (BFS) algorithm on the context tables. Listing 7 shows the pseudo-code of the test generation algorithm. In this pseudo-code, lines 1-11 define the parsing of the resulting file from the PQ-Analysis tool (including CT). Lines 12-16 show the construction of an adjacency matrix from the parsed file, and lines 17-33 define the DFS algorithm used on the adjacency matrix to construct a set of test sequences based on the symbol coverage criterion. Finally, lines 34-39 complete the construction of test sequences.

A symbol coverage criterion terminates the DFS algorithm if the desired coverage ratio is achieved. This criterion defines the amount of the symbol covered by the CT. However, in this thesis, the symbol coverage criterion is set to 100%, which leads to the covering of all symbols of CT in the test sequences.

6.1.4 *Test Composition*

The test composition step contains pre-selection and test suite construction sub-steps, see Figure 6.1.4, in which original and mutant HDL programs are considered as SUT and mutant SUTs, respectively. The following sections define the details of these sub-steps.

Pre-Selection

In the *test composition* step, the test sequences generated from analyzed RE models are executed on the corresponding original and mutant HDL programs. To this end, four different testing scenarios are used to do holistic testing, as it is defined in Section 2.3;

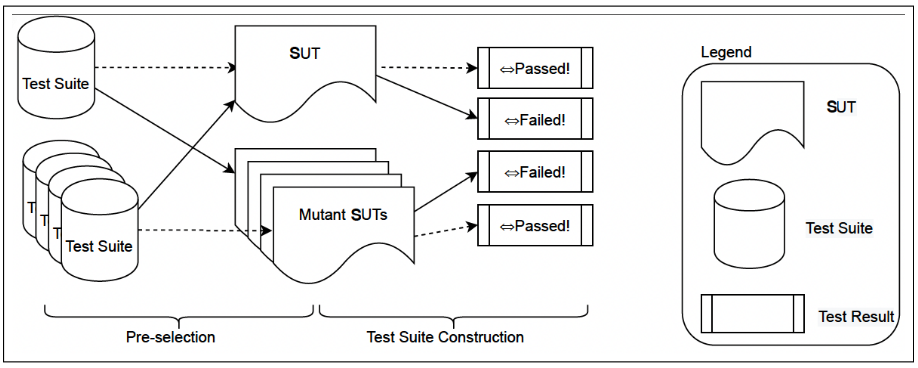


Figure 6.1.4: Test Composition For HDL

1. Positive Testing with original HDL program: Execution of test sequences generated from the original model on the corresponding original program.
2. Positive Testing with mutant HDL program: Execution of test sequences generated from the original model on the corresponding mutant programs.
3. Negative Testing with original HDL program: Execution of test sequences generated from the mutant models on the corresponding original program.
4. Negative Testing with mutant HDL program: Execution of test sequences generated from the mutant models on the corresponding mutant programs.

Test execution scripts are written in Verilog using Xilinx Design Suite. These scripts execute the generated test sequences in the corresponding HDL program using the test bench will be defined in Section 7.5. There are positive and negative testing steps for differentiating the execution processes. In positive testing, the test sequences generated from the original ((supposedly) fault-free) model are executed on the corresponding mutant designs that are derived from mutant FSM models using code-level mutation operators. Thus, there is a one-to-one correspondence between mutant models and mutant (faulty) designs.

Regarding the fault coverage aspect (considering the HDL-RQ1), the mutants generated from the FSM address the following HDL faults: single-output bit stuck-at 0/1, case bit stuck-at 0/1, condition stuck-at True/False, and their combinations.

Test Suite Construction

The test sequences resulting from the above-mentioned scenarios are selected based on the requirements, which are called test selection criteria, defined in Table 6.1.1. There are two execution steps for each of the positive and negative tests. These executions are based on the test sequences generated from the original analyzed RE and the mutant models. The results of these executions are either passed or failed, from which successfully passed tests are selected when the test sequences from the original model are applied to the original HDL program and the test sequences from mutant models are applied to the mutant HDL program. Moreover, the failed tests are selected when the test sequences from the original model are applied to the mutant HDL and the test sequences from mutant models are applied to the original HDL.

Table 6.1.1: Test Selection Criteria

Test Execution	Original HDL Program	Mutant HDL Program
(1) Positive	Test passed!	-
(2) Positive	-	Test failed!
(3) Negative	Test failed!	-
(4) Negative	-	Test passed!

In a similar manner, the test sequences generated from the mutant models can be applied to the mutant HDL programs. Therefore, an arbitrarily faulty design would often return a test result of failed using any other faulty version.

This can verify the presence and absence of defined faults by the positive and negative test suites (as part of the ideal test). This can be considered as the main outcome of applying the ideal test procedure (which addresses a part of HDL-RQ1).

There are three fundamental requirements of the ideal testing defined in Section 2.5. The first one is regarding the results of the test execution that can be either successful (pass) or unsuccessful (fail). The second one is the reliability (consistent) of each test criterion for the test generation, which is successful if the criterion is satisfied OR if all tests that satisfy the criterion are not successful (fail). The last requirement is the validity (effectiveness) of the test criterion for the test execution to check whether the test sequences satisfying the criterion are revealing the fault(s) or not.

For test suites given in Table 6.1.1, the test selection criteria are reliable because the test execution results are either successful (pass) or unsuccessful (fail). The "failed" test sequences satisfy validity as they reveal the fault(s). Thus, the resulting test suites satisfy the requirements of the *ideal test*.

6.2 MODEL-BASED IDEAL TESTING FOR GRAPHICAL USER INTERFACE

In this section, test preparation and testing steps are provided in Section 6.2.1 and 6.2.2, being two main MBIT stages, offer necessary information supporting sub-steps. The FSM and the RE models are used as defined by Hopcroft et al. [79] in this thesis.

In Figure 6.2.1, the general flow of the current methodology is shown. The test preparation step contains the model and test generation sub-steps. The testing step contains test selection and test execution sub-steps. The straight lines perform the paths that are utilized by the thesis. The dashed lines show other options can be employed. For example, the FSM of the GUI program can be obtained from the specification by the designer and then given to the PQ-Analysis tool.

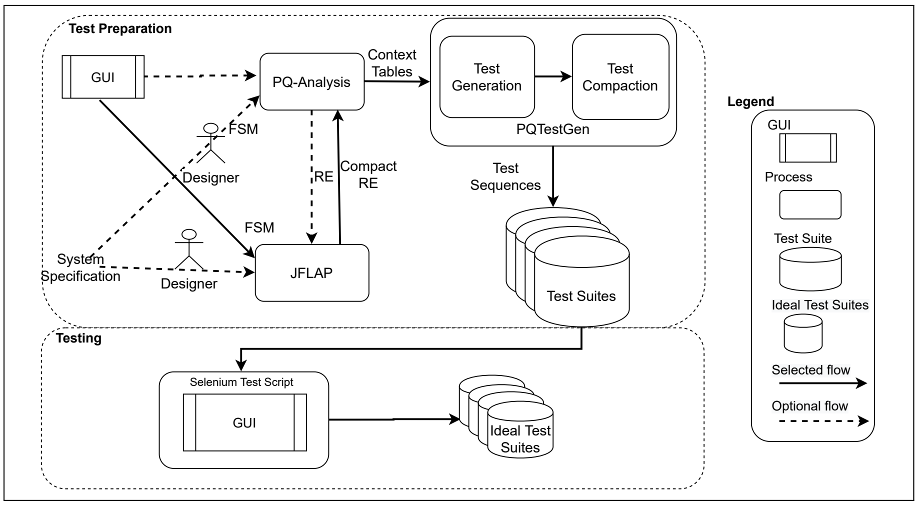


Figure 6.2.1: General Overview of MBIT For GUI

The formal definitions and corresponding proofs for MBIT are provided in Section 6.1.1, and these explanations extend their applicability to GUI programs. For a comprehensive understanding of the sophistication involved, the reader can consult Section 6.1.1 for detailed exposition and rigorous explanation of the concepts therein.

6.2.1 *Test Preparation*

An FSM represents the GUI under test and is then converted to a corresponding RE by the JFLAP tool. Artificial faults are seeded into the FSM to acquire mutants. Each mutant can contain one or more faults.

An FSM model can be automatically generated from the GUI specification, or one of the GUI ripping methods [97, 98] automatically generates the proper model by using reverse engineering techniques. It is assumed that the specification is missing, and the model of the GUI under test is generated manually using the JFLAP [1]. The pseudo-code is provided for generating a model from a GUI of a web page in the listing 8. The algorithm given the listing 8 starts opening the system under test (SUT). Then, it proceeds by checking all elements of the current page of the SUT. These elements can be "radio button", "edit field", "text box", "combo box", or "button". Once selecting the current element (event), the corresponding entry is added to the model with its input and output response. After finishing all elements on the current page, the algorithm proceeds to the next page. This procedure continues until all the elements for all pages of the SUT are explored. While exploration carries on, the corresponding responses are added to the model. Once the exploration is finished, the model generation is also finished.

Listing 8: Pseudocode of the model generation algorithm

```
Open the SUT (GUI of a web page);
while (check all events of the SUT for all pages) {
    for (each elements (events) at the currentPage){
        if (event==radio button){
            click=radio button;
            Model=AddEvent(click);
        }else if (event== edit field){
            click=edit field;
            Model=AddEvent(click);
        }else if (event== text box){
            click=text box (addRandomText);
            Model=AddEvent(click);
        }else if (event== combo box){
            click=text box;
            Model=AddEvent(click);
        }else if (event== button){
            click=text box;
            Model=AddEvent(click);
        }
    }
    currentPage = nextPage;}
```

For example, SUT is Gmail login page [2]. Once the user clicks this page, he/she has several options, such as a textbox for the user's email or telephone number. The user must enter his/her email account into this text box to proceed to the next step, or he/she can create a new account by clicking the "create account" button. Let us again suppose that the tester (the person responsible for the model generation) enters the correct email address into the textbox. He/she needs to add this event to the model with the email entry and the corresponding response of the SUT to this action. This procedure is

1 An open-source modeling tool, Available online at http://www.jflap.org/

2 Gmail Login Page, Available online at https://gmail.com/

applied to all elements of the current login page by the tester and added to the model with corresponding responses. Finally, the tester constructs the Gmail login page model once he/she finishes all elements of this web page. The manual construction is straightforward and easy for this kind of login page, which is the main bottleneck of automatic model generation approaches due to missing correct information on user accounts. Therefore, the automatic model generation approach requires user intervention to cope with this kind of problem. Another advantage of manual construction is to decide the model's capacity by neglecting unnecessary features.

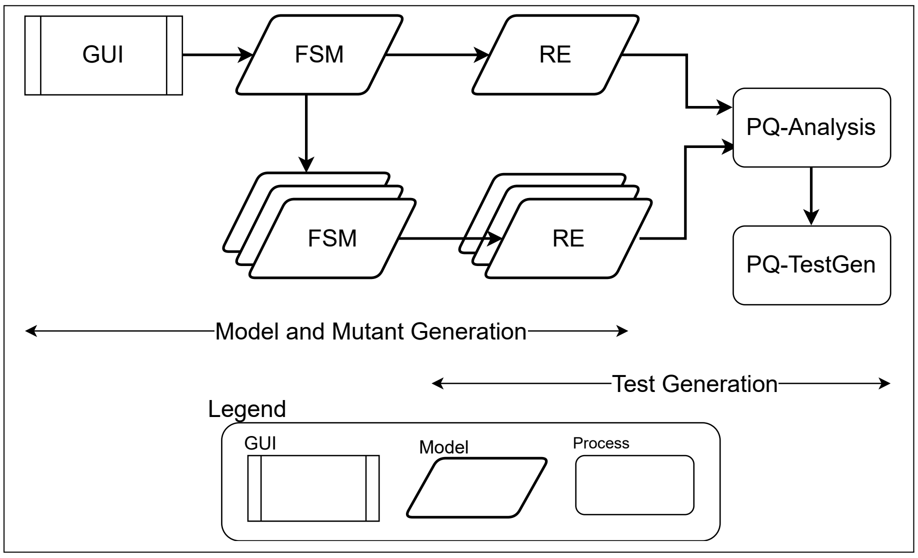


Figure 6.2.2: Test Preparation For GUI

The FSM's insertion, omission, or replacement of the state(s) or transition(s) to acquire mutants are utilized. The following definitions and examples are presented to elaborate on acquiring mutant FSM models.

- The insertion operator (IO) adds an extra transition(s) or state(s) into the FSM.
  Example: IO(s0, z, s1) refers to adding an extra "z" transition from s0 to s1, or IO(s0, z, s2, x, s1) refers to adding an extra state s2 between s0 and s1 with "z" and "x" transitions.
- Omission operator (OP) deletes a transition(s) or state(s) from the FSM.
  Example: OP(s0, x, s0) refers to deleting the transition "x", or OP(s1) refers to deleting the state s1 with corresponding transitions.
- Replace operator (RO) substitutes a transition(s) or state(s) from the FSM.
  Example: RO(s0, x, s0, z) refers to replacing the transition "x" with "z", or RO(s1, s2) refers to replacing the state s1 with s2.

To model semantic faults, the mutants require higher-order mutation; in contrast, insert a single fault in the model or code to create first-order mutants. In this higher-order, mutation applies the mutation operator more than once [99].

Once mutants are acquired, they are transformed the resulting FSM model into the RE models using the JFLAP tool that provides a more compact RE than the PQ-Analysis. The procedure of the FSM to RE conversion for the PQ-Analysis tool is presented in Section 3.3, including the pseudo-code for the conversion. The PQ-Analysis tool generates CTs that accommodate forward and backward information. This information is useful for generating more efficient test suites to increase the possibility of covering the faults because covering the only symbol without its right and left context does not guarantee the coverage of the modeled fault(s).

The PQTestGen [56] tool is used for test generation. The CT contains two different and independent tables, namely forward right and left CT. Therefore, two sets of test sequences from the forward right and left tables are collected. The forward right table, considering any overlapping between the two tables, is selected.

PQTestGen [56] parses the table and then traverses, starting from the initial symbol in a depth-first search manner in the first step. The traversing finishes once the final symbol is reached to construct complete test sequences. However, some sequences can be incomplete because of a different symbol in the final test suite by reaching coverage criteria to assess adequacy. For those partial sequences, the algorithm uses already complete sequences to complete them in the second step. The algorithm utilizes a compaction procedure to eliminate redundant sequences in the final step while keeping the coverage criteria in a predefined ratio.

PQTestGen [56] initiates from the opening symbol "[" and selects the next symbol from its forward right context. Test generation continues until the assessing coverage criterion is satisfied when all different symbols are in the resulting test suite. Kilincceker and Belli define the coverage criteria depending on the CT and extensively analyze their effectiveness for GUI testing [100].

6.2.2 *Testing*

The test execution and then test composition are the sub-steps of the testing stage. In the test execution, the test suites are run automatically on the corresponding GUI programs, and then these sequences are collected into MBIT test suites in the test composition step.

The test suites are run on the (supposedly) fault-free (original) and faulty (mutant) GUI programs in this step (see Figure 6.1.4). Note that this step is similar to the test composition step in applying MBIT on HDL programs. Hence, two different testing scenarios happen as follows;

Test selection criteria are a filtering mechanism to select satisfying test cases. It is important to note that the coverage criteria and selection criteria are not to be mixed. Coverage criteria are termination criteria utilized for the test generation procedure. However, test selection criteria (also called test criteria) are a filtering mechanism to accomplish the conditions of the ideal test. There is no intention or attempt in the current methodology for code-level or function-level coverage at the program level. The main intention of the coverage criteria used in the thesis is to assess the adequacy of GUI testing at the functional level with respect to events captured by test sequences.

Based on the test criteria, "failed" test sequences are collected into ideal test suites with respect to the PT and NT. To test the presence and absence of predefined faults, these test suites are utilized by using the PT and NT, respectively.

6.2.3 *Model Correctness*

The test generation algorithm from the model under analysis checks the model's correctness. Therefore, the suite generated from the model is executed on the system modeled. The entire test suite must be "passed" on the system modeled to satisfy model correctness. The test generation algorithm must be deterministic to avoid different results in each generation.

Criteria 6.1 and 6.4 (defined in Section 6.1.1) are used to satisfy model correctness. Hence, the original model is checked by executing the test suite generated from the original model on the original system under test concerning Criterion 6.1. Any mutant model is checked by executing the test suite generated from the corresponding mutant model on the mutant system under test. To this end, it can be satisfied that all the models hold model correctness.

Moreover, an assumption for the model's correctness is also made and presented in Chapter 1 to address this concern.

CHAPTER 7

CASE STUDIES, RESULTS, AND EVALUATION

This chapter presents case studies, results, and evaluations, including threats to the validity of the MBIT approach for HDL and GUI programs.

7.1 HDL-BASED CASE STUDIES

This section discusses the case study, namely a simplified Traffic Light Controller (TLC), implemented in Verilog HDL and modeled by an FSM. As shown in Figure 7.1.1, four traffic signals and four lanes are present. Every lane has a separate traffic light with Red, Yellow, and Green lights. The position of the lane and the corresponding light are specified using the input variables and previous state information. For security reasons, jumping between specific lane positions is not permitted to avoid a possible crash. This security mechanism is implemented into the HDL program. Any faults may cause severe car crashes.

The results on the sequence detector (SD) and the RISC-V processor [1] case studies are presented. The RISC-V processor [101] case study is called "CORE-V CV32E40P RISC-V IP" and was developed by OpenHW Group [2]. It is a 32-bit RISC-V core with a 4-stage pipeline. It provides higher code density, performance, and energy efficiency. The SD is a sequential HDL design to detect specific input patterns. HDL implementation of the SD, TLC, and RISC-V processor is realized by using a Xilinx Basys 3 Artix-7 FPGA development board and the Vivado 2017.4 design suite.

7.1.1 *Test Preparation for HDL Programs*

The preparation step comprises four sub-steps. These steps are model construction, model mutation, model conversion, and test generation. Briefly, an FSM model, defined formally in Section 3.1, is constructed from the HDL program (the construction approach is discussed in Section 7.5). Moreover, the selected mutation operators are applied to the FSM model to construct mutant models. Then, the original and

1 OpenHW Group CORE-V CV32E40P RISC-V, https://github.com/openhwgroup/cv32e40p

2 OpenHW Group, https://github.com/openhwgroup/cv32e40p

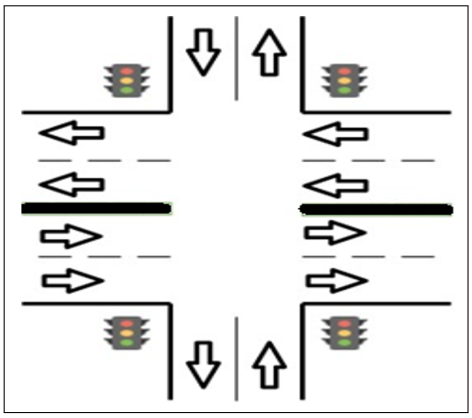


Figure 7.1.1: Block Diagram of the TLC

mutant FSMs are converted to RE. Finally, these models are analyzed to construct context tables from which test sequences are generated for positive and negative testing in the test generation step.

Model Construction

In the model construction step, the FSM model is extracted from the HDL of TLC automatically by means of the FSM-Extractor tool. This FSM is modeling the behavior of TLC. Then, it is converted to an RE, with an encoding of input/output combinations as the symbols, given in Table 2.2.1. The FSM model of the TLC, which is called the original model, is given in Figure 7.1.2. The FSM contains 9 states and 18 transitions.

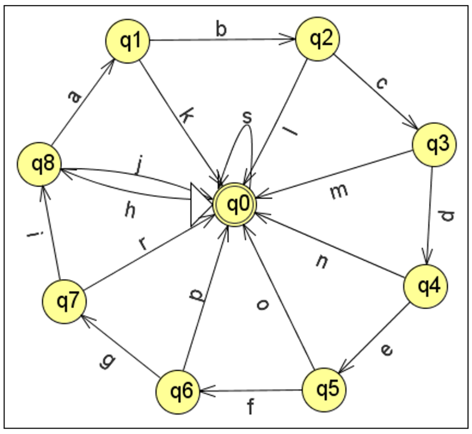


Figure 7.1.2: FSM of the TLC

The colors of the TLC are labeled as "g" for Green, "y" for Yellow, and "r" for Red, which are represented in the circuit by "001", "010", and "100" in binary format, respectively. Of course, this labeling with symbols and binary numbers is simply a design choice.

Table 7.1.1 presents the symbols, which are used for different transitions in the original FSM model, encoded with "input/output" com-

Table 7.1.1: Encoding of Transitions

Symbol	Combination	Symbol	Combination	Symbol	Combination
a	grrr o / yrrr	g	rrrg o / rrry	n	xxxx b - rrgr o / rrrr
b	yrrr o / rgrr	i	rrrg o / rrry	o	xxxx b - rryr o / rrrr
c	rgrr o / ryrr	j	xxxx b - grrr o / rrrr	p	xxxx b - rrrg o / rrrr
d	ryrr o / rrgr	k	xxxx b - yrrr o / rrrr	r	xxxx b - rrry o / rrrr
e	rrgr o / rryr	l	xxxx b - rgrr o / rrrr	s	xxxx 1 / rrrr
f	rryr o / rrrg	m	xxxx b - ryrr o / rrrr	h	xxxx o / grrr

binations. For example, the symbol "a" is encoded with "grrro/yrrr" in which "grrro" is a 13-bit input (3 bits for each color and 1 bit for reset signal) and "yrrr" is a 12-bit output. In this FSM, "x" represents the "don't care" condition. For example, in Table 2.2.1, the combination "xxxxb - grrr o" refers to a set of all 12-bit "don't care" followed by 1-bit "b" for the reset signal. In this combination, "-" refers to the exclusion operator for "grrr o", which means the "don't care" condition cannot be "grrr o".

Model Mutation

In this step, the supposedly fault-free model is used to generate faulty models by using mutation operators. For example, two faults are injected into the FSM by a combination of insertion/omission operators to realize the model mutation. Then, the related faults are also injected into the HDL program to realize the code-based mutation.

The first mutant, depicted in Figure 7.1.3, represents the combination of "missing state" and "transition fault" in the model. Therefore, state q3 and corresponding transitions are omitted in the model of this fault. In addition, an excerpt of the HDL codes of the first mutant and the original TLC is given in Fig 7.1.4. The code level fault in this figure represents stuck-at-bit fault in which bit(s) of the signals are stuck-at either 1 or 0. In order to represent the model-level faults at the code level, a higher-level mutation is required.

The second mutant, depicted in Figure 7.1.5, represents "extra state" and "transition fault" in the model. The state nine and corresponding transitions are inserted into the original FSM. Extra transitions are u, v, and y that are encoded with "ryrr o/rrgg", "rrgg o/ rrgr", and "xxxxx – rrgg o/ rrrr", respectively.

Model Conversion

In this step, the FSMs (the original and mutants) are converted to the corresponding REs. The JFLAP tool [92] is used to carry out the conversion. As a result, the RE (16) is obtained from the original FSM model. The same conversion steps are applied to the mutant models, and the corresponding mutant RE models are obtained.

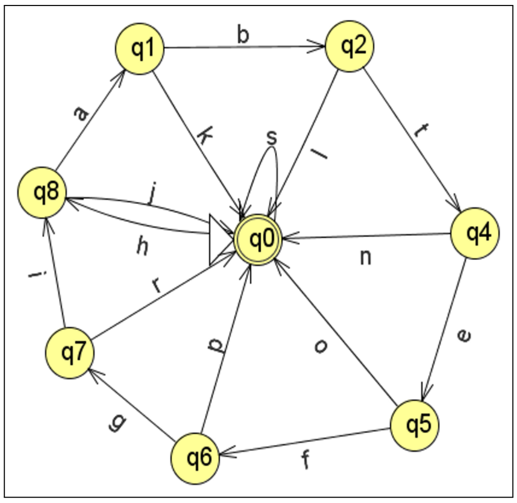
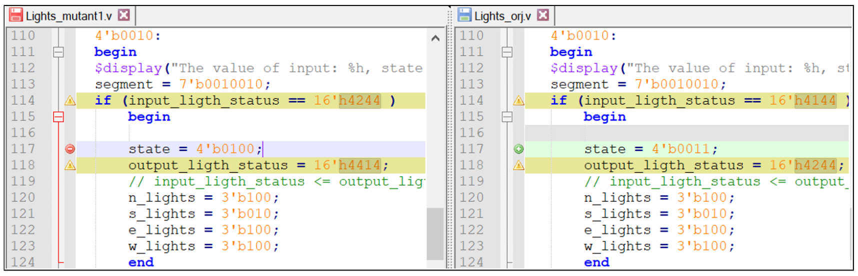


Figure 7.1.3: FSM of the Mutant One



```
4'b0010:
begin
$display("The value of input: %h, state
segment = 7'b0010010;
if (input_ligth_status == 16'h4244 )
    begin

    state = 4'b0100;
    output_ligth_status = 16'h4414;
    // input_ligth_status <= output_lig
    n_lights = 3'b100;
    s_lights = 3'b010;
    e_lights = 3'b100;
    w_lights = 3'b100;
    end
```

```
4'b0010:
begin
$display("The value of input: %h, st
segment = 7'b0010010;
if (input_ligth_status == 16'h4144 )
    begin

    state = 4'b0011;
    output_ligth_status = 16'h4244;
    // input_ligth_status <= output_
    n_lights = 3'b100;
    s_lights = 3'b010;
    e_lights = 3'b100;
    w_lights = 3'b100;
    end
```

Figure 7.1.4: The Code-Based Difference of Original and Mutant HDL

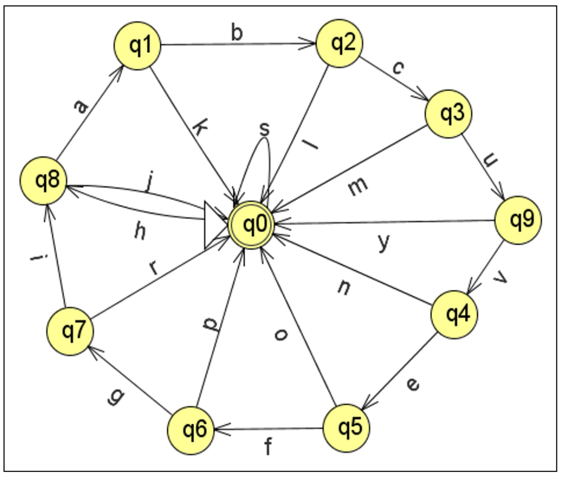


Figure 7.1.5: FSM of the Mutant Two

Test Generation

The test generation step starts by constructing the context table after analyzing the RE (for example, RE in (16)) using the PQ-Analysis tool. Then, test sequences are generated from the context table that represents RE in (16). These procedures are applied to the original RE and the mutant REs.

$$[(h(abcdefgi)^*(abcdefgr + abcdefp + abcdeo + abcdn + abcm + abl + ak + j) + s)^*] \tag{16}$$

To exemplify the procedure, the test sequences t_1 in (17) and t_2 in (18) are selected from the test suite generated from the context table of RE in (16).

$$t_1 = xxx0, grrr0, yrrr0, rgrr0, ryrr0, xxxxx - rrgr0 \tag{17}$$

$$t_2 = xxx0, grrr0, yrrr0, rgrr0, ryrr0, rrgr0, xxxxx - rryr0 \tag{18}$$

The exemplary test sequences, t_3 in (19) and t_4 in (20) are generated from the first and second mutants respectively.

$$t_3 = xxxx0, grrr0, yrrr0, rgrr0, rrgr0 \tag{19}$$

$$t_4 = xxxx0, grrr0, yrrr0, rgrr0, ryrr0, rrgg0 \tag{20}$$

These test sequences are members of the test suites generated from the original and mutant context tables. These test suites are an input of the composition step. The test suite obtained from the original model is used in positive testing, and the test suites obtained from the mutant models are used in negative testing for the composition step.

7.1.2 *Composition*

The composition step comprises pre-selection and test suite construction sub-steps. In the pre-selection step, the test suite generated from the original model is executed on the mutant HDL programs. In contrast, test suites generated from the mutant models are executed on the original HDL program for positive and negative testing, respectively. In the test suite construction step, the results of test execution are collected and analyzed to construct the test suites that satisfy the requirements of the ideal test, defined in Section 2.5.

Pre-Selection

The generated test sequences are executed on the corresponding HDL programs for positive and negative testing. The test bench executes these test sequences on the HDL programs. The results shown in Table 7.1.2 are collected from the execution of test sequences t_1 in (17), t_2 in (18), t_3 in (19), and t_4 in (20).

Table 7.1.2: Results From the Negative and Positive Testing

Test sequence	Original HDL	Mutant One	Mutant Two
t_1	-	Test failed!	Test failed!
t_2	-	Test failed!	Test failed!
t_3	Test failed!	-	-
t_4	Test failed!	-	-

Test Suite Construction

In this step, the results collected from the test execution step are used to test sequences that give a "fail" result (see Table 7.1.2) for positive and negative testing. The example test sequences t_1 and t_2 generated from the original model are executed on the mutant HDL programs. Both failed, meaning that the mutant gave a different output than the original HDL program. So, the mutants are killed. Therefore, the test sequences t_1 and t_2 satisfy the criterion C_2 defined in the proof of the ideal test in Section 2.5. So, they become members of the *ideal* test suite, which is used to test the presence of faults modeled by corresponding mutants. t_1 and t_2 imply the presence of faults since they are generated from the original model and executed on the mutant models. Therefore, they used to show that the fault is present in the system since they both failed, and they pointed out that the system they failed is faulty.

The example test sequences t_3 and t_4 generated from mutant one and mutant two models failed when they were executed on the original HDL program, meaning that the original HDL gave a different output than the mutant programs. Therefore, test sequences t_3 and t_4 satisfy the criterion C_4, see Table 7.1.2. These test sequences satisfy corresponding criteria and fulfill the requirements of the *ideal test*. To this end, the test sequences t_3 and t_4 become a member of the ideal test suit, which tests the absence of faults modeled by corresponding mutants. t_3 and t_4 imply the absence of faults since they are generated from the mutant model and executed on the original models. They are pointing out that the system failed and used to show the absence of faults. This means they represent the faulty behavior of the system in the model scope. Therefore, if they pass on any system, it can be concluded that the corresponding fault represented by the mutant model where t_3 and t_4 is absent in this system.

7.1.3 *Experimental Setup For HDL Programs*

Experimental studies were conducted on a Sequence Detector (SD), a Traffic Light Controller (TLC), and a RISC-V processor [101] [3]. Insert, replace, and delete mutation operators [58] were used to obtain model and code-level mutants. Table 7.2.2 provides HDL code level mutant profiles, including fault types and their quantities for TLC and SD. 158 HDL faults (18 for SD, 82 for TLC, and 58 for RISC-V) were studied with corresponding mutants. "Total" refers to the sum of the respective fault type rows or case studies columns.

The experiments concerned fault coverage, mutation scores, test suite sizes, test generation times, and test execution times for each case study. These metrics are investigated for different model-based test generation tools and algorithms. These tools and algorithms use coverage criteria (defined in Section 3.2) to assess its adequacy as a termination criterion of test generation. The symbols represent input/output combinations for each case study at a specific clock time corresponding to their sequential operating principle. These combinations are stored in the transitions of each FSM model (for example, see Table 2.2.1 for symbols and corresponding combinations for TLC). Therefore, the tools and algorithms terminated when achieving a specified coverage ratio.

The mutation score [102] is calculated as the ratio of the number of mutants killed to the total number of mutants. Equivalent mutants are subtracted from the total number of mutants to obtain more accurate results. The equivalent mutants are models or programs that the test suites cannot kill and exhibit the same behavior as the original model or program (refer to Section 2.4 for more details). Fault coverage is calculated as a percentage of detected faults different from mutation score. While the maximum mutation score is 1, the maximum fault coverage value is 100. If there is no equivalent mutant within the scope of the experimental study, the fault coverage percentage can be obtained by multiplying the mutation score by 100.

The strategy, explained in Section 6.2.3, is used to validate the original program's compliance with the extracted model for both original and mutants. Based on this strategy, the test suites generated from the original extracted model are executed on the corresponding original program. Also, the test suites generated from the mutant models are executed on the corresponding mutant programs. These original and mutant models are utilized when all these test suites "passed" successfully, meaning that the mutation scores for original and mutant models over original and mutant programs equal one. These execution results from the experiments are excluded because they are the

[3] OpenHW Group CORE-V CV32E40P RISC-V, `https://github.com/openhwgroup/cv32e40p`

Table 7.1.3: Mutant Profiles

Fault Types	Quantities			
	SD	TLC	RISC-V	Total
Output Bit Stuck-at 0/1	6	24	22	31
Case Bit Stuck-at 0/1	4	20	13	24
Condition Stuck at True/False	4	16	10	20
Higher Order	4	15	0	19
Hard to Detect	0	6	10	6
Total	18	82	58	158

starting point for selecting and utilizing the original and mutant models.

7.2 GRAPHICAL USER INTERFACE BASED CASE STUDIES

ISELTA is a commercial web portal for marketing tourist services and an online reservation system for hotel providers. It is a cooperative work between ISIK Touristic company and the University of Paderborn. The "Special" module provides agents the ability to promote special advertisements, such as the New Year event. The "Additional" module offers other advertisements rather than regular events. For each module, the GUI of ISELTA enables agents and providers to use different attributes for specific events to catch the interest of customers. It is written in PHP programming languages and contains 69323 lines of code for five different modules for each provider under the "Hotels" branch. Readers can log in to ISELTA using demo information given on the website as being a provider role.

7.2.1 *Test Preparation*

This section presents the test preparation step for only the "Special" module case study. However, a supplementary website for the "Additional" module[4] is provided, which also introduces the required information for reproduction.

Firstly, an FSM is manually constructed from the GUI program of "Special" for the ISELTA website. This module is called GUIs Under Test. An omission, insertion, and replace mutation operators [76] are performed on these FSM(s) for a mutant generation. Then, further steps are carried on as provided in Section 6.2.1 and 6.2.2.

There are many input areas and buttons in the main GUI of the "Special" module, and it is redundant and tedious to test each case of the module. Therefore, the thesis restricts the scope to a relatively

4 MBIT4SW, https://kilincceker.github.io/MBIT4SW/

small module in the application for evaluation. In this step, the FSM model is acquired by the GUI. To do this, the tester enters the ISELTA web page and proceeds to the "Special" module. He/She has listed all elements (events) of this module. These elements are given in Table 7.2.1. Then he/she applies the algorithm given in Section 6.2.2 in the listing 7. First of all, he/she tries to set required input boxes such as "price", "title", "number" elements. While the tester provides this information to the SUT, he/she is also added this information, including SUT's responses to the FSM model. When the tester satisfies about covering all elements and their combinations in the FSM model, he/she finishes the model construction procedure.

In the (supposedly) fault-free FSM of the "Special" module, a symbol is assigned for each event that becomes a transition label in the FSM. All symbols represent filling an input, clicking a button, or removing a text from an input. These action symbols enable the implementation of the Selenium test script. All event symbols are listed in Table 7.2.1.

In the case study, it is intended to catch functional faults (see the definition in Chapter 1 on page 5, paragraph 2, line 5) that directly affect the desired operation of the system based on user interaction with the GUI.

Example 7.1. *Functional Fault: Mutant one given in Table 7.1.3 is a functional fault in which the system does not add a new offer due to required empty input boxes. However, it reaches the final event called "add" event. Once the user clicks the "add" button, he/she might receive an error message that is triggered by the fault. This type of fault is called functional fault.*

Table 7.2.1: Symbols and Their Corresponding Events

Symbol	Event	Symbol	Event
e	Click Back	k	Click Edit
l	Click Save	v	Click Add
u	Set Title	x	Set Number
y	Set Price	z	Set Description
r	Remove All	t	Remove Title
p	Remove Price	n	Remove Number

In total, 12 different types of mutants are acquired at code and model levels. Table 7.1.3 gives the semantics of these mutants for the "Special" module, including mutation operators utilized for the generation of these mutants. GUIs under test enable only this number of faults at a model level based on experimentation for a mutant generation. The FSM model of the "Special" module permits only the faults using "Add", "Update", "Edit", and "Save" events due to the functionalities of these events. For example, "Add" and "Add" events permit

"Add with empty input boxes" and "Update" permits "Update with empty input boxes". Note that the number of possible mutants at the code level can be more than the number of modeled mutants related to the utilized mutation operators. The number of modeled mutants is determined by the GUI under test and its model where each mutant model needs to have a semantic as listed in Table 7.1.3. So, there isn't any correlation between the number of states or transitions in the FSM model.

In this case study, it is only focused on mutants that the FSM can acquire. Hence, the mutant and test generation were carried out on the FSM model, but the test execution and selection steps were carried out on the code level in this study.

Table 7.2.2: Mutant Semantics for MBIT

Mutant	Semantics	Mutation Operator(s)
1	Add with empty input boxes	OP,IO
2	Update with empty input boxes	OP,IO
3	Add with empty Number of Packages	OP,RO
4	Add with empty price	OP,RO
5	Add with empty title	OP,RO
6	Update with empty title	RO, OP
7	Update with empty price	RO, OP
8	Update with empty number	RO, OP
9	Add click does not respond	OP, RO
10	Edit click does not respond	OP, IP
11	Add and Edit click does not respond	OP, IP
12	Save button move to initial state	OP, IO, RO

In this step, the JFLAP tool is used to transform original and faulty FSMs into REs. Then, the PQ-Analysis [81, 85] tool is applied to these RE models to obtain CTs.

The original (supposedly fault-free) RE is provided below. The RE in (21) is converted from the FSM model and contains symbols that are embedded in the events provided in Table 7.1.2. The RE in (21) is shortened to fit the page.

$$[(ve)^*(knl(el)^*exl + ... + ky(xl + l) + kxl + kl)^*] \tag{21}$$

The PQTestGen [56] tool acquires test suites using each CT of both original ((supposedly) fault-free) and mutant (faulty) RE models. Finally, the PQTestGen tool utilizes test compaction to obtain the final form of the test suites by removing redundant sequences. Those sequences are the ones for the test execution part of the study, which operates on Selenium [5] test automation.

$$t1 = \text{"klktleulkl"}, t2 = \text{"klknlexlkl"} \tag{22}$$

$$t3 = \text{"yxzveuvkl"}, t4 = \text{"yuzvexvkl"} \tag{23}$$

For instance, test sequences are given above in (22) and (23) are acquired from RE given in (21) by means of the PQTestGen tool.

7.2.2 *Testing*

A Selenium test script is used for test execution. The Selenium script runs all test sequences on each original and faulty GUIs that are "Special" and "Additional" modules of the ISELTA. Then, test scripts result in either "Pass" or "Fail" for each test sequence. They will then group those to analyze them and decide the required set of test cases for constructing the MBIT suites. These suites agree with the conditions of the ideal testing.

In this thesis, 12 different mutants for the ISELTA website's "Special" and "Additional" modules are obtained, and 12 different test suites from these mutants are acquired for the NT.

Selenium is used for test automation, which enables us to execute test sequences automatically on the web-based software and collect the test execution results. It contains two parts, which are the Web-Driver and the IDE. The Web-Driver provides functionality with Java and Python programming languages for automation of the test execution. The IDE contains an easy-to-use interface, including plugins for specific internet browsers and simple record-and-playback of interactions with the browser. Too many commercial or non-commercial web or mobile test automation tools exist, especially for test execution. Selenium is selected due to its robustness, simplicity, and popularity among the scientific community. Besides Selenium, Sikuli [6] is also an open-source and robust solution. Sikuli is more useful for black-box testing when there is no access to the system's internal components or source code of the system under test. Sikuli uses image recognition powered by OpenCV to capture GUI events. Both Selenium and Sikuli can run various internet browsers from different vendors. Selenium is more community-driven and supported among other testers. Selenium can be easily adapted to MBIT methodology for test execution.

5 Selenium Test Automation, https://www.selenium.dev/
6 Sikuli Test Automation, http://sikulix.com/

7.2.3 *Experimental Setup For Graphical User Interface Programs*

The proposed approach was evaluated through the ISELTA "Special" and "Additional" modules with respect to experimental studies. For this evaluation, 24 mutants of the ISELTA "Special" and "Additional" modules (see Table 7.2.3) at code and model levels were obtained. The list of these mutants is provided in Section 7.2.1 with details. To show the presence and absence of the faults, the evaluation is carried out in this section. Considering the presented general methodology, test generation is only one of the stages. However, in the evaluation step, test generation has become a priority.

For the FSM models used within the scope of experimental studies to comply with the definition of model correctness proposed in Section 6.2.2, the test suites obtained from the original model within the scope of Criterion 6.1 were also run on the original GUI system, and it was observed that all test sequences passed the test successfully. Similarly, each test suite generated from mutant FSM models is executed on the corresponding mutant GUI system based on Criterion 6.4 to avoid wrong model utilization.

Test generation is optional for the current methodology. Thus, the general methodology can be realized by changing the test generation stage. However, an approach that provides coverage of all modeled faults has been proposed in this study. An approach that produces random test generation has been developed and used for this evaluation. Also, an industrial-level model-based testing tool called Graphwalker is adapted to the methodology and utilized for evaluation. Thus, the extent to which the overall approach is effective and appropriate to test for the presence and absence of faults has also been evaluated.

In the literature, the mutation score, fault coverage, test suite size, test generation time, and test execution time metrics are utilized to evaluate the approaches proposed for software testing (see Section 3.2). The most important and preferred of these is fault coverage. Other important metrics are the time for test generation, test execution, and test suite size. All of these metrics were considered to evaluate the current methodology.

By means of Selenium, an open-source test automation tool, the test execution process is automated.

Table 7.2.3: Mutant Profiles for GUIs

Fault Type	Quantities		
	Special Module	Additional Module	Total
Functional Fault	12	12	24

For example, in mutant number 3 (Add with empty Number of Packages input box) for the Special module, in the original system, the user cannot add a new form to the system if he/she does not fill the "Number of Packages" input box. However, to create a mutant and a test sequence for this mutant, the "Number of Packages" input state from the FSM is removed in mutant number 3. Because the "Number of Resource" input state is removed from the FSM, the mutant test file does not contain the test sequence for adding the "Number of Packages" action. When the test suite is executed on the original GUI under test, the system fails in some test sequences for adding the new special form action. This is because the original system actually expects the "Number of Packages" input box to be filled to save the form to the system database. These validation points in the application lead to several failing test sequences in each mutant because those parts from the mutant are deliberately removed. Later, these failing sequences are used to assert the required parts of the GUI under test.

7.3 RESULTS

The MBIT approach is proposed for HDL and GUI programs. The results of the experimental evaluations for HDL and GUI programs are presented in the following sections.

7.3.1 *Results For HDL Programs*

The results from the ideal test experiment are collected in Table 7.3.1 and Table 7.3.2 for the negative and positive testing with respect to three methods and five techniques for SD and TLC, respectively. The proposed method for MBIT is a context-based test generation approach called PQTestGen utilizing context-related coverage criteria [100]. The other approaches are regular expression-based test generation, RETestGen [103], and context-based random test generation, PQRTestGen [104], for which users have the option to set a specific symbol coverage ratio. Thanks to this option, the thesis sets coverage ratio to 90, 80, and 70 values to utilize three more techniques for experimental evaluation by using PQRTestGen [104]. It cannot provide a practical result once the ratio is set to 100. Also, test suites were not generated in some cases even when the coverage ratio was set to 90. These cases are not considered in the evaluation.

The reader can refer to Chapter 4 for the details of the approach implemented in the RETestGen tool.

As seen in Table 7.3.1 and Table 7.3.2, PQTestGen has the highest fault coverage. In addition, the mutation score of test sets produced for positive testing with PQTestGen is 1. Therefore, it was possible to generate an ideal test suite for all mutant models from the test suite generated with PQTestGen for TLC and SD. For TLC, after removing the equivalent mutants from the total set of 82 mutants, all remaining

Table 7.3.1: Results of the Positive and Negative Testing for SD

	PQTestGen		RETestGen		PQRTestGen90		PQRTestGen80		PQRTestGen70	
Symbol Coverage	100		100		90		80		70	
Mutation Score	1	0,94	0,94	0,78	0,94	0,78	0,89	0,56	0,61	0,5
Fault Coverage (%)	100	94	94	78	94	78	89	56	61	50
Test Suite Size (Symbols)	64	52	66	51	68	35	28	18	21	14
Test Generation Time (ms)	7	7	21509	20445	9	6	4	4	4	4
Test Execution Time (ns)	116	156	130	156	175	108	97	64	107	73

mutants were killed. It is not possible to kill equivalent mutants by the proposed or other compared methods. The reason for the fault coverage value not being 100 is that equivalent mutants are included in calculating the fault coverage. The method with the best fault coverage and mutation score after PQTestGen is that of RETestGen with TLC. For the PQRTestGen method, the fault coverage and mutation score values decrease as the symbol coverage value decreases. A very small difference that does not confirm this hypothesis is the value obtained when the symbol coverage value is set to 80 and to 70 for the positive testing.

As part of the experimental work for TLC, the equivalent mutant models of TLC are deliberately seeded to see how the methodology behaves. These equivalent mutants are artificially generated from the FSM model of the TLC, and then their code-based mutant programs can also be obtained. Then, the mutant models and programs are integrated into the experimental study. However, the current methodology is not able to detect the equivalent mutants. To have a more accurate mutation score, these deliberately seeded equivalent mutants are isolated from the calculation of the mutation score.

Table 7.3.2: Results of the Positive and Negative Testing for TLC

	PQTestGen		RETestGen		PQRTestGen90		PQRTestGen80		PQRTestGen70	
Symbol Coverage	100		100		90		80		70	
Mutation Score	1	0,95	0,94	0,89	0,83	0,89	0,80	0,83	0,82	0,79
Fault Coverage (%)	90	90	84	79	73	79	71	73	72	70
Test Suite Size (Symbols)	157	199	169	195	99	104	90	84	67	68
Test Generation Time (ms)	21	19	23501	33213	25	9	14	7	13	4
Test Execution Time (ns)	363	285	478	430	298	238	282	222	222	197

Based on the experimental study, PQTestGen and RETestGen can kill mutants for "hard to detect faults", which is not the case with the other techniques. However, RETestGen takes longer time for test generation, see Table 7.3.1 and Table 7.3.2, and is unable to kill the mutants that PQTestGen can kill. After "hard to detect faults", the "condition stuck at true/false faults" are the most difficult to detect faults. The "output bit stuck at 0/1 faults" are easily detected if their corresponding output behavior is included in the test suite.

For the RISC-V processor, the PQTestGen achieves the maximal fault coverage and mutation score for both PT and NT. RE-TestGen also has the maximal scores (fault coverage and mutation score) for

Table 7.3.3: Results of the Positive and Negative Testing for RISC-V Processor

	PQTestGen		RETestGen		PQRTestGen90		PQRTestGen80		PQRTestGen70	
Symbol Coverage	100		100		90		80		70	
Mutation Score	1*	1*	0,82	1*	0,74	0,89	0,67	0,83	0,58	0,79
Fault Coverage (%)	100*	100*	82	100*	74	91	67	79	58	67
Test Suite Size (Symbols)	522	464	3405*	3036*	158	167	148	152	110	116
Test Generation Time (ms)	49	48	41928*	26939*	25	24	22	23	20	20
Test Execution Time (ns)	359	345	1831*	3636*	203	332	98	314	81	134

NT. The maximal scores are 100 for fault coverage and 1 for mutation score. However, RE-TestGen results in more than 6 times larger test suite and more than 100 times longer test generation time, leading to more than 6 times longer test execution time. When the coverage criterion was set to 100, PQR-TestGen could not generate the test suites in any time interval that was set for about 1 hour. Even in 1 hour, PQR-TestGen generates at least 500 MB of data containing mostly redundant test cases in case of setting the coverage criterion to 100. Therefore, the coverage criterion was set to 90, 80, and 70 to generate different test suites for evaluation. The coverage criterion was set to 100 for PQTestGen and RE-TestGen, directly affecting the fault coverage and mutation scores. This explains why PQR-TestGen also has a lower fault coverage and mutation score due to the effects of the symbol coverage. However, PQR-TestGen utilizing a random test generation algorithm is very effective for achieving a reasonable fault coverage quickly.

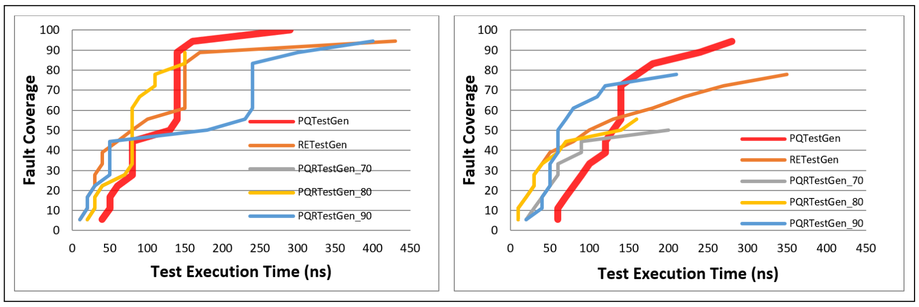


Figure 7.3.1: Fault Coverage and Test Execution Time Curve for SD (left diagram is for positive and right diagram is for negative testing)

Figure 7.3.1 and Figure 7.3.2 show cumulative distribution curves of the methods used for fault coverage with respect to test execution time for positive and negative testing for TLC and SD cases, respectively. All of the methods achieve at least 70% fault coverage in 400 nanoseconds for TLC. PQTestGen reaches maximum fault coverage in about 1,300 nanoseconds (positive testing) and 700 nanoseconds (negative testing). For the SD case, it is about 300 nanoseconds in positive and negative testing.

Figure 7.3.3 shows cumulative distribution curves of the methods used for fault coverage concerning test execution time for positive

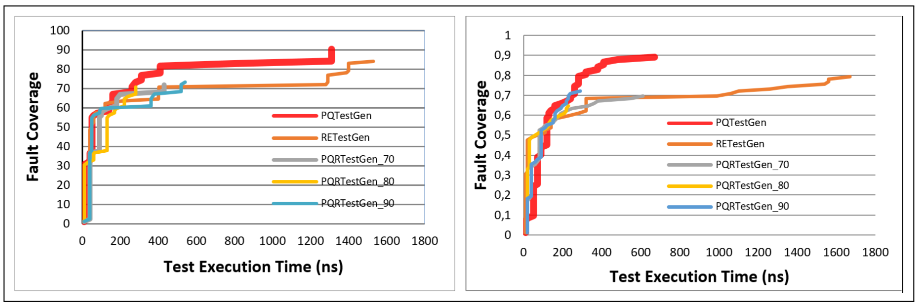


Figure 7.3.2: Fault Coverage and Test Execution Time Curve for TLC (left diagram is for positive and right diagram is for negative testing)

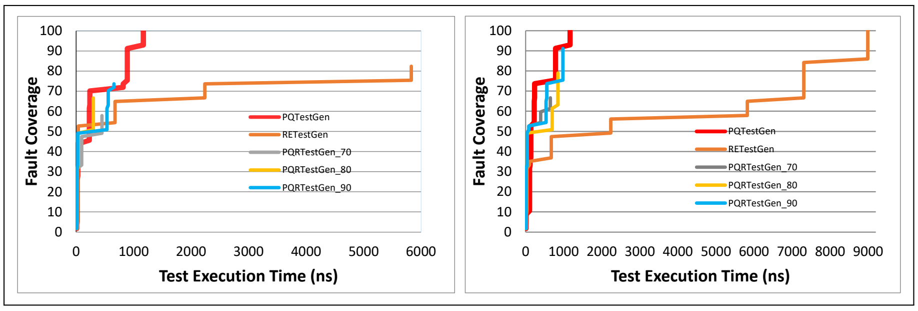


Figure 7.3.3: Fault Coverage and Test Execution Time Curve for RISC-V Processor (left diagram is for positive and right diagram is for negative testing)

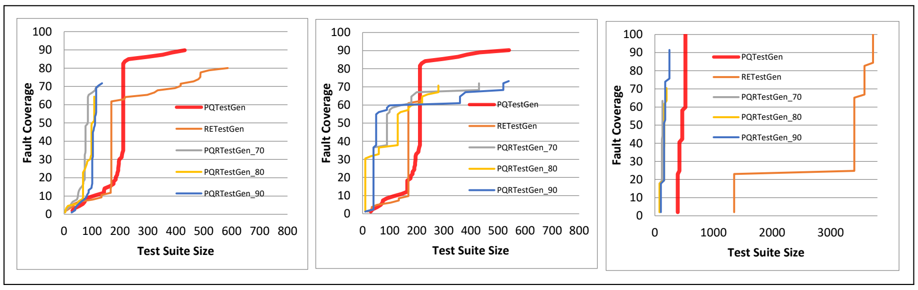


Figure 7.3.4: Fault Coverage and Test Suite Size Curves (left diagram is for SD, middle diagram for TLC, and right diagram is for RISC-V)

and negative testing for the RISC-V processor. PQTestGen achieves the highest scores in about 1000 ns for both PT and NT. RE-TestGen has the second highest fault coverage. However, it requires a longer test execution time to achieve its current scores.

The fault coverage and test suite size relation are presented in Figure 7.3.4 for SD, TLC, and RISC-V case studies in order from left to right only for NT. Since the test suite is fixed, this relationship is not provided for PT. Interestingly, the PQTestGen curves for SD and TLC are very similar and achieve the highest fault coverage for about 400 and 600 test cases for SD and TLC, respectively. The curves for RISC-V are very separated due to the different algorithms each tool utilizes. RE-TestGen needs a larger test suite size to reach the same fault coverage as PQTestGen.

The results are collected from the automatized processes. The manual effort is neglected from the results, which takes more than the total time for the test generation and execution. The mutant generation requires the highest manual effort.

Based on the results, the proposed methodology with PQTestGen is effective in showing the presence and absence of faults using positive and negative testing in the scope of the model. Finally, the Test Suite Analyzer was utilized to collect MBIT test suites from the results of PQTestGen for TLC and SD.

7.3.2 *Results For Graphical User Interface*

Together with the mutation score, fault coverage, test suite size, test generation time, and test execution time metrics for evaluation, Table 7.3.4 and Table 7.3.5 are provided with a comparison of two different configurations of PQRTestGen, namely, PQRTestGen100 [104] and PQRTestGen60 [104], and Graphwalker [49]. The symbol coverage criterion is set to 100 and 60 for the random test generation tool, PQRTestGen100 and PQRTestGen60. Using its visual editor, a new FSM model is created to adapt the Graphwalker [49]. Then, Graphwalker is run to generate test sequences by setting symbol coverage to 100. After running Graphwalker for about one hour, it is terminated due to excessive memory usage, which resulted in an enormous output file. Experimentally, the coverage value was decreased, and decided that 90 percent coverage was the optimum value. The RETestGen [103] tool is utilized for test generation. However, RETestGen resulted in excessive-size test suites, which could not execute on the GUI under test at acceptable times. Therefore, RETestGen from the experimentation is neglected. To eliminate randomness on evaluated metrics, each tool ran using a random test generation algorithm 10 times and selected average test suite size among others with their corresponding metrics.

The reader can refer to Chapter 5 for the details of the approach implemented in the PQRTestGen tool.

Table 7.3.4: Results of the PT and NT for Special Module

	PQTestGen		PQRTestGen100		PQRTestGen60		Grapwalker	
Symbol Coverage	100		100		60		90	
Mutation Score	1*	0,92	0,75	0,75	0,58	0,75	0,92	0,92
Fault Coverage (%)	100*	92	75	75	58	75	92	92
Test Suite Size (Symbols)	412	486	228	209*	193	154	767*	594*
Test Generation Time (ms)	249	178	263	203	216	157	3870*	3897*
Test Execution Time (s)	207	180	84	72	120	71	283*	262*

PS: Positive Testing (left), NT: Negative Testing (right), ms: milliseconds, s: seconds

Table 7.3.5: Results of the PT and NT for Additional Module

	PQTestGen		PQRTestGen100		PQRTestGen60		Graphwalker	
Symbol Coverage	100		100		60		90	
Mutation Score	1*	1*	0,58	0,75	0,5	0,58	0,92	0,83
Fault Coverage (%)	100*	100*	58	75	50	58	92	83
Test Suite Size (Symbols)	486	412	215	215	180	160	554*	508*
Test Generation Time (ms)	254	181	275	218	217	158	3856*	3676*
Test Execution Time (s)	104	88	47	47	40	40	133*	132*

PS: Positive Testing (left), NT: Negative Testing (right), ms: milliseconds, s: seconds

As provided in Table 7.3.4 and Table 7.3.5, PQTestGen attained the highest coverage of faults. The details of the PQTestGen tool are elaborated in Section 7.5. Moreover, PQTestGen has the highest mutation score for the PT, which is 1. Therefore, MBIT test suites are acquired for Special and Additional GUI under test using PQTestGen for the entire set of mutants. For the PQRTestGen method, the symbol coverage value also decreases while the fault coverage and mutation score values decrease. However, the PQRTestGen method resulted in the same coverage for the "Special" module in the NT.

Graphwalker resulted in a larger test suite than other techniques, and its test generation time is about 15 times higher than others. The reason for the larger test suite is the random test generation algorithm utilized by Graphwalker. Graphwalker is run on the command-line interface (CLI) and calculates test generation time using its jar file. Execution of jar files may explain the excessive test generation time of Graphwalker, among others. Except for Graphwalker, PQTestGen yielded the largest test suite results among others. This is because the test generation algorithm used by PQTestGen includes even each repeating symbol in the coverage value as if it were a different symbol. Thus, while some redundant symbols are included in the test suite, this situation directly affects the fault detection capability.

The cumulative distribution curves for fault coverage and normalized test execution time is given in Figure 7.3.5 and in Figure 7.3.6 for the "Special" and the "Additional" modules, respectively. All the techniques have at least 58 fault coverage in around 225 seconds for both GUIs under test. In about 207 seconds (PT) and 180 seconds (NT) for the "Special" module, PQTestGen achieves the maximum fault coverage. It is around 104 seconds and 88 seconds in the PT and NT, respectively, for the "Additional" module. To calculate normalized test execution times, the minimum (min) and maximum (max) values of all test execution times for the respective GUI were found. Then, the normalized form of the x value as (x-min) / (max-min) was calculated. The normalization process resulting from the proportional difference between PQTestGen and PQRTestGen test execution times is needed. Thus, Figure 7.3.5 and Figure 7.3.6 were obtained.

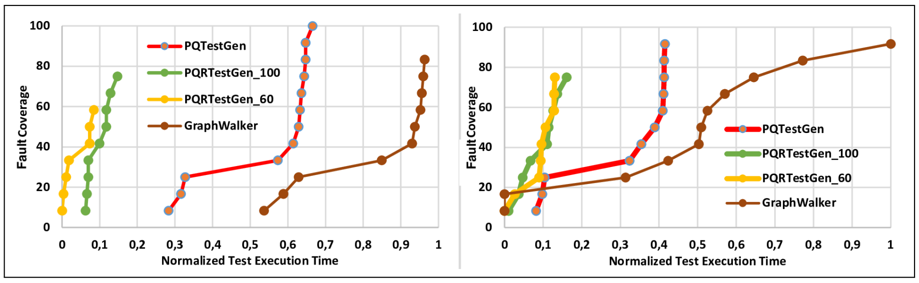


Figure 7.3.5: Fault Coverage and Normalized Test Execution Time Curve for Special Module (left diagram is for the PT and right diagram is for the NT)

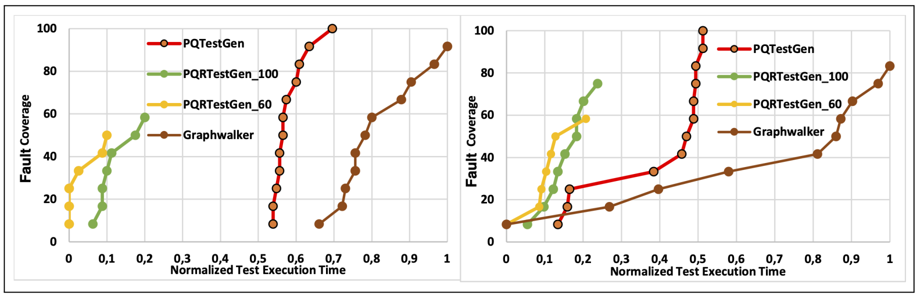


Figure 7.3.6: Fault Coverage and Normalized Test Execution Time Curve for Additional Module (left diagram is for positive and right diagram is for negative testing)

These results are collected using automatized processes. The manual effort is neglected. Usually, the manual effort requires more time than the automatized process. The highest manual effort is the mutant generation.

Based on the collected results, the proposed approach efficiently shows the presence and absence of faults in the model's scope. Finally, using a Selenium test script, MBIT test suites are automatically collected for the "Special" and "Additional" modules.

The Web-Driver part is integrated into the current framework. Selenium is used for the test execution and selection steps performed using the Web-Driver, which contains generic Java test scripts to execute test sequences from PQTestGen and PQRTestGen automatically. Then, it collects the "pass" or "fail" results.

7.4 EVALUATION

The following sections present the evaluation results for the MBIT concerning research questions for HDL and GUI programs.

7.4.1 *HDL Programs*

This section analyzes the results of the case studies and discusses the aspects of the selection of test and mutant generation techniques within the proposed methodology. Moreover, it discusses the evaluation of the algorithms and procedures utilized in the methodology. The research questions defined in Chapter 1 are examined to check whether those questions have been answered satisfactorily. Finally, the threats to the validity of the proposed methodology are discussed.

7.4.1.1 *Test and Mutant Generation Techniques*

This thesis uses the PQTestGen tool for test generation, as it is effective with respect to its fault detection ability by means of exploiting the RE. This tool uses the context tables generated with PQ-Analysis by analyzing the RE. The tool's effectiveness comes from these context tables, which enable to remove the ambiguity in the RE (see details in Section 3.3). This improves the ability of the test sequences to detect faults and increases the number of mutants killed. The reason is that this thesis addresses showing the absence of HDL faults (as part of the ideal test), and to show the absence of a fault, it should be modeled in the mutants of the FSM. Therefore, it requires the application of specific mutant operations to the original FSM. As a result, the automatic generation of mutants, which produces random mutants, is not useful for this purpose.

7.4.1.2 *Evaluation of The Proposed Methodology*

The proposed approach is evaluated considering the computational complexity of the different steps required for applying ideal testing, including model construction, model mutation, model conversion, test generation, pre-selection, and test suite construction. It is worthwhile to know that among these steps, the model mutation is done manually (discussed in Section 6.1.3); the rest of the steps will automatically be carried out by means of the available toolchain.

According to the procedure described in Section 7.5, the model construction step needs two iterations to extract an FSM out of an HDL program. It is assumed that the HDL program has N states. The first iteration of the model construction procedure is to list all states (with the complexity of N), and the second iteration is to find the transitions between the states, which have the complexity of N^2 in case the FSM is complete. Therefore, the total complexity for the whole procedure in the worst case is $O(N+N^2)$, which equals $O(N^2)$.

As discussed earlier, the model mutation is done manually for specific faults and is mapped to mutant HDL programs. Therefore, it is not considered in the calculation of the computational complexity of the proposed methodology. However, the number of mutants (let this be M) that are provided manually for the faults is important in the calculation for its effect on the complexity of further steps, as the following steps will be repeated for each of the mutants.

The model conversion step can be implemented using different algorithms, such as state elimination [90] or the Brzozowski method [96]. However, the best result can be achieved without considering memory/space limitations with the state elimination algorithm in polynomial time [105].

The test generation step includes 2 sub-steps, namely model analysis using PQ-Analysis to create context tables and test sequence generation. The model analysis step can be done with a complexity of $O(N^2)$. The test sequence generation is implemented based on the context tables and using a breadth-first search algorithm, presented in Section 6.1.3 with the algorithm in Listing 7, which results in a complexity of $O(N^2)$.

Finally, pre-selection and test suite construction steps are applied on all test sequences; let their number be K, each of which can have at most N-1 test symbols, as the loops and circles of the FSM are not considered while traversing it for test generation. So, the complexity of the test execution will be $O(K \times N)$ in the worst case.

Table 7.4.1: The Complexity of the Used Algorithms

	Automatic	Manual
Model Construction (FSM-Extractor)	$O(N^2)$	N/A
Model Mutation	N/A	O(N) with respect to N states or O(M) with respect to M transitions using mutation operators
Model Conversion (JFLAP)	$O(N^3)$	N/A
Model Analysis (PQ-Analysis)	$O(N^2)$	N/A
Test Execution (Test Bench)	O(K*N)	N/A
Test Selection (Test Suite Analyzer)	$O(K \times N)$	N/A
Total	$O(N^2 + M \times (N^3 + 2 \times O(N^2)+2 \times K \times N)) = O((M \times N^3) + (M \times K \times N))$	

Table 7.4.1 summarizes the above-mentioned calculations. As it can be seen in this table, the total complexity of the proposed approach is $O(M \times N^3+(M \times K \times N))$ at the worst case (where M is the number of mutants, N is the number of states and K is the total number of

test sequences from each mutant), as the model conversion, model generation, test execution, and test composition steps are applied for each of the M mutants.

7.4.1.3 *Checking The Research Questions*

This section examines the research questions according to the results achieved throughout this thesis.

HDL-RQ 1. Is it possible to apply the code-based ideal testing approach (proposed by Goodenough and Gerhart [2]) to model-based testing and exemplary to HDL programs that will be viewed as hardware specifications?

The proposed methodology, called MBIT, satisfies the requirements of ideal testing [2] that is supported by the theoretical evidence provided in Section 6.1.1. The experiments conducted using the case studies support the claim of satisfying the requirements of ideal testing with the HDL program. Thus, HDL-RQ1 can clearly be answered: Yes, it is.

- HDL-RQ 1.1 What kinds of HDL system can be addressed? Sequential, combinational, Cyber-Physical Systems, embedded systems? What are the borders?

In the proposed methodology, Verilog HDL is addressed, which models a sequential circuit at the behavioral level. However, the proposed methodology can be adapted to any sequential system that can be represented by behavioral models, such as user interfaces, for cyber-physical, interactive, and/or embedded systems. Behavioral models cannot represent the combinational circuits due to their stateless nature.

- HDL-RQ 1.2 What kinds of HDL faults can be targeted?

The fault types [13] in HDL are single output bit stuck-at 0/1, case bit stuck-at 0/1, condition stuck-at True/False faults, and their combinations.

- HDL-RQ 1.3 What are the outcomes of applying the ideal testing?

The proposed model-based method satisfies the reliability and validity requirements of the Fundamental Test Theory of Goodenough and Gerhart [2]. Also, it enables testing for the presence and absence of fault(s) in the scope of the HDL model.

HDL-RQ 2. What are the costs of applying ideal testing in terms of time and size of test suites (a set of test sequences), and is this approach scalable?

The cost of the proposed methodology is O(M×N^3+(M×K×N)) for N states, M mutants, K is the total number of test sequences from each mutant. Test time for positive and negative testing is given in Table 7.3.1, Table 7.3.2, and Table 7.3.3 for the SD, TLC, and RISC-V case studies, including the size of the test suite. Based on these findings, yes, the current methodology is scalable.

- HDL-RQ 2.1 How does this cost affect the scalability of the application of ideal testing?

The main bottleneck of the approach is the PQ-Analysis-based test sequence generation, which requires converting the models for analysis. However, the main steps are automated by means of the tool support given in Section 7.5. Moreover, future directions are discussed to cope with the scalability problems identified in Section 7.4.2.3.

- HDL-RQ 2.2 What are the complexities of the algorithms used?

The complexity of the used algorithms in the proposed methodology is given in Table 7.4.1.

7.4.1.4 *Threats to the Validity*

In this section, the possible threats to the validity of the proposed methodology are discussed with respect to the internal, external and construct validities.

Internal Validity

A model-based ideal testing methodology is proposed to validate HDL design by utilizing holistic [7] and mutation testing [9, 10]. To verify this claim, the theoretical evidence is already given in Section 6.1.1, in which the requirements for being an ideal test are proven. Moreover, the basic idea to support the proposed approach is given in Section 6.1.2. More importantly, the experiments are conducted on case studies in Section 7 for evaluation, which supports theoretical evidence by resulting in ideal test suites. In experimentation, randomly chosen faults are seeded into the model to address common faults in practice in the HDL. Then, the ideal test suites are executed on both original and mutant models. Results show that the ideal test suites reveal the presence of those faults by means of positive testing. To test the absence of the faults, the ideal test suites generated from the mutants are executed on the original HDL in negative testing. To this end, the experimental and theoretical evidence verify that the claim is correct, showing no threats to internal validity.

A critical question can be asked about the model used in the approach: How do you show that the model is correct? A faulty model would lead to corrupted tests. This question can ruin all of the model-based techniques developed over more than three decades and thus has satisfactorily been answered: Use any support to validate the model, for example, model checkers [106], and more importantly, involve end users as early as possible to extensively assess the model, long before you start with test generation and testing.

External Validity

To generalize the obtained results from this thesis, the proposed method must be applied to different case studies.

The Verilog HDL is selected due to its availability for fault models and simplicity than the others [107]. However, the SystemVerilog HDL is an industry-standard HDL, especially for verification. The proposed methodology is also applicable and valid for the SystemVerilog HDL appropriately defining and addressing design faults that are nearly the same as Verilog HDL faults.

The number of states in the FSM model may cause a problem with respect to scalability. The problem is already mentioned in Section 7.4.1.2, including complexity analysis that is N^2 in the worst-case scenario for test sequence generation from the indexed regular expression. Experimental results related to HDL-RQ 2 show that the problem is solved in 78 milliseconds for a case study with 10 states. However, for larger problem sizes, the response time can be unreasonable. To tackle this problem, scalability can be improved using the available techniques in the literature [71, 108, 109].

The proposed method only applies at the behavioral level HDL that implements a sequential circuit that an FSM can model. An FSM model is required to be extracted from this behavioral level HDL to proceed to further steps. The proposed method offers an automatic approach to obtaining an FSM model from well-structured HDL. However, a tester can manually extract an FSM model from either the specification or directly from HDL. The above-mentioned limitations can affect the generality of the proposed method in terms of external validity.

Moreover, a study can be addressed to adapt the current methodology for re-configuring it at run-time so the fault can be detected and fixed. These types of faults are called soft errors, such as single-event upset. A similar idea proposed by Iqbal et al. [110] can be used to apply for automated testing at run-time using a model-based testing approach.

The proposed approach addresses the following common fault types [12, 13]: single output bit stuck-at 0/1, case bit stuck-at 0/1, condition stuck-at True/False, their combinations at the behavioral level HDL. However, generating tests for HDL of hardware, and its HDL signals can take on quaternary values (0,1,X,Z), the thesis does not target stuck-open (stuck-at Z (High-Impedance state)) or stuck-short (stuck-at X) faults for especially bus related faults because it is not possible to model them at the behavioral level. This causes an external threat due to its generality.

Detecting bus-related stuck-open (stuck-at Z (High-Impedance state)) or stuck-short (stuck-at X) faults at the behavioral level requires looking at the bus-enabled signals in the FSM used for the representation of behavioral level HDL. To this end, this can be extended for this thesis by addressing these kinds of serious bus-related faults. For this, it can be first planned to simulate such faults to check their effects on output signals, which are directly represented at the transitions of the FSM. Then, this can have a direct representation at the FSM, which can be used as mutant models. On the other hand, its representation and definition at the behavioral level HDL code is needed to execute positive test suites. On the other hand, the different approaches [111, 112] can be utilized for bus-related faults represented by graph models.

This thesis only handles validation activities without considering timing issues that may lead us to another external threat. Even though the proposed methodology targets the HDL design at the behavioral level, the generated test suites can also be valid at the lower level (that is, Gate level) to address the lower-level faults. Testing activities at the lower level may require the consideration of timing behaviors that are neglected in the current methodology.

Construct Validity

The mutants at the model and code levels are generated manually, even though the original model, which represents supposedly fault-free HDL code, is constructed automatically. Therefore, the mutant developer presents a bias that might threaten the construct validity. To mitigate this threat, the existing mutation operators [67] in the literature are analyzed and adapted into the proposed methodology to minimize the bias by the mutant developer for the model mutation. Similarly, the existing fault models [26] (such as stuck-at 0/1 fault) are linked to mutations in the HDL program level while performing mutant generation.

7.4.2 *Graphical User Interface*

This section discusses MBIT for GUI programs concerning testing techniques and the selection of mutant generation based on experimental and theoretical evaluation. The research questions are answered, including the internal and external validity.

7.4.2.1 *Test and Mutant Generation Techniques*

The FSM and RE are used to acquire mutants and test suites, respectively. However, alternatively, it is possible to use others, such as the ESG or EFG, for these processes. The FSM and RE are adapted to this thesis.

The PQTestGen tool is used in this thesis to obtain test suites due to its effectiveness in detecting faults. The PQTestGen is an efficient tool that uses tables to eliminate uncertainty by using the right and left context of each symbol in the RE. This enhances the capability of fault detection.

7.4.2.2 *Checking the Research Questions*

The research questions provided in Section 1 are answered concerning theoretical and experimental evaluation within the thesis's scope.

1. GUI-RQ 1. Is it practically and theoretically possible to offer an ideal testing [2] approach for GUI testing? It is applicable based on the theoretical evidence provided in Section 6.1.1. Also, the experiments' result (given in Section 7.2) is that the conditions of being an ideal test [2] are satisfied with the GUI programs based on the "Special" and "Additional" modules of the ISELTA. Therefore, the answer to GUI-RQ 1. is yes.
   - GUI-RQ 1.1 What types of systems can be tested in this way? In this thesis, the GUI system of a computer's sequential programs and a sequential slice of them, representing an application by a combination of icons, menus, buttons, bars, boxes, and windows is addressed. However, this thesis applies to any mobile GUI program. Other types of computer programs, such as the GUI of game programs, are neglected, which requires the utilization of different abilities and methods.
   - GUI-RQ 1.2 What types of faults can be targeted with the proposed approach? The fault type addressed in the thesis is a functional fault [113] in the GUI under test that cannot deliver desired and expected behavior/function in case this fault occurs.
2. GUI-RQ 2. What is the cost of applying this approach to GUI testing? The computational complexity of the proposed approach is O(M * N3 + M * K * N) for N states, M mutants, and K is the total number of test sequences from each mutant. Table 7.3.4 and Table 7.3.5 are presented for the cost of the "Special" and "Additional" modules of the ISELTA case studies, including the size of the test suite. Since the proposed approach is generic, the computational complexities are the same for both HDL and GUI programs.

3. GUI-RQ 3. How is scalability affected? The CT-based test generation is the main bottleneck of the thesis due to the transformation of the models for analysis. Nonetheless, the main steps of using the toolchain are already automated.

7.4.2.3 *Threats to the Validity*

This section presents potential threats to internal, external, and construct validity, including mitigation methods.

Internal Validity

Similar to internal validity for HDL programs, there exists a point that needs to be addressed here related to model correctness. The mitigation method presented for HDL programs in Section 7.4.1.4 also applies to GUI programs.

External Validity

In the test composition part of the proposed approach, there is the step of running the test suites. Although these suites run automatically thanks to the Selenium tool, each test run can take a few minutes, as shown in the tables of Section 7. This time varies in proportion to the number of states in the model. Moreover, considering that the PT and NT are applied for each mutant, the total number of Selenium test operations will be the mutant number times 2. This can be a few hours, even in the small GUI form used in case studies of GUI programs. This approach will take much longer when applied in larger models with more mutants. For this reason, the most important external validity can be considered as this complexity problem.

To cope with complexity issues leading to state explosion problems, one of the GUI ripping methods [97, 98] can be utilized to obtain the model automatically. On the other hand, there is another solution [114, 115] to cope with huge models utilizing layered modeling methods that are well-suited for the hierarchical structure of GUI systems. These layers can be manually created by the tester or automatically extracted from a non-layered model using a community detection algorithm introduced by Silistre et al. [52] to mitigate the complexity thread.

This thesis addresses the detection of functional faults rather than other types of faults related to visual attributes (such as overlap or rendering problems [116]) as they are mostly utilized in the GUI of games. This may lead to external validity. Models functionally represent systems under test in the thesis. Testing visual attributes on the screen is unsuitable for the thesis due to the constructed model's use. Testing such visual attributes requires the utilization of white box code-based testing methods.

Construct Validity

The current methodology requires manual efforts for the model and mutant generation. The test expert carries out the model generation using the JFLAP tool. However, the test expert may construct the wrong model as an original model, as it is also addressed in the internal validity. In this way, the test expert manually constructs the wrong mutant models from the wrong original model. This is also considered a potential threat to construct validity. To overcome this threat, the generated test suites from an original model are executed on the original SUT and from mutant models on the corresponding mutant SUTs. Furthermore, it is checked that all test cases pass on these executions to ensure that the model and the SUT are equivalent concerning the generated test suites.

7.5 TOOL SUPPORT

The proposed methodology is supported by using a chain of dedicated tools to run the testing process automatically and, thus, to remove the manual effort as much as possible for both GUI and HDL systems. Note that PQ-Analysis and PQTestGen are applicable for both systems, and the rest are specific to HDL systems. To this end, a set of tools is used, from which some are developed in this thesis's scope, and others are from the literature and merged into this toolchain. The tools used in this thesis are shown in Figure 7.5.1. From these tools, PQ-Analysis [81, 85], PQTestGen [56] and Test Suite Analyzer are domain-independent tools, but Test Bench is a domain-specific tool and in the scope of this thesis it is based on an FPGA Test Suite. The complete tooling for HDL and GUI systems are provided in MBIT4HW [7] and MBIT4SW [8] repositories, respectively.

Moreover, the reader can refer to Chapter 4 and 5 for the details of the approach implemented in the RETestGen and PQRTestGen tools. These tools are developed within the scope of this thesis and are used in the evaluation, comparing them with the PQTestGen tool.

The sequence illustrated in Figure 7.5.1 shows the order in which the tools are used to realize the ideal test. These tools are elaborated underneath.

7 MBIT4HW, `https://kilincceker.github.io/MBIT4HW/`
8 MBIT4SW, `https://kilincceker.github.io/MBIT4SW/`

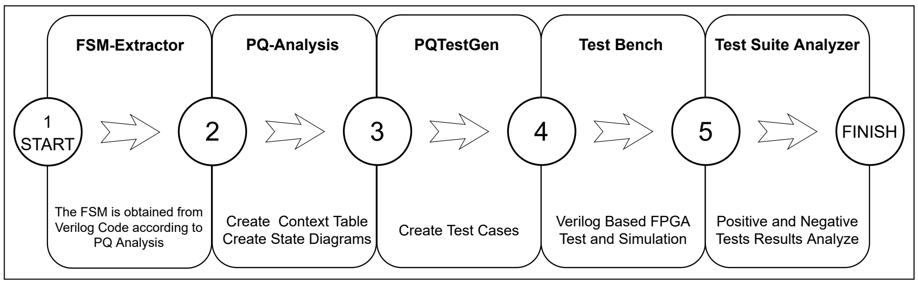

FSM-Extractor
PQ-Analysis
PQTestGen
Test Bench
Test Suite Analyzer
1 START
2
3
4
5
FINISH
The FSM is obtained from Verilog Code according to PQ Analysis
Create Context Table
Create State Diagrams
Create Test Cases
Verilog Based FPGA Test and Simulation
Positive and Negative Tests Results Analyze

Figure 7.5.1: Tools Used in the Proposed MBIT Methodology

Test Preparation

FSM-Extractor

The FSM-Extractor generates an FSM model from a given HDL program if any model is provided. The FSM-Extractor parses the HDL program and processes the code line by line to extract the states and transitions of the FSM model.The tool uses some of the variables defined in the HDL program to detect some specific patterns.For example, the tool employs "btn", "state", and "outputlightstatus" variables as the input, state, and output of the FSM model.The FSM-Extractor tool can be adjusted to accept any user-defined variables of an HDL program. The tool is implemented in Java programming language and accepts an HDL program (Verilog) as an input. The output file can be directly imported into the RE conversion tool.

PQ-Analysis

The PQ-Analysis tool [9] requires the RE form of the FSM representing the behavior model of the design. This FSM can be graphically designed by using any general tool such as the JFLAP [92] with which one can draw an FSM and generate a RE from it. The PQ-Analysis tool accepts the RE of an FSM and generates the context tables (or CT). PQ-Analysis comprises seven steps for which the details are provided in Section 3.3.

Test Sequence Generator (PQTestGen)

The PQTestGen generates test sequences using the output of the PQ-Analysis tool (that is, CTs). The PQTestGen is standalone software with a Graphical User Interface (GUI) for generating test sequences and saving these sequences as a test suite in a file. The tool uses a Breadth First Search (BFS) algorithm [117] to traverse all symbols

9 PQ-Analysis tool, `https://github.com/kilincceker/MBIT4HW`

and to select test sequences. The pseudocode of the algorithm implemented in the PQTestGen is given in Section 6.1.3. The tool creates test sequences by visiting all the cases, according to the previously obtained adjacency matrix, and their children who were not visited previously. Once the test sequences are generated, the compaction process is initiated to remove redundant test sequences that are overlapping with others.

Test Composition

Test Bench

Test Bench is a domain-specific code. In the scope of this thesis, the test bench is based on an FPGA environment. In this domain, Test Benches are pieces of Verilog codes that are used during FPGA simulations. Simulation is a critical step when designing a new hardware system. It is necessary to be sure that the FPGA design covers the system requirements. Furthermore, test benches are used to simulate the designs without the need for any physical hardware. Therefore, it is easy to test all system functions without hardware. To validate the HDL program, the test sequences require execution on the target system. This thesis's target system is an FPGA-based system (a Traffic Light Control and a Sequence Detector). The simulation environment [10] is the only commercial solution used in this work. However, the free version [11] of the tool that can be easily downloaded is used for the simulation at no cost. The Test bench executes these test sequences using Verilog HDL; see Figure 7.5.2 demonstrating the block diagram of the Test Bench procedure.

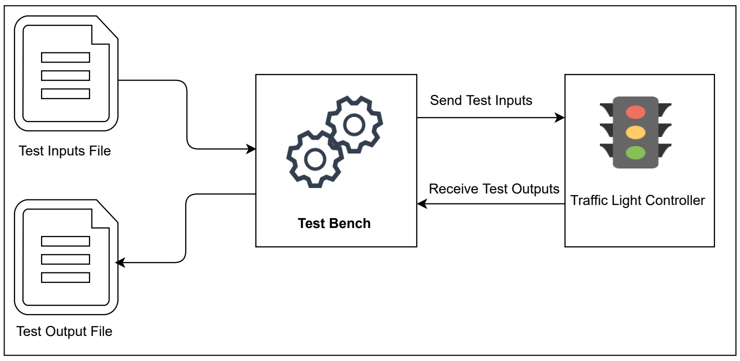


Figure 7.5.2: FPGA-based Test Bench

10 Xilinx Vivado, https://www.xilinx.com/products/design-tools/vivado.html
11 Vivado HL WebPACK, https://www.xilinx.com/products/design-tools/vivado/vivado-webpack.html

Test Suite Analyzer

The Test suite analyzer tool provides a selection mechanism to collect test sequences that satisfy the requirements of the MBIT. As given in Section 2.5, these requirements are reliability and validity, for which test selection criteria are defined. The test suite analyzer collects the test sequences that comply with these criteria, which results in MBIT test suites.

Additional details about the framework, including a user guide and a tutorial video along with the bundle for the toolchain, are publicly available online [12].

12 MBIT4HW, https://github.com/kilincceker/MBIT4HW

Part III

FURTHER PERSPECTIVES AND CONCLUSIONS

CHAPTER 8

EXTENDING SEQUENTIAL SYSTEMS FOR ACHIEVING FAULT TOLERANCE

In this chapter, an approach for extending sequential systems is proposed to achieve fault tolerance as a perspective of this thesis. In the following sections, a motivation for fault tolerance, related work, the proposed approach, a case study, results, and evaluations are presented.

8.1 MOTIVATION FOR ACHIEVING FAULT TOLERANCE

A growing spectrum of software systems safety- and security-critical applications can be observed today where an extremely high degree of reliability is required. Considerable efforts have been made to ensure protection against simple system faults (for example, single-bit stuck-at faults in memory or hardware circuits) using appropriate fault-tolerance (FT) techniques. Although such methods usually cover a large percentage of all potential error sources, several other potential error sources remain for software parts. Generally speaking, errors in the code or input data of components in a system may arise due to:

- erroneous data from the environment of the given system; for example, errors caused by faulty behavior of other systems or user errors,
- undetected hardware failures,
- undetected design errors in the hardware/software components.

Errors in software or hardware parts can cause failures or malfunctions. For example, Boeing's use of the Maneuvering Characteristics Augmentation System (MCAS) resulted in flights of Indonesian Lion Air (in 2018) and Ethiopian Airlines (in 2019) crashes, which caused 346 people to lose their lives [118].

In today's smart automotive, for instance, embedded systems can grow to 100 million lines of code running on 70 to 100 microprocessors [119]. Reliability is vital for systems of this level and importance. Otherwise, software or hardware errors encountered in these systems may cause very serious problems. Due to these post-production errors, companies recalling cars cause billions of dollars in losses to these companies.

The aim of this section is to make the errors that can cause such serious loss of life and money tolerable with low costs before production. To this end, the approach offers a uniform method that can be applied to sequential systems (SE), both software and hardware, to self-detect and self-correct faults.

As in previous sections, also this section applies sound, formal techniques of Automata Theory and Formal languages, more precisely, regular expressions to handle faults in SE with emphasis on (i) fault modeling to operationalize FT notions, (ii) analysis of a given SE to determine whether it possesses the required properties to handle faults modeled, and (iii) if not, the extension of the given system to enable it to possess the required, desirable feature, and revalidation of its original functionality. Note that faults in digital systems are root events that stem from deviations between the specified requirements and the implemented system behavior.

The method derived applies to the systems that are aimed to be protected against the considered faults, that is, functional and sequencing faults. The functional faults directly affect the desired operation of the system, which reaches the final operation but does not provide the expected outputs, likely to cause undesirable event(s). The sequencing faults affect the desired order of the operation in which the system cannot reach the final operation due to a system crash [113]. A case study presented in Section 8.4 will exemplify the benefits of the strategy. The acronym FT reads either as "fault-tolerant," "fault tolerance," "fault tolerating," etc., depending on the context in which it is used.

8.2 RELATED WORK ON FAULT TOLERANCE

This section provides related work and background information from general to more specific perspectives for fault tolerance for software and hardware sequential systems (sequential circuits).

Faults can be handled in three consecutive stages: detection, localization, and correction. If a fault in an SE can be unambiguously detected and corrected, then the system can perform the correction itself without needing control from outside. It is then said to be self-correcting in response to a fault of this type. The system is fault-tolerant (FT) if it can, despite a failure caused by a fault of the considered type, continue delivering the specified services, perhaps at a reduced level, but still to the satisfaction of the user (See Belli and Quella [120], page 20-49 for more details).

Many approaches have been proposed to make SE robust against errors for software: by introducing reset properties as to rollback, recovery lines, etc., and providing spare software mechanisms as to recovery block [121], n-version programs [122], etc. Recovery can then be accomplished if an error is detected during the run-time operation of the system. However, inherent to these strategies is the complexity of the utilized verification procedures as to acceptance tests, decision algorithms that usually deploy more or less heuristic and incomplete tests for errors [122, 123].

Several classical approaches using redundancy for hardware are popularly used in the aircraft and automotive industry in safety-critical applications [124]. As an example, the triple-modular redundancy (TMR) structure includes three redundant units and a majority voter selecting a unit that provides the same value as the majority of the three units.

Another example is the standby dynamic hardware redundancy (HWR), which uses duplex units where the unit and a spare are compared and selected by a multiplexer. Strano et al. suggested a Build-In Self-Test (BIST) diagnosis procedure of on-chip networks (NoCs) where stuck-at faults can be tested with a hardware overhead of less than 11% [125]. In another work, Strano et al. achieved hardware overhead of less than 10% using Single Instruction, Multiple Data (SIMD) accelerators used in systems on chips (SoC) [125].

Morgan et al. [126] presented a comparison between TMR with other three FT techniques, which are quadded logic [127], state machine encoding [128], and temporal redundancy [129]. The study concludes that these three techniques are more costly and provide less reliability than the TMR for both area and single-event upsets sensitivity.

Besides protecting and securing the entire system as TMR, securing some components of the system by using formal methods has recently become one of the preferred methods. The reason for this is the burden that occurs with TMR. Choi et al. [130], similar to Augustin et al. [131] offered selective fault tolerance for sequential circuits modeled by the finite state machine. In contrast to TMR, they tolerate faults on a subset of inputs with corresponding states in the FSM model by using a simple heuristic algorithm. The method provides the same degree of fault tolerance as TMR with reasonable area overhead. El-Maleh and Al-Qahtani introduced a fault tolerance method based on a finite state machine where a few states having a high probability of occurrence are protected [132]. El-Maleh [132] also proposed a method based on El-Maleh and Al-Qahtani [132] to optimize area and power and to preserve the original behavior of the system. Park and Yoo also proposed a fault tolerance method based on an encoding technique to secure and protect a set of states having high importance using FSM[133].

In order to increase the fault tolerance of a system, structural redundancy can be employed by adding extra components that can be activated if a fault is detected. This approach is similar to maintaining spare part storage, where the extra components act as backups for the system. With these extra components available, the system can continue functioning even if one or more of its primary components fail. Structural redundancy can provide an added layer of protection to ensure that the system can continue to operate without significant disruptions, even in the presence of faults. This redundancy approach can be applied to various systems, including mechanical, electrical, and software-based systems. On the other hand, functional redundancy adds extra components that are functionally similar. Those redundancies can be realized in a static or dynamic manner. Static functional redundancy is employed regardless of the presence or absence of faults, so it runs continuously. Dynamic functional redundancy is utilized on demand in the occurrence of a fault. The reader can refer to Belli and Quella [120] on pages 46-47 for more details.

All these approaches use system redundancy to enable FT. To our knowledge, there is no other approach considering the determination and implementation of static functional redundancy and structural redundancy on software and hardware systems. The approach employs static redundancy to perform the specified function during the whole time of operation.

In Section 8.5, a comparison table of selected different approaches provides an overview of the estimated hardware overhead and time consumption for hardware systems. As there seems to be no other procedure using structural redundancy aiming at fault tolerance, the comparison will be kept on hardware systems only.

8.3 SYSTEM EXTENSION TO ACHIEVING FAULT TOLERANCE

The proposed approach consists of three parts, as shown in Figure 8.3.1. In the first part of the approach, the given system, modeled by FSM, is converted to RE for analyzing whether the desired property is fulfilled or not and is designated as the second part using Brzozowski's well-defined algorithm [90, 96], especially for HS. In the final part, after determining redundancy, if necessary, the extended SE* or HS* is obtained using reconversion of RE*, for which the JFLAP tool is utilized.

The first and the third parts of the proposed approach are conversion and re-conversion stages from a system under test to RE and from extended RE to an extended system. For details of the conversion of FSM into RE and RE into FSM, the reader may refer to the book of Hopcroft et al. [79] (on page 83). In the second part, the model is analyzed and then extended if necessary. If it is determined not to extend, the system already comprises the desired property, which is defined as being FT in terms of being a self-detecting and self-correcting system.

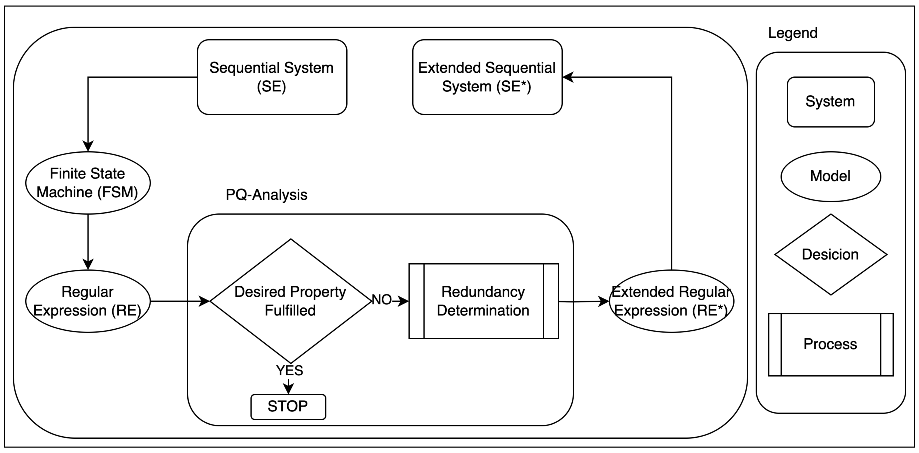


Figure 8.3.1: The Overview of the Proposed Approach for Achieving Fault Tolerance

Redundancy Determination for Achieving Desirable Properties

A RE is an algebraic term that is constructed and forms a string of symbols, which are interpreted as events that represent combinations of input and output signals. This term will be used to model and generate a set of elementary faults that can be corrected by inserting (I-correction), deleting (D-correction), or replacing (R-correction) symbols interpreted as events.. The elementary faults affect input/output behavior that can be modeled with FSM and RE. For the HS, these faults can be a bit stuck-at 0/1, a case bit stuck-at true/false, or their combinations with multiple faults at the system [134]. These faults are also called high-level faults [135].

A collection of fault prototypes will be used to analyze RE to determine whether the modeled HS is fault-detecting or fault-correcting, that is, if all faults of this type can be detected, localized, or corrected unambiguously.

Detection Step [84]: The input string is indexed forward and backward from left to right and right to left. If the input string is correct, it is converted to a backward and forward-coded form (as demonstrated in Section 3.3). During the coding of this form, the input string cannot be accepted and indexed due to an unacceptable symbol(s). This is where the detection step for an erroneous input occurs.

Localization Step [84]: A substring causes the machine to halt during forward and backward indexing. This is always the substring between the left-most acceptable symbol for back-indexing and the right-most acceptable symbol for forward-indexing. The correction hypothesis can be applied to this substring if necessary (refer to Section 3.3 for more details).

Correction Step [84]: A correcting symbol is injected into a position where it is between the left context relation of its successor and the right context relation of its predecessor in the correction area (refer to Section 3.3 for more details).

Simultaneously forward and backward indexing is called the coding of a regular expression. The coding is the basis of both tools needed for error treatment. As a second, more complex tool, the relations for the right context and left context r^{forw}, r_{back}, and l^{forw}, l_{back}, respectively, are used. They determine the symbols for every s^i or s_j that may appear, respectively, as its immediate successor or predecessor.

With compatibility and context relation tables, the faulty event sequence (FES) analysis is done by forward and backward tracing the indices for systematic detection and correction, respectively. FES refers to an illegal input sequence (unexpected) to the system, whereas event sequence (ES) is a legal and expected input sequence. The position in the sequence where a proposal might be applied is called a "correction position", and the symbol used for the correction is called a correcting symbol. Note that the terms (such as context table, compatibility table, forward/backward indexing, left/right context) are already explained and exemplified in Section 3.3, which will be also used in this section.

The hypotheses for the correction purpose are "insert(I)", "replace(R)", and "delete(D)" which are used to give possible proposals.

The index n defines the number of symbols. If n equals one, only one symbol is considered, and if n is bigger than one, more than one symbol as being a string is considered. Given any FES, when the error position is localized unambiguously, then the correction proposals are given considering their independence to deduce the self-correcting characteristic of a system that also defines self-detecting characteristics. Thus, the system is FT performing correction itself. The details of dependence or independence of proposals and ambiguous or unambiguous localization. Furthermore, comprehensive information can be found in the PhD thesis by Belli [81].

To add desirable properties to SE or HS in case needed, the necessary redundancy can be determined by a complete analysis of the FESs as test sequences. Symbols are extended by embedding them into others using well-selected symbols. The symbols that reveal ambiguity or make proposals dependent are selected, and then the symbol is extended by considering the context table to make it appropriate for its position. The extension refers to embedding at least one well-selected symbol into the appropriate left or right side of the chosen symbol for an extension so that the correction can only apply to one position [94].

These steps are already automated into the PQ-Analysis tool [1] that takes a RE as input and results in context and compatibility tables with redundancy determination if required.

1 PQ-Analysis tool, `https://github.com/kilincceker/MBIT4HW`

8.4 CASE STUDY

8.4.1 *Systems Under Consideration*

This section provides a case study for a sequential circuit that is a sequence detector (SD) system given with HDL at the behavioral level. The SD is implemented in HDL programming language and runs on XC3S100E from the Spartan-3 FPGA family.

Sequence Detector

The SD, represented in Figure 8.4.1, produces outputs (given in (24) after the slash (/)) for corresponding (primary) inputs (given in (24) before the slash (/)) and the outputs depend not only on the inputs but also secondary inputs or current states q0, q1, q2, q3, and q4.

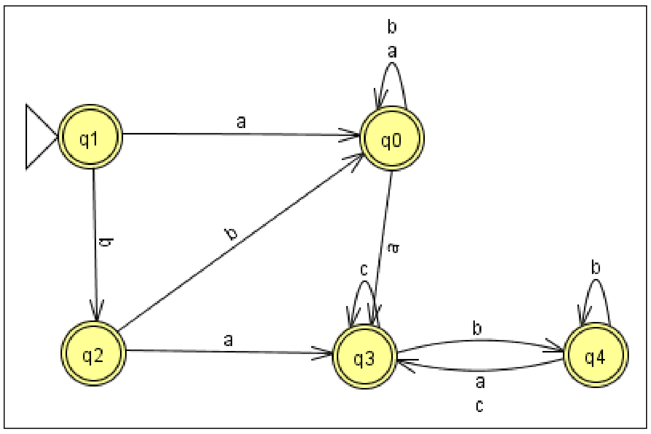


Figure 8.4.1: FSM of SD with Symbols

To represent the current and next states on the diagram, the states q0, q1, . . . , and q4 are defined, and the indices of states refer to the decimal equivalency of tree bits binary values. The corresponding input and output values for the current and next states are aggregated with the slash (/) symbol to label the transitions. Already initialized start state (q0) and final states are also represented with corresponding symbols embedding input/output signals.

For compact representation, the following settings allow concentrating the input $(i_1 i_2 i_3)$ and output $(o_1 o_2 o_3 o_4 o_5 o_6 o_7)$ information as transition events denoted by symbols instead of using binary values.

$$x = 001/0011101; y = 010/0011111; z = 100/0001101 \tag{24}$$

Applying the conversion algorithm from FSM, given in Figure 8.4.1, to RE for SD results in the following expression:

$$T_{SD} = [(ba(b+c)^*)^*(a+b)^*] \qquad (25)$$

The RE given in (25) is the same one used to explain steps of RE analysis in Section 3.3 in Example 3.4.

8.4.2 *Applying the Approach*

The uniform approach includes a complete analysis of already-attained RE and determining redundancy if needed. The stepwise PQ-Analysis of RE is already provided in Section 3.3, and then, using the analysis result, the redundancy determination will be given in the following sections for SD. Note that this section explains the redundancy determination part of the approach in case the desired property is not fulfilled, as shown in Figure 8.3.1 and the redundancy determination for SD case study has already been presented by Belli [85]. Therefore, this thesis focuses on evaluating the SD and presenting the results based on this evaluation in Section 8.5.

Sequence Detector (SD)

The reader can refer to for forward and backward indexing of the T_{SD} in (25) and for constructing compatibility, context tables, and their codings in Section 3.3 (Figure 3.3.1, Step 1-7).

For the SD, symbols revealing ambiguity or making the hypothesis dependent are detected to determine extension in proposals using the PQ-Analysis method originally introduced by Eggers and Belli [81, 84]. To systematically construct all correction areas of varying lengths, we need to generate all combinations of symbols that "do not fit together," meaning they cannot be adjacent in a correct sequence. For correction areas with a length of l=1, this involves placing each symbol between neighbors that are not valid left or right contexts for that symbol. The correction area and the I-R-D correction proposals are shown in Figure 8.4.2. Note that some of the results are taken from Belli [85] for more detailed analysis for SD. This subsection shows how to apply the approach to the SD. The results and evaluation will be presented in the following Section 8.5.

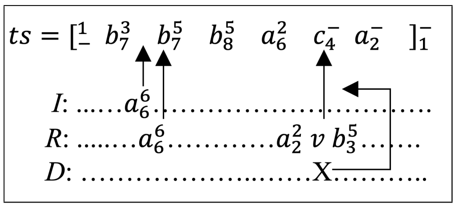

$ts = [^1_- \; b^3_7 \; b^5_7 \; b^5_8 \; a^2_6 \; c^-_4 \; a^-_2 \;]^-_1$
I: ......a^6_6..........................
R: ....a^6_6...........$a^2_2 \; v \; b^5_3$.......
D: ....X...........

Figure 8.4.2: Correction Area and R-D Corrections for SD [85]

The sequence (ts) shown in Figure 8.4.2 can be corrected in three alternative ways (as Step 8, following the steps as explained in Section 3.3).

- I-correction through inserting a_6^6 between b_7^3 and b_7^5,
- R-correction through replacing b_7^5 by a_6^6 or replacing c_4 either a_2^2 or b_3^5,
- D-correction through deleting c_4.

The correction proposals for the SD, such as symbol a_6^6 make the hypotheses I and R dependent. The system is not R-detecting because of the symbol a_2^2, which is not R-correcting because of the symbol b_3^5. This information enables selecting the symbols a_6^6, a_2^2, and b_3^5 for an extension. The location of these symbols is found using their indices and the T_{back}^{forw}. For example, inserting a_6^6 between b_7^3 and b_7^5 may cause the detection of a wrong sequence and result in a system failure. Then, the symbols are replaced with their extensions, as represented in Figure 8.4.3.

$$T_{SD} = [^1_5(b_{3+8}^{3+7}\ a_{2+6}^6\ (b_{3+7}^7\ +\ c_4^8)^*\ )^*(a_2^{2+6} + b_3^{3+5+7})^*]_1^4$$

$$a_6^6 \longrightarrow x\ a_6^6\ \mathrm{x} \qquad a_2^2 \longrightarrow y\ a_2^2 \qquad b_3^5 \longrightarrow \mathrm{z}\ b_3^5$$

$$T_{SD}^* = [^1_6(b_{12}^{2+11} x_{11}^6 a_{10}^9 x_5^{10}(b_{3+8}^{11} + c_4^{12})^*)^*(y_7^3\ a_2^7 + z_9^4 b_3^8)^*]_1^5$$

Figure 8.4.3: Extentions for SD

Consequently, the symbols a_6^6, a_2^2 and b_3^5 are extended using the symbols "xax", "ya", and "zb" respectively (Figure 8.4.3), based on the PQ-Analysis method (as Step 9).

To avoid undesirable or unnecessary redundancy, the number of symbols for extension should be as small as possible [94]. The extended RE for SD (Figure 8.4.3) becomes self-detecting and -correcting as the PQ-Analysis applied on T_{SD}^* (Figure 8.4.4).

$$ts^* = [^1_-\ b_8^2\ b_8^-\ b_{12}^-\ \ x_{11}^-\ a_{10}^-\ \ x_5^-\ c_4^-\ y_7^-\ a_2^-\ ]_1^-$$

I: ……………………………………...
R: ……………………………………..
D: …………………………………….

Figure 8.4.4: Correction Area and R-D Corrections for extended SD [85]

Due to ambiguity in SD between given symbols representing binary patterns, the system may lead to detecting wrong patterns caused by behavioral level bit stuck-at faults. For instance, a sequence detector integrated into a flame detector recognizes specific input patterns gathered from sensors to warn people to avoid possible flames.

This extended RE contains redundant symbols. Therefore, this may lead to non-functional equivalence. However, these redundant symbols are guarding symbols. For example, the symbol a is replaced with "xax" to secure the symbol "a" with the symbol "x". In this way, the system is aware of distinguishing this symbol from other symbols in the system. This ambiguity between the same symbols is removed thanks to the extended version in Figure 8.4.3.

The construction of the forward and backward indexing for extended SD and its codings and context table are included in the public repository (as Step 10) [2].

The impact of the correction proposals (Figure 8.4.4) enables the exclusion of conflicts shown in Figure 8.4.2 (as Step 11).

The cost of redundancy can be determined two-fold (as Step 12). One is related to the length of the T^*_{SD}. For length, the T^*_{SD} contains four additional symbols. This leads to 0.36 percent redundancy for RE in Figure 8.4.3. However, the VHDL implementation costs less redundancy (refer to Section 8.5).

The second redundancy concern is the execution time. According to the T^*_{SD}'s manual (static) analysis, the execution time redundancy is about 0.23 percent. For precise measuring, a dynamic analysis is required to compare the original and extended VHDL implementation.

8.5 RESULTS AND EVALUATION

SD and SD* were implemented in VHDL programming language and run on XC3S100E from the Spartan-3 FPGA family for both SD and TLS. Reader can refer to the GitHub repository [3] for the implementation of both SDs. Xilinx ISE Project Navigator was used for the design, implementation, and synthesis of circuits. A comparison of the circuits SD and SD* is given in Table 8.5.1 calculated from synthesis reports.

To employ the TMR method for the comparison, two more copies of the SD case studies are made, and a voter for the selection of the majority of the outputs to achieve SD* is developed. SD* is run on XC3S100E from the Spartan-3 FPGA family using Xilinx ISE Project Navigator to measure redundancy compared with SE. The Xilinx tool automatically generated Lookup Tables (LUTs) using synthesis reports and provided a number LUTs. The synthesis operation was also applied to SD to measure the required redundancy for TMR with respect to the number LUTs and time overhead.

For structural BIST, a custom LFSR (Linear Feedback Shift Register) system is used for test generation into the system itself and then

2 PQ-Analysis Results for extended SD, https://github.com/kilincceker/RD4FT/blob/main/PQ-results-SD-extended.txt

3 Sequence Detector VHDL implementation, https://github.com/kilincceker/RD4FT

developed a comparator for checking the response of the systems to obtain SD* which is synthesized using the Xilinx tool to generate a report for number LUTs. The synthesis operation was also applied to SD to compare with SD* with respect number of LUTs and time overhead.

Table 8.5.1: Results For the Current Approach with TMR-Method and Structural BIST

	Current approach	TMR-Method	Structural BIST
Hardware Overhead in %	~2	200	< 11
Overhead time consumption in %	~4.2	73	~0

Additionally, TMR and structural BIST are compared with the current approach in terms of estimated hardware overhead and overhead time consumption in Table 8.5.1.

Our approach has the lowest hardware overhead value, about 2. While its overhead time is about 4.2, BIST's overhead time is almost 0 for both SD and TLS. Note that structural BIST provides only fault detection but no fault correction. The TMR method needs much more overhead than the current and structural BIST approaches.

To evaluate the current approach, ITC'99 [136] benchmarks are used. In ITC'99, the first ten benchmark circuits provided VHDL language at Register Transfer Level (RTL) are experimented with using their state diagram and RE models (Refer Corno et al. [136] for details of models and their analysis). These benchmarks enable us to apply the current approach to higher-level VLSI circuits.

The proposed approach is also evaluated on ITC'99 [136] benchmarks. The results for these benchmarks based on experimental evaluation are shown in Table 8.5.2. The columns for Circuits, VHDL, and Gate-Level explain details for the corresponding circuits. PI and PO stand for primary inputs and primary outputs. LOC refers to the line of codes. The gate-level column provides details such as the number of gates and the number of flip-flops for the gate-level implementation of the corresponding circuits.

Table 8.5.2: ITC'99 Benchmark Results

Circuits				VHDL	Gate-Level		PQ-Analysis Results							
Name	Function	PI	PO	LOC	# Gates	#FF	I-D	I-C	R-D	R-C	D-D	IR	ID	IC
b01	FSM one	2	2	110	46	5	1	1	1	1	1	1	1	1
b02	FSM two	1	1	70	28	4	1	1	1	1	1	1	1	1
b03	Resource arbiter	4	4	141	149	30	1	1	1	1	1	1	1	1
b04	Compute min and max	11	8	102	597	60	1	1	1	1	1	1	1	1
b05	Elaborate the contents	1	36	332	935	34	1	1	1	1	1	1	1	1
b06	Interrupt handler	2	6	128	60	9	1	1	1	1	1	0	1	1
b07	Count points	1	8	92	420	49	1	1	1	1	1	1	1	1
b08	Find inclusions	9	4	89	167	21	1	1	1	1	1	1	1	1
b09	Converter	1	1	103	159	28	1	1	1	1	1	1	1	1
b10	Voting system	11	6	167	189	17	1	1	1	1	1	1	1	1

Based on the results given in TABLE 8.5.2, the circuits except for b06 (Interrupt handler) are fault-tolerant for I, R, and D operators. The self-detecting and self-correcting ability of the b06 circuit is corrupted due to insertion and replace operators' dependence shown as 0 in the table. However, the b06 circuit can be extended by structural redundancy, which leads b06 to self-detecting and correcting. The b06 circuit starts with an initialization state and then goes to the instruction state 1, namely "s_{intr1}", from which it can go to the initialization state or the instruction state, namely "s_{intr}". The "s_{intr1}" state triggers an ambiguity due to a further loop driving the system to the initialization state. The system needs to know which "s_{intr1}" state is triggered by first initialization or further loop-based initialization. By adding an idle state between the initialization state and "s_{intr1}" state, the system is able to overcome this ambiguity and become self-correcting.

Outputs of circuits before and after extension are evaluated for equivalency of input sequences. Also, the behavior of the extended circuit is also observed for the test sequence tn. It is assessed that both circuits produce equivalent outputs. Consequently, for operational purposes, circuits before and after extension are considered equivalent.

8.6 THREATS TO VALIDITY

The potential internal, external, and construct threats of the approach are presented in this section.

Internal Validity

The approach offers a uniform method for modeling, detecting, and self-correcting faults in sequential systems by structural redundancy determined by analysis of the system under test. It also employs an algorithm to determine the redundancy to be introduced for tolerating the faults modeled. This algorithm takes a RE model converted from a finite state machine. Due to the fact that this approach uses two different models and their conversions, any internal flaw in these models or their conversions leads to wrong analysis and results. Belli and Gueldali also proposed a test generation approach based on model checking that can also be a solution to model correctness [137].

Time redundancy caused by additional structural and functional properties is an essential part of fault-tolerant systems and may also cause threats to time-critical systems. However, the proposed approach aims to keep the time redundancy minimal, as validated by the experimental evaluation compared with other approaches.

External Validity

The size of the model for the system under test can be increased based on the size of this system. This may lead us to a scalability problem for the proposed approach because the current approach only applies to small and medium-sized systems. Another solution can be to apply community detection algorithm [109] to automatically obtain smaller parts of the huge size model and apply the current approach to these smaller models to cope with huge models that are well-suited for the hierarchical control structure of sequential systems.

Construct Validity

This thesis uses the PQ-Analysis tool to analyze the model of the system under test automatically and extract features for fault tolerance by defining the events for possible extensions. However, the FSM model is manually obtained for the SE from its specifications and codes. This may lead us to construct validity and affect the results. For the sequential systems using their Verilog codes, the FSM models can be automatically obtained using a static analysis method that is already automatized by a tool called Verilog2FSM [103, 104].

CHAPTER 9

CONCLUSIONS

This chapter presents the conclusion for proposed test generation methods and the MBIT approach for sequential HDL and GUI programs.

9.1 TEST GENERATION

In the scope of this thesis, an approach for test sequence generation using RE is introduced to target design faults at the behavioral level. The proposed approach starts with extracting the FSM model from a given HDL program. Then, the obtained FSM is converted into RE by means of the well-known Brzozowski algorithm [96]. Afterwards, this RE model is represented by ST from which the test sequences are generated using a tree traversal algorithm considering pre-defined coverage criteria on RE.

A contextual RE-based test generation approach is also proposed for testing GUI programs. The proposed approach is used effectively in revealing functional faults (defined in Section) 1 and used in Section 7.2.1. Coverage-oriented random test generation enables control of the test process by setting the desired coverage ratio based on the contextual RE model. The context-driven test generation from the RE model offers a new perspective on the model-based test generation area.

Briefly, the contributions of the proposed approaches for RE and contextual RE can be listed as follows:

1. Test sequence generation based on RE in terms of operator and alphabet coverage criteria to target design faults at the behavioral level for HDL validation.

2. Test sequence generation based on contextual RE in terms of right and left context table coverage criteria to target functional faults for GUI testing.

3. Developing a toolchain to support both approaches.

4. Supporting both available HDL programs and specification of the system under development in the proposed approach within this thesis.

These approaches and their tools are used in MBIT methodology and its evaluation.

9.2 MODEL-BASED IDEAL TESTING

This section provides a conclusion for sequential HDL and GUI programs with respect to the MBIT approach.

A method called MBIT is proposed to show the presence and absence of HDL and GUI program faults when using its model. The introduced methodology is called model-based ideal test (MBIT) with respect to satisfying the reliability and validity requirements of the Fundamental Test Theory of Goodenough and Gerhart [2]. To achieve the ideal test, this thesis utilizes holistic [7] and mutation testing [9, 10]. Holistic testing proposes negative and positive testing to check the desired and undesired features of the system under test, respectively. Therefore, this idea is adapted to show the presence and absence of faults. Mutation testing provides the generation of an erroneous or faulty version of the system under test by using mutation operators. This thesis uses mutation testing to generate mutants of the system under test and its specification. Test sequences are generated from both the supposedly fault-free (original) model and the mutant (fault-injected) models and executed on both the mutant programs and the original programs, respectively. Test selection collects the results of each execution and constructs an ideal test suite that is used for checking the presence and absence of faults. Moreover, the theorems and their proof for MBIT methodology are also provided to show that it satisfies the reliability and validity requirements of the Fundamental Test Theory of Goodenough and Gerhart [2].

The proposed approach is implemented with FSMs for model mutation, and later, their REs are analyzed for test generation (using the PQ-TestGen tool). However, any model-based approach for system behavior modeling (such as FSM [73], State Chart [31], RE [103], ESG [8], and EFG [39]) can be adapted in the proposed methodology for model mutation and test generation. Additionally, a tester is enabled to resize the domain and corresponding range of a program by means of utilizing a model, which provides a "pay as you go" solution. The tester can incorporate as many features as he/she wishes into the model within the framework of his/her resources with respect to time and money.

Briefly, the following contributions are given with respect to the proposed methodology:

1. A novel validation methodology is proposed to address the design faults of the HDL program at the behavioral level and the functional faults of the GUI programs.

  - The holistic and mutation-testing approaches, well-understood and widely used in software testing, are utilized to achieve a model-based *ideal test* (MBIT) of HDL and GUI programs.
  - The proposed methodology, with respect to the provided formal proof, satisfies the requirements of the *ideal testing* that is used to test the presence and absence of design and functional faults in the HDL and GUI programs.

2. The experimentation of the proposed methodology is conducted on three case studies (a sequence detector, a traffic light controller, and a RISC-V processor) for HDL programs and on two case studies, namely "Special" and "Additional" modules of the ISELTA webpage for GUI programs.

3. A toolchain is developed to automate the testing process and shared in the public domain [1] for HDL programs and in the public domain [2] for GUI programs.

The advantages of the proposed approach are:

1. The model-based ideal testing guarantees coverage of the modeled faults through positive and negative testing in the scope of a system model.
   - The ideal testing paves the way for construction of reliable and valid test suites.
   - Thus, it is possible to show both the presence and the absence of faults, as far as they can be modeled.

2. It offers high-level test generation based on analysis of the RE model.

3. The proposed approach for test generation is effective in terms of fault coverage, mutation score and test generation/execution time when compared with the other methods based on the experimental evaluation.

The proposed test generation method, called PQTestGen, is compared with different algorithms and different coverage settings, such as the RETestGen, PQRTestGen, and Graphwalker for HDL and GUI programs. PQTestGen achieves higher fault coverage than the other methods. Besides, PQTestGen has a higher mutation score than the other PT methods, meaning that it can kill all mutants for the HDL and GUI case studies. However, PQTestGen results in a more extensive test suite than the other methods due to the Breadth-First Search (BFS) algorithm (refer to Section 6.1.3 with the algorithm in Listing 7). This large test suite size in PQTestgen also increases test execution

1 MBIT4HW, `https://github.com/kilincceker/MBIT4HW`
2 MBIT4SW, `https://github.com/kilincceker/MBIT4SW`

time. However, the test sets’ size is within acceptable limits, considering the time required for test generation and execution steps, which finish in milliseconds.

CHAPTER 10

FOLLOW-ON PROJECTS

This chapter presents the possible future work and follow-on projects on test generation and model-based ideal testing of HDL and GUI programs within the scope of this thesis. These possible directions start with proposed test-generation algorithms (for Chapters 4 and 5) and continue with model-based ideal testing (for Chapter 6).

As one of the future projects for test generation based on RE, it is planned to cover some of the faults in the gate level, such as stuck-at-0/1 faults. To do this, a fault simulator can be used at the gate level, and generated test sequences from this approach can be applied to determine the fault coverage at the gate level. Another future work is to decrease and adjust the alphabet and operator coverage to analyze its impact on fault coverage. In this way, the suite can be tightened; thus, the cost of test generation and execution may be reduced.

As future work and for evaluating and improving the proposed approach for contextual RE-based test generation, further experiments with other model-based random test generation approaches such as AutoTest and GraphWalker are planned. Moreover, conducting more experiments is intended using different fault models to evaluate the effectiveness of PQRTestGen and thus improve these techniques and tools.

For model-based ideal testing, despite testing at the behavioral level, which is at a higher level of abstraction (resulting in much fewer components), there is a limitation in the scalability of the model. It is possible to encounter the problem of a state explosion if the SUT becomes very large. To tackle this problem, future work is planned by using model refinement [115] and/or model decomposition [138]. It is also planned to utilize the scalable mutation-based test generation methodology [71] to cope with this limitation. Finally, improving the efficiency of the algorithms to implement different steps of the methodology is another future work.

It is focused on enriching the concept of ideal testing by considering modern model-based testing techniques, thus introducing a novel approach to model-based ideal testing. In doing so, it does not yet focus on reducing the efficiency of the algorithms to implement different steps of the methodology, which is planned as the next step. All procedures for the proposed methodology are already automated except for the model and mutant generation steps that the test expert carries out. These steps require knowledge about the system under test. However, it is planned to automate the model generation step using one of the appropriate GUI ripping methods to extract the model from the GUI program automatically. It is also planned to automate the mutant generation step using omission, insertion, and replacement mutation operators on the FSM models. Finally, obtaining an end-to-end solution for test automation of GUI programs for the MBIT approach is another follow-on project.

To cope with a threat for MBIT related to model-correctness, it is planned to use model checkers [139] or, more importantly, get end-user feedback as early as possible to assess the model long before starting with test generation and testing itself. Also, it is intended to employ model refinement techniques [140], or layer-centric modeling proposed by Belli and Guler [140] to cope with scalability.

Lastly, the experimental evaluation will include different HDL program examples (such as SystemVerilog) as another future work to expand the MBIT approach's availability and generality.

BIBLIOGRAPHY

[1] Edsger W Dijkstra. *Chapter I: Notes on structured programming*. Academic Press Ltd., 1972.

[2] John B Goodenough and Susan L Gerhart. "Toward a theory of test data selection." In: *IEEE Transactions on software Engineering* SE-1.2 (1975), pp. 156–173.

[3] Kshirasagar Naik and Priyadarshi Tripathy. *Software testing and quality assurance: theory and practice*. John Wiley & Sons, 2011.

[4] Tsun S. Chow. "Testing software design modeled by finite-state machines." In: *IEEE transactions on software engineering* SE-4.3 (1978), pp. 178–187.

[5] William E. Howden. "Reliability of the path analysis testing strategy." In: *IEEE Transactions on Software Engineering* SE-2.3 (1976), pp. 208–215.

[6] M Morris Mano. *Digital logic and computer design*. Pearson Education India, 2017.

[7] Fevzi Belli. "Finite state testing and analysis of graphical user interfaces." In: *Proceedings 12th international symposium on software reliability engineering*. IEEE. 2001, pp. 34–43.

[8] Fevzi Belli, Christof J Budnik, Axel Hollmann, Tugkan Tuglular, and W Eric Wong. "Model-based mutation testing—approach and case studies." In: *Science of Computer Programming* 120 (2016), pp. 25–48.

[9] Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. "Hints on test data selection: Help for the practicing programmer." In: *Computer* 11.4 (1978), pp. 34–41.

[10] Richard G. Hamlet. "Testing programs with the aid of a compiler." In: *IEEE transactions on software engineering* SE-3.4 (1977), pp. 279–290.

[11] Jeff Offutt. "A mutation carol: Past, present and future." In: *Information and Software Technology* 53.10 (2011), pp. 1098–1107.

[12] Fabrizio Ferrandi, Franco Fummi, and Donatella Sciuto. "Implicit test generation for behavioral VHDL models." In: *Proceedings International Test Conference 1998 (IEEE Cat. No. 98CH36270)*. IEEE. 1998, pp. 587–596.

[13] Fabrizio Ferrandi, G Ferrara, Donatella Sciuto, Alessandro Fin, and Franco Fummi. “Functional test generation for behaviorally sequential models.” In: *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*. IEEE. 2001, pp. 403–410.

[14] Valéria Lelli, Arnaud Blouin, and Benoit Baudry. “Classifying and qualifying GUI defects.” In: *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2015, pp. 1–10.

[15] Samuli Rahkonen. “Mutation-Based Qualification of Module Verification Environments.” MA thesis. Tampere University of Technology, 2016.

[16] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. “MuJava: an automated class mutation system.” In: *Software Testing, Verification and Reliability* 15.2 (2005), pp. 97–133.

[17] Sandra Camargo Pinto Ferraz Fabbri, José Carlos Maldonado, and ME Delamaro. “Proteum/FSM: a tool to support finite state machine validation based on mutation testing.” In: *Proceedings. SCCC’99 XIX International Conference of the Chilean Computer Science Society*. IEEE. 1999, pp. 96–104.

[18] Bernhard K Aichernig, Harald Brandl, Elisabeth Jöbstl, Willibald Krenn, Rupert Schlick, and Stefan Tiran. “Killing strategies for model-based mutation testing.” In: *Software Testing, Verification and Reliability* 25.8 (2015), pp. 716–748.

[19] Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. “Basic concepts and taxonomy of dependable and secure computing.” In: *IEEE transactions on dependable and secure computing* 1.1 (2004), pp. 11–33.

[20] Boris Beizer. *Software testing techniques*. Dreamtech Press, 2003.

[21] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

[22] Michael Bushnell and Vishwani Agrawal. *Essentials of electronic testing for digital, memory and mixed-signal VLSI circuits*. Vol. 17. Springer Science & Business Media, 2004.

[23] Prabhat Mishra, Ronny Morad, Avi Ziv, and Sandip Ray. “Post-silicon validation in the soc era: A tutorial introduction.” In: *IEEE Design & Test* 34.3 (2017), pp. 68–92.

[24] William K Lam. *Hardware Design Verification: Simulation and Formal Method-Based Approaches (Prentice Hall Modern Semiconductor Design Series)*. Prentice Hall PTR, 2005.

[25] Tomas Bengtsson and Shashi Kumar. *A survey of high level test generation: Methodologies and fault models*. Ingenjörshögskolan, Högskolan i Jönköping, 2004.

[26] Gert Jervan, Zebo Peng, Olga Goloubeva, Matteo Sonza Reorda, and Massimo Violante. “High-level and hierarchical test sequence generation.” In: *Seventh IEEE International High-Level Design Validation and Test Workshop, 2002.* IEEE. 2002, pp. 169–174.

[27] Marcello Lajolo, Maurizio Rebaudengo, Matteo Sonza Reorda, Massimo Violante, and Luciano Lavagno. “Behavioral-level test vector generation for system-on-chip designs.” In: *Proceedings IEEE International High-Level Design Validation and Test Workshop (Cat. No. PR00786)*. IEEE. 2000, pp. 21–26.

[28] Markus Stumptner and Franz Wotawa. “A model-based tool for finding faults in hardware designs.” In: *Artificial Intelligence in Design'96*. Springer, 1996, pp. 541–559.

[29] Roderick Bloem and Franz Wotawa. “Verification and fault localization for VHDL programs.” In: *Journal of the Telematics Engineering Society (TIV)* 2 (2002), pp. 30–33.

[30] Donghwan Shin, Eunkyoung Jee, and Doo-Hwan Bae. “Comprehensive analysis of FBD test coverage criteria using mutants.” In: *Software & Systems Modeling* 15.3 (2016), pp. 631–645.

[31] Tom Mens, Alexandre Decan, and Nikolaos I Spanoudakis. “A method for testing and validating executable statechart models.” In: *Software & Systems Modeling* 18.2 (2019), pp. 837–863.

[32] Mark Glasser. *Open verification methodology cookbook*. Springer Science & Business Media, 2009.

[33] Allon Adir, Shady Copty, Shimon Landa, Amir Nahir, Gil Shurek, Avi Ziv, Charles Meissner, and John Schumann. “A unified methodology for pre-silicon verification and post-silicon validation.” In: *2011 Design, Automation & Test in Europe*. IEEE. 2011, pp. 1–6.

[34] Raghudeep Kannavara. “Towards a unified framework for pre-silicon validation.” In: *IISA 2013*. IEEE. 2013, pp. 1–7.

[35] Nathaniel August. “A robust and efficient pre-silicon validation environment for mixed-signal circuits on intel's test chips.” In: *9th International Symposium on Quality Electronic Design (isqed 2008)*. IEEE. 2008, pp. 423–428.

[36] Amr Lotfy, Syed Feruz Syed Farooq, Qi S Wang, Soner Yaldiz, Praveen Mosalikanti, and Nasser Kurd. “A system-verilog behavioral model for PLLs for pre-silicon validation and top-down design methodology.” In: *2015 IEEE Custom Integrated Circuits Conference (CICC)*. IEEE. 2015, pp. 1–4.

[37] Richard K Shehady and Daniel P Siewiorek. "A method to automate user interface testing using variable finite state machines." In: *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing*. IEEE. 1997, pp. 80–88.

[38] Atif M Memon, Martha E Pollack, and Mary Lou Soffa. "Hierarchical GUI test case generation using automated planning." In: *IEEE transactions on software engineering* 27.2 (2001), pp. 144–155.

[39] Atif M Memon. "An event-flow model of GUI-based applications for testing." In: *Software testing, verification and reliability* 17.3 (2007), pp. 137–157.

[40] Lee White and Husain Almezen. "Generating test cases for GUI responsibilities using complete interaction sequences." In: *Proceedings 11th International Symposium on Software Reliability Engineering. ISSRE 2000*. IEEE. 2000, pp. 110–121.

[41] Qing Xie and Atif M Memon. "Using a pilot study to derive a GUI model for automated testing." In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 18.2 (2008), pp. 1–35.

[42] Fevzi Belli, Christof J Budnik, and Lee White. "Event-based modelling, analysis and testing of user interactions: approach and case study." In: *Software Testing, Verification and Reliability* 16.1 (2006), pp. 3–32.

[43] Si Huang, Myra B Cohen, and Atif M Memon. "Repairing GUI test suites using a genetic algorithm." In: *2010 Third International Conference on Software Testing, Verification and Validation*. IEEE. 2010, pp. 245–254.

[44] Fevzi Belli, Mutlu Beyazit, and Nevin Güler. "Event-oriented, model-based GUI testing and reliability assessment—Approach and case study." In: *Advances in Computers*. Vol. 85. Elsevier, 2012, pp. 277–326.

[45] Ishan Banerjee, Bao Nguyen, Vahid Garousi, and Atif Memon. "Graphical user interface (GUI) testing: Systematic mapping and repository." In: *Information and Software Technology* 55.10 (2013), pp. 1679–1694.

[46] Fevzi Belli, Mutlu Beyazıt, Christof J Budnik, and Tugkan Tuglular. "Advances in model-based testing of graphical user interfaces." In: *Advances in Computers*. Vol. 107. Elsevier, 2017, pp. 219–280.

[47] Emil Alégroth and Robert Feldt. "On the long-term use of visual gui testing in industrial practice: a case study." In: *Empirical Software Engineering* 22.6 (2017), pp. 2937–2971.

[48] Tom Yeh, Tsung-Hsiang Chang, and Robert C Miller. "Sikuli: using GUI screenshots for search and automation." In: *Proceedings of the 22nd annual ACM symposium on User interface software and technology*. 2009, pp. 183–192.

[49] N Olsson and K Karl. *Graphwalker: The open source model-based testing tool*. 2015.

[50] Juha Eskonen, Julen Kahles, and Joel Reijonen. "Automating GUI Testing with Image-Based Deep Reinforcement Learning." In: *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE. 2020, pp. 160–167.

[51] David Adamo, Md Khorrom Khan, Sreedevi Koppula, and Renée Bryce. "Reinforcement learning for android gui testing." In: *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. 2018, pp. 2–8.

[52] A. Silistre, O. Kilincceker, F. Belli, M. Challenger, and G. Kardas. "Community Detection in Model-based Testing to Address Scalability: Study Design." In: *2020 15th Conference on Computer Science and Information Systems (FedCSIS)*. 2020, pp. 657–660.

[53] Fevzi Belli, Andre Takeshi Endo, Michael Linschulte, and Adenilso Simao. "A holistic approach to model-based testing of Web service compositions." In: *Software: Practice and Experience* 44.2 (2014), pp. 201–234.

[54] Fevzi Belli and Michael Linschulte. "On negative tests of web applications." In: *Annals of Mathematics, Computing & Teleinformatics* 1.5 (2008), pp. 44–56.

[55] Fevzi Belli, Christof J Budnik, and Axel Hollmann. "Holistic testing of interactive systems using statecharts." In: *Sicherheit 2006, Sicherheit–Schutz und Zuverlässigkeit* (2006).

[56] Onur Kilincceker, Ercument Turk, Moharram Challenger, and Fevzi Belli. "Applying the Ideal Testing Framework to HDL Programs." In: *ARCS Workshop 2018; 31th International Conference on Architecture of Computing Systems*. VDE. 2018, pp. 1–6.

[57] Gizem Mercan, Evrim Akgündüz, Onur Kilincceker, Moharram Challenger, and Fevzi Belli. "Android uygulaması testi için ideal test ön çalışması." In: *CEUR Workshop Proceedings*. Vol. 2201. 12. National Software Engineering Symposium. 2018.

[58] Onur Kilincceker and Fevzi Belli. "Towards Uniform Modeling and Holistic Testing of Hardware and Software." In: *2019 1st International Informatics and Software Engineering Conference (UBMYK)*. IEEE. 2019, pp. 1–6.

[59] Gordon Fraser and Franz Wotawa. “Using model-checkers to generate and analyze property relevant test-cases.” In: *Software Quality Journal* 16.2 (2008), pp. 161–183.

[60] Yue Jia and Mark Harman. “An analysis and survey of the development of mutation testing.” In: *IEEE transactions on software engineering* 37.5 (2010), pp. 649–678.

[61] TB Nguyen and C Robach. “Mutation Testing Applied to Hardware: the Mutants Generation.” In: *Proceedings of the 11th IFIP International Conference on Very Large Scale Integration*. 2001, pp. 118–123.

[62] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. “Mutation testing advances: an analysis and survey.” In: *Advances in Computers*. Vol. 112. Elsevier, 2019, pp. 275–378.

[63] Kim N King and A Jefferson Offutt. “A fortran language system for mutation-based software testing.” In: *Software: Practice and Experience* 21.7 (1991), pp. 685–718.

[64] W Eric Wong and Aditya P Mathur. “Reducing the cost of mutation testing: An empirical study.” In: *Journal of Systems and Software* 31.3 (1995), pp. 185–196.

[65] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. “MuJava: a mutation system for Java.” In: *Proceedings of the 28th international conference on Software engineering*. 2006, pp. 827–830.

[66] Sandra Camargo Pinto Ferraz Fabbri, José Carlos Maldonado, Tatiana Sugeta, and Paulo Cesar Masiero. “Mutation testing applied to validate specifications based on statecharts.” In: *Proceedings 10th International Symposium on Software Reliability Engineering (Cat. No. PR00443)*. IEEE. 1999, pp. 210–219.

[67] Fevzi Belli and Mutlu Beyazit. “Event-based mutation testing vs. state-based mutation testing-an experimental comparison.” In: *2011 IEEE 35th Annual Computer Software and Applications Conference*. IEEE. 2011, pp. 650–655.

[68] Gordon Fraser, Franz Wotawa, and Paul E Ammann. “Testing with model checkers: a survey.” In: *Software Testing, Verification and Reliability* 19.3 (2009), pp. 215–261.

[69] Paul E Ammann, Paul E Black, and William Majurski. “Using model checking to generate tests from specifications.” In: *Proceedings Second International Conference on Formal Engineering Methods (Cat. No. 98EX241)*. IEEE. 1998, pp. 46–54.

[70] Andreas Fellner, Mitra Tabaei Befrouei, and Georg Weissenbacher. “Mutation testing with hyperproperties.” In: *Software and Systems Modeling* (2021), pp. 1–23.

[71] Andreas Fellner, Willibald Krenn, Rupert Schlick, Thorsten Tarrach, and Georg Weissenbacher. "Model-based, mutation-driven test-case generation via heuristic-guided branching search." In: *ACM Transactions on Embedded Computing Systems (TECS)* 18.1 (2019), pp. 1–28.

[72] ISWB Prasetya and Rick Klomp. "Test model coverage analysis under uncertainty: extended version." In: *Software and Systems Modeling* 20.2 (2021), pp. 383–403.

[73] Alexandre Petrenko. "Toward testing from finite state machines with symbolic inputs and outputs." In: *Software & Systems Modeling* 18.2 (2019), pp. 825–835.

[74] Chantal Robach and Mathieu Scholive. "Simulation-Based Fault Injection and Testing Unsing the Mutation Technique." In: *Fault injection techniques and tools for embedded systems reliability evaluation*. Springer, 2003, pp. 195–215.

[75] Mark Hampton and Stephane Petithomme. "Leveraging a commercial mutation analysis tool for research." In: *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE. 2007, pp. 203–209.

[76] Onur Kilincceker and Fevzi Belli. "Towards Uniform Modeling and Holistic Testing of Hardware and Software." In: *2019 1st International Informatics and Software Engineering Conference (UBMYK)*. IEEE. 2019, pp. 1–6.

[77] Luc Bougé. "A contribution to the theory of program testing." In: *Theoretical Computer Science* 37 (1985), pp. 151–181.

[78] Hans Langmaack. "Contribution to Goodenough's and Gerhart's theory of software testing and verification: Relation between strong compiler test and compiler implementation verification." In: *Foundations of Computer Science*. Springer. 1997, pp. 321–335.

[79] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. "Introduction to automata theory, languages, and computation." In: *Acm Sigact News* 32.1 (2001), pp. 60–65.

[80] Pan Liu, Jun Ai, and Zhenning Jimmy Xu. "A study for extended regular expression-based testing." In: *2017 IEEE/ACIS 16th International Conference on Computer and Information Science (ICIS)*. IEEE. 2017, pp. 821–826.

[81] Fevzi Belli. "Extending Regular Languages for Self-Detection and Self-Correction of Syntactical Faults (PhD Thesis in German; Technical Univ. Berlin)." In: *Bericht* 119 (1978).

[82] Aditya P Mathur. *Foundations of software testing, 2/e*. Pearson Education India, 2013.

[83] Leonardo Mariani, Mauro Pezze, and David Willmor. "Generation of integration tests for self-testing components." In: *International Conference on Formal Techniques for Networked and Distributed Systems*. Springer. 2004, pp. 337–350.

[84] B Eggers and Fevzi Belli. "Eine Theorie der Analyse und Konstruktion fehlertolerierender Systeme." In: *Fehlertolerierende Rechensysteme*. Springer, 1984, pp. 139–149.

[85] Fevzi Belli. "Regular Expressions for Fault Handling in Sequential Circuits." In: *ARCS 2015-The 28th International Conference on Architecture of Computing Systems. Proceedings*. VDE. 2015, pp. 1–5.

[86] Victor Mikhaylovich Glushkov. "The abstract theory of automata." In: *Russian Mathematical Surveys* 16.5 (1961), p. 1.

[87] Robert McNaughton and Hisao Yamada. "Regular expressions and state graphs for automata." In: *IRE transactions on Electronic Computers* EC-9.1 (1960), pp. 39–47.

[88] Robert R Schaller. "Moore's law: past, present and future." In: *IEEE spectrum* 34.6 (1997), pp. 52–59.

[89] Stephen Cole Kleene. *Representation of events in nerve nets and finite automata*. Tech. rep. RAND PROJECT AIR FORCE SANTA MONICA CA, 1951.

[90] Janusz A Brzozowski and Edward J McCluskey. "Signal flow graph techniques for sequential circuit state diagrams." In: *IEEE Transactions on Electronic Computers* EC-12.2 (1963), pp. 67–76.

[91] Dean N Arden. "Delayed-logic and finite-state machines." In: *2nd Annual Symposium on Switching Circuit Theory and Logical Design (SWCT 1961)*. IEEE. 1961, pp. 133–151.

[92] Susan H Rodger and Thomas W Finley. *JFLAP: an interactive formal languages and automata package*. Jones & Bartlett Learning, 2006.

[93] Tsong Yueh Chen, Hing Leung, and Ieng Kei Mak. "Adaptive random testing." In: *Annual Asian Computing Science Conference*. Springer. 2004, pp. 320–329.

[94] Fevzi Belli and Karl-Erwin Grosspietsch. "Specification of fault-tolerant system issues by predicate/transition nets and regular expressions-approach and case study." In: *IEEE Transactions on software engineering* 17.6 (1991), pp. 513–526.

[95] Alex Groce, Josie Holmes, Darko Marinov, August Shi, and Lingming Zhang. "An extensible, regular-expression-based tool for multi-language mutant generation." In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. 2018, pp. 25–28.

[96] Janusz A Brzozowski. "Derivatives of regular expressions." In: *Journal of the ACM (JACM)* 11.4 (1964), pp. 481–494.

[97] Atif Memon, Ishan Banerjee, and Adithya Nagarajan. "GUI ripping: Reverse engineering of graphical user interfaces for testing." In: *10th Working Conference on Reverse Engineering, 2003. WCRE 2003. Proceedings.* Citeseer. 2003, pp. 260–269.

[98] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M Memon. "Using GUI ripping for automated testing of Android applications." In: *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. IEEE. 2012, pp. 258–261.

[99] Yue Jia and Mark Harman. "Higher order mutation testing." In: *Information and Software Technology* 51.10 (2009), pp. 1379–1393.

[100] Onur Kılınççeker and Fevzi Belli. "Grafiksel kullanıcı arayüzleri için düzenli ifade bazlı test kapsama kriterleri." In: *the 11th Turkish National Software Engineering Symposium*. UYMS 2017. 2017, pp. 332–343.

[101] Michael Gautschi, Pasquale Davide Schiavone, Andreas Traber, Igor Loi, Antonio Pullini, Davide Rossi, Eric Flamand, Frank K Gürkaynak, and Luca Benini. "Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices." In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25.10 (2017), pp. 2700–2713.

[102] A Jefferson Offutt, Gregg Rothermel, and Christian Zapf. "An experimental evaluation of selective mutation." In: *Proceedings of 1993 15th international conference on software engineering*. IEEE. 1993, pp. 100–107.

[103] Onur Kilinccceker, Ercument Turk, Moharram Challenger, and Fevzi Belli. "Regular Expression Based Test Sequence Generation for HDL Program Validation." In: *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE. 2018, pp. 585–592.

[104] Onur Kilincceker, Alper Silistre, Moharram Challenger, and Fevzi Belli. "Random Test Generation from Regular Expressions for Graphical User Interface (GUI) Testing." In: *2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE. 2019, pp. 170–176.

[105] Hermann Gruber, Markus Holzer, and Michael Tautschnig. "Short regular expressions from finite automata: Empirical results." In: *Implementation and Application of Automata: 14th International Conference, CIAA 2009, Sydney, Australia, July 14-17, 2009. Proceedings 14*. Springer. 2009, pp. 188–197.

[106] Edmund M Clarke Jr, Orna Grumberg, Daniel Kroening, Doron Peled, and Helmut Veith. *Model checking*. MIT press, 2018.

[107] Peter Flake, Phil Moorby, Steve Golson, Arturo Salz, and Simon Davidmann. "Verilog HDL and its ancestors and descendants." In: *Proceedings of the ACM on Programming Languages* 4.HOPL (2020), pp. 1–90.

[108] Gordon Fraser and Andrea Arcuri. "Achieving scalable mutation-based generation of whole test suites." In: *Empirical Software Engineering* 20.3 (2015), pp. 783–812.

[109] Alper Silistre, Onur Kilincceker, Fevzi Belli, Moharram Challenger, and Geylani Kardas. "Community Detection in Model-based Testing to Address Scalability: Study Design." In: *2020 15th Conference on Computer Science and Information Systems (FedCSIS)*. IEEE. 2020, pp. 657–660.

[110] Muhammad Zohaib Iqbal, Andrea Arcuri, and Lionel Briand. "Environment modeling and simulation for automated testing of soft real-time embedded software." In: *Software & Systems Modeling* 14.1 (2015), pp. 483–524.

[111] Elmira Karimi, Mohamad Hashem Haghbayan, Adele Maleki, and Mahmoud Tabandeh. "Functional fault model definition for bus testing." In: *East-West Design Test Symposium (EWDTS 2013)*. 2013, pp. 1–4.

[112] Elmira Karimi, Mohamad Hashem Haghbayan, Adele Maleki, and Mahmoud Tabandeh. "Graph based fault model definition for bus testing." In: *2013 IFIP/IEEE 21st International Conference on Very Large Scale Integration (VLSI-SoC)*. 2013, pp. 54–55.

[113] Fevzi Belli, Nimal Nissanke, Christof J Budnik, and Aditya Mathur. "Test generation using event sequence graphs." In: *University of Paderborn, Institute for Electrical Engineering and Information Technology* (2005).

[114] Ana CR Paiva, Nikolai Tillmann, João CP Faria, and Raul FAM Vidal. "Modeling and testing hierarchical GUIs." In: *Proceedings of the 12th International Workshop on Abstract State Machines*. 2005.

[115] Fevzi Belli, Nevin Güler, and Michael Linschulte. "Layer-centric testing." In: *FERS-Mitteilungen: Vol. 30, No. 1* (2012).

[116] Luca Pascarella, Fabio Palomba, Massimiliano Di Penta, and Alberto Bacchelli. "How is video game development different from software development in open source?" In: *Proceedings of the 15th International Conference on Mining Software Repositories*. 2018, pp. 392–402.

[117] Edward F Moore. "The shortest path through a maze." In: *Proc. Int. Symp. Switching Theory, 1959*. 1959, pp. 285–292.

[118] Joseph Herkert, Jason Borenstein, and Keith Miller. "The Boeing 737 MAX: Lessons for engineering ethics." In: *Science and engineering ethics* 26.6 (2020), pp. 2957–2974.

[119] Robert N Charette. "This car runs on code." In: *IEEE spectrum* 46.3 (2009), p. 3.

[120] Fevzi Belli and Ferdinand Quella. *Holistic View of Software and Hardware Reuse*. Springer, 2021.

[121] Brian Randell and Jie Xu. "The evolution of the recovery block concept." In: *Software fault tolerance* 3 (1995), pp. 1–22.

[122] L Chen and A Avizienis. *N-version programming: a fault tolerant approach to reliability of software operation," Digest of the 1978 Fault Tolerant Computing Symposium*. 1978.

[123] Brian Randell. "System structure for software fault tolerance." In: *Ieee transactions on software engineering* 2 (1975), pp. 220–232.

[124] Mostafa I Abd-el barr. *Design and analysis of reliable and fault-tolerant computer systems*. World Scientific, 2006.

[125] Alessandro Strano, C Gómez, Daniele Ludovici, Michele Favalli, María Engracia Gómez, and Davide Bertozzi. "Exploiting network-on-chip structural redundancy for a cooperative and scalable built-in self-test architecture." In: *2011 Design, Automation & Test in Europe*. IEEE. 2011, pp. 1–6.

[126] Keith S Morgan, Daniel L McMurtrey, Brian H Pratt, and Michael J Wirthlin. "A comparison of TMR with alternative fault-tolerant design techniques for FPGAs." In: *IEEE transactions on nuclear science* 54.6 (2007), pp. 2065–2072.

[127] William H Pierce. *Failure-tolerant computer design*. Academic Press, 2014.

[128] Shailesh Niranjan and James F Frenzel. "A comparison of fault-tolerant state machine architectures for space-borne electronics." In: *IEEE Transactions on Reliability* 45.1 (1996), pp. 109–113.

[129] Whitney J Townsend, Jacob A Abraham, and Earl E Swartzlander. "Quadruple time redundancy adders [error correcting adder]." In: *Proceedings 18th IEEE Symposium on Defect and Fault Tolerance in VLSI Systems*. IEEE. 2003, pp. 250–256.

[130] Soyeon Choi, Jiwoon Park, and Hoyoung Yoo. "Area-Efficient Fault Tolerant Design for Finite State Machines." In: *2020 International Conference on Electronics, Information, and Communication (ICEIC)*. IEEE. 2020, pp. 1–2.

[131] Michael Augustin, Michael Gossel, and Rolf Kraemer. "Selective fault tolerance for finite state machines." In: *2011 IEEE 17th International On-Line Testing Symposium*. IEEE. 2011, pp. 43–48.

[132] Aiman H El-Maleh and Ayed S Al-Qahtani. “A finite state machine based fault tolerance technique for sequential circuits.” In: *Microelectronics Reliability* 54.3 (2014), pp. 654–661.

[133] Jiwoon Park and Hoyoung Yoo. “Area-Efficient Differential Fault Tolerance Encoding for Finite State Machines.” In: *Electronics* 9.7 (2020). ISSN: 2079-9292.

[134] Fulvio Corno, Gianluca Cumani, Matteo Sonza Reorda, and Giovanni Squillero. “An RT-level fault model with high gate level correlation.” In: *HLDVT*. IEEE Computer Society, 2000, pp. 3–8. ISBN: 0-7695-0786-7.

[135] Adeboye Stephen Oyeniran, Raimund Ubar, Maksim Jenihhin, and Jaan Raik. “High-Level Implementation-Independent Functional Software-Based Self-Test for RISC Processors.” In: *J. Electron. Test.* 36.1 (2020), pp. 87–103.

[136] Fulvio Corno, Matteo Sonza Reorda, and Giovanni Squillero. “RT-Level ITC’99 Benchmarks and First ATPG Results.” In: *IEEE Des. Test Comput.* 17.3 (2000), pp. 44–53.

[137] Fevzi Belli and Baris Güldali. “Software Testing via Model Checking.” In: *ISCIS*. Ed. by Cevdet Aykanat, Tugrul Dayar, and Ibrahim Korpeoglu. Vol. 3280. Lecture Notes in Computer Science. Springer, 2004, pp. 907–916. ISBN: 3-540-23526-4.

[138] Sergei Devadze, Elena Fomina, Margus Kruus, and Alexander Sudnitson. “Web-based system for sequential machines decomposition.” In: *The IEEE Region 8 EUROCON 2003. Computer as a Tool.* Vol. 1. IEEE. 2003, pp. 57–61.

[139] Eduard P Enoiu, Daniel Sundmark, Adnan Čaušević, Robert Feldt, and Paul Pettersson. “Mutation-based test generation for plc embedded software using model checking.” In: *IFIP International Conference on Testing Software and Systems*. Springer. 2016, pp. 155–171.

[140] Fevzi Belli, Nevin Güler, and Michael Linschulte. “Does "Depth" Really Matter? On the Role of Model Refinement for Testing and Reliability.” In: *COMPSAC*. IEEE Computer Society, 2011, pp. 630–639. ISBN: 978-0-7695-4439-7.

DECLARATION

Put your declaration here.

Paderborn, July 2024

Onur Kilincceker