Learning-based Testing for Automotive Embedded Systems

A requirements modeling and Fault injection study

HOJAT KHOSROWJERDI

Licentiate Thesis
Stockholm, Sweden
May 2019

KTH School of Electrical Engineering and Computer Science
TRITA-EECS-AVL-2019:19 SE-100 44 Stockholm
ISBN: 978-91-7873-121-3 SWEDEN

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av teknologie licentiatexamen i datalogi onsdagen den 15 maj 2019 klockan 10.00 i E2, E-huset, huvudbyggnaden, Kungl Tekniska högskolan, Lindstedtsvägen 3, Stockholm.

Tryck: Universitetsservice US AB

In the name of Allah,
who is the God who is the greatest, than the computer science, computations, logic and all the world.

"Praise is due to Allah whose worth cannot be described by speakers, whose bounties cannot be counted by calculators and whose claim (to obedience) cannot be satisfied by those who attempt to do so, whom the height of intellectual courage cannot appreciate, and the divings of understanding cannot reach; He for whose description no limit has been laid down, no eulogy exists, no time is ordained and no duration is fixed. He brought forth creation through His Omnipotence, dispersed winds through His Compassion, and made firm the shaking earth with rocks."

"He is a Being, but not through phenomenon of coming into being. He exists but not from non-existence. He is with everything but not in physical nearness. He is different from everything but not in physical separation. He acts but without connotation of movements and instruments. He sees even when there is none to be looked at from among His creation. He is only One, such that there is none with whom He may keep company or whom He may miss in his absence."

Ali-Ibn Abi Taleb
Nahjul Balagha

To Imam Mahdi, The Twelfth Imam

O secret of God! No doubt indeed!
God! O God! Send him please!
You are the light, reflector of unseen!
God! O God! Send him please!
This night of separation,
May it end soon dear!
May it end and you - soon appear!
God! O God! Send him please!

Abstract

This thesis concerns applications of *learning-based testing* (LBT) in the automotive domain. In this domain, LBT is an attractive testing solution, since it offers a highly automated technology to conduct safety critical requirements testing based on machine learning. Furthermore, as a black-box testing technique, LBT can manage the complexity of modern automotive software applications such as advanced driver assistance systems. Within the automotive domain, three relevant software testing questions for LBT are studied namely: effectiveness of requirements modeling, learning efficiency and error discovery capabilities.

Besides traditional requirements testing, this thesis also considers fault injection testing starting from the perspective of automotive safety standards, such as ISO26262. For fault injection testing, a new methodology is developed based on the integration of LBT technologies with virtualized hardware emulation to implement both test case generation and fault injection. This represents a novel application of machine learning to fault injection testing. Our approach is flexible, non-intrusive and highly automated. It can therefore provide a complement to traditional fault injection methodologies such as hardware-based fault injection.

Sammanfattning

Denna uppsats handlar om tillämpningar av inlärningsbaserad testning för fordon. Inlärningsbaserad testning är lämpligt för fordon eftersom sådan testning möjliggör i hög grad automatiserad testning av säkerhetskritiska egenskaper baserat på maskininlärning. Som en testningsteknik betraktad som en svart låda kan inlärningsbaserad testning hantera komplexiteten i moderna fordons mjukvaruapplikationer så som avancerade förarassistanssystem. Tre relevanta frågor för inlärningsbaserad mjukvarutestning inom fordonsindustrin betraktas: kravmodelleringseffektivitet, inlärningseffektivitet och felupptäckter.

Denna uppsats betraktar utöver traditionell testning också testning baserad på felinjicering utifrån fordonssäkerhetsstandarder, som exempelvis ISO26262. En ny metodik har utvecklats som baserats på inlärningsbaserat testning och emulering av hårdvara för att implementera både testfallsgenerering och felinjecering. Detta är en ny användning av maskininlärning för testning baserad på felinjicering. Ansatsen är flexibel, smidig och i hög utsträckning automatisk. Den kan därför vara ett komplement till traditionella felinjeceringsmetoder som exempelvis baserats på hårdvara.

Acknowledgments

"He who does not thank people, does not thank Allah"
Prophet Muhammad (PBUH)

I thank all those who provided me the possibility in carrying out the research and to complete this licentiate step of my PhD studies. This is just a list that I can remember now. Please forgive me if your name is not mentioned :).

I am deeply grateful to my supervisor Karl Meinke for his infinite wisdom and knowledge and all the support throughout the course of my studies and this thesis. His apt comments and constructive criticism always helped me to improve my work. His patience and encouragement whenever I was down during the time, helped me to re-start and recover with new energy. I would also like to thank my second supervisor Dilian Gurov for his suggestions and comments. Furthermore, I would like to thank my Scania colleague Andreas Rasmusson for providing the framework to perform fault injection studies.

Special thanks to my teachers who introduced me to the wold of formal methods, Mads Dam and Per Austrin (also Karl Meinke and Dilian Gurov). I would like to thank Henrik Artman for his advise when I was struggling with educational issues.

I want to thank my previous and current office mates Adam Schill Collberg, Daniel Rosqvist, Hamed Nemati (my friend), Emma Riese and Karl Norrman with whom I shared several light moments in the office.

Many thanks also to all people I met at TCS, especially Johan Håstad, Viggo Kann (thanks for helping for teaching hours), Martin Monperrus, Philipp Haller (thanks for helping me in the compiler course), Cyrille Artho (thanks for reading the thesis draft and providing good suggestions), Musard Balliu (thanks for the model checking book), Sonja Buchegger, Roberto Guanciale, Johan Karlander (thanks for the yearly supervisory group meetings), Danupon Nanongkai, Stefan Nilsson, Jakob Nordström, Elena Troubitsyna, Douglas Wikström, Christoph Baumann, Sagnik Mukhopadhyay, Michele Scquizzato, Dmitry Sokolov, Sten Andersson, Richard

Glassey, Linda Kann (you always smile and is kind), Vahid Mosavat (take care my friend), Daniel Bosk (Salam, Khubi?), Mohit Daga (time to Semla), Susanna F. de Rezende (don't laugh when you have Christmass lunch :)), Jan Elffers, Stephan Gocht, Jonas Haglund (thanks for Swedish abstract help), Christian Lidström, Md Sakib Nizam Khan (thanks for the usb cable), Andreas Lindner (I won't forget Marktoberdorf), Guillermo Rodriguez Cano, Kilian Risse, Thatchaphol Saranurak, Joseph Swernofsky, Jan van de Brand, Zhao Xin, He Ye, Long Zhang, Meysam Aghighi, Ilario Bonacina, Marc Vinyals, Mladen Miksa, Benjamin Greschbach (you introduced me to the rest of TCS), Oliver Schwarz (chetori javoon?), Pedro de Carvalho, Massimo Lauria, Freyr Svarsson (thanks for the kitchen games), Siavash Soleimanifard (it was a nice quadcopter).

And finally, I would like to thank my beloved partner for her support and incredible patience and my parents for always being there for me. I am deeply grateful to my little boy Nikan who was born just before my PhD started and brought joy to our life in the family.

List of Abbreviation

V2V	Vehicle-to-Vehicle
V2I	Vehicle-to-Infrastructure
OEM	Original equipment manufacturer
ASIL	Automotive software integrity level
HIL	Hardware-in-the-loop
HIFI	Hardware-implemented fault injection
SWIFI	Software-implemented fault injection
VFI	Virtualization-based fault injection
FI	Fault injection
FSM	Finite state machine
SFI	Simulation-based fault injection
DBT	Dynamic binary translation
LBT	Learning-based testing
SUT	Software under test
SUL	System under learning
EFI	Emulation-based fault injection
FPGA	Feld programmable gate array
FSM	Finite state machine
MIFI	Model-implemented fault injection
VMs	Virtual machines
DFA	Deterministic finite automata
NFA	Nondeterministic finite automata
LTL	Linear time temporal logic
CTL	Branching time temporal logic
BDD	Binary decision diagrams
SAT	Satisfiability solvers
SMV	Symbolic model verifier
PLTL	Propositional linear temporal logic
GPR	General purpose register
IR	Intermediate representation
TCG	Tiny code generator
PC	Program counter
RTSP	Real-time specification pattern

SPS	Specification Pattern System
TCTL	Timed computation tree logic
MTL	Metric temporal logic
SMT	Satisfiability modulo theories

Included papers in this thesis

Paper I:

Hojat Khosrowjerdi, Karl Meinke, and Andreas Rasmusson. *"Learning-Based Testing for Safety Critical Automotive Applications."* 5th International Symposium on Model-Based Safety and Assessment (IMBSA), 2017, Trento, Italy, pp. 197-211.

Paper II:

Hojat Khosrowjerdi, Karl Meinke, and Andreas Rasmusson. *"Virtualized-Fault Injection Testing: a Machine Learning Approach."* In Software Testing, Verification and Validation (ICST), 2018 IEEE International Conference on, pp. 297-308. IEEE, 2018.

Contents

Part I

Introduction and Literature Survey

Chapter 1

Introduction

Within the automotive electronics and control domain, the trend in recent years has been towards handing more safety-related functionality to embedded systems, especially electronic control units (ECUs) [92]. These are termed safety-critical embedded systems, because their malfunction can lead to hazardous situations where the safety of drivers or other road users is compromised. Already today, not only premium segment cars, but also series-produced vehicles carry dozens of operational ECUs realizing hundreds of proactive and driver assistive features (e.g., adaptive cruise control, steer-by-wire, brake-by-wire, collision avoidance). The software typically contains roughly 100 million software lines of code (SLOC) which is many times larger compared with modern fighter jets and some commercial airplanes and is indeed very complex [31].

In addition, with the advent of V2V (Vehicle-to-Vehicle) and V2I (Vehicle-to-Infrastructure) communications [14] which provide interaction between road users, automotive safety-related systems are destined to become collaborative; and therefore, even more complex. These wireless communication infrastructures allow motor vehicles and other road users to prevent accidents by transmitting data (e.g., position and speed) over a network and take preemptive actions such as braking. The safe cooperation of such systems relies not only on the individual traffic agents but also on their collaboration and the wireless communication network. Automatic collaborative safety scenarios are introduced to address the verification challenges of cooperating traffic agents. One initiative is the SafeCOP project [101] with the aim to provide an approach to the safety assurance of cyber-physical systems-of-systems (e.g., cooperative vehicle platoons) using wireless communication.

The development costs associated with software and its controlled electronics can amount up to 40% of the overall vehicle's price [18]. This significant cost is still preferable and superior to the expenses and losses imposed on OEMs (Original Equipment Manufacturer) and other affected parties due to issues in ECU software or hardware after production. Hundreds of thousands of automotive recalls are issued annually, that are related to unforeseen scenarios or testing imperfections.

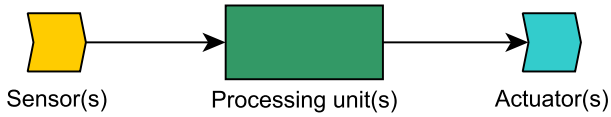

Figure 1.1: The common architecture of embedded systems

Given the Honda [8], Toyota [31] and Jaguar [30] cases, in some extreme environmental conditions hardware malfunctions could potentially give rise to faults in the ECU software. This evidence highlights the importance of safety analysis and comprehensive testing of embedded software to capture the anomalies and avoid such tremendous costs.

An embedded system with its architecture simplified in Figure 1.1, is composed of three central units, namely sensors, processors, and actuators (e.g., a fuel sensor, an ECU, and an electric pump). However, such a system in practice includes other hardware components (e.g., power supply) to provide reliable operating conditions for the main parts. Another common classification is to divide embedded systems into embedded hardware and embedded software sub-systems and study them separately. In the context of this thesis, the term *system* refers to an embedded system or its respective sub-systems unless otherwise stated.

Faults are defects within a system which may cause an error to occur [109]. Errors are incorrect states in the system which may lead to deviations from the required operation. Finally, failures are the events when incorrect states prevent the system from delivering its required service. Even the slightest failures in safety-critical systems can be catastrophic and should be avoided. One possible solution to accomplish safety is to develop such systems according to the recommendations of the current best practice standards. For example, ISO 26262 [9] (the tailored automotive version of IEC 61508 [7]) is a functional safety standard for the so-called E/E (electrical/electronic) systems over the life-cycle of a vehicle. ISO26262 specifies the methods that are utilized in the V-model of development to reduce risks and improve the safety of vehicular embedded systems. Testing is a significant validation activity that expands over all the verification phases, including the software, hardware, and system levels. In particular, the standard highly motivates the application of requirements-based testing as an approach to look for anomalies and check compliance with specified requirements [9].

Intuitively, requirements testing can be defined as an approach that revolves around requirements to derive preconditions, test data, and post-conditions of a test case. Hence, it can verify test results when the expected and actual behavior of a system under test match each other. One primary concern of requirements-based testing is how to generate test cases that cover the preconditions and postconditions of requirements. In addition, precise requirements modeling is also a major concern, since many safety issues are related to incompleteness or omissions of requirements.

Various formal notations have been proposed over the years to reduce the ambiguity of natural language texts and provide support for requirements analysis and testing; an extensive survey is [61]. Most of the related work in the literature has been devoted to the application of temporal logics [50] in formalizing behavioral safety requirements since these are expressive enough to describe most properties of continuous and reactive systems. Other requirements modeling approaches such as Live Sequence Charts (LSC) [59], Specification Pattern System (SPS) [42] and its real-time extension RTSP [75], have also been shown to be applicable, for example in the automotive domain.

Temporal logic reasons about events in discrete time where time is not bound to any specific scale. In other words, the aim is to describe the partial order of events beyond their exact time values. Temporal logic, especially linear time (LTL) and branching time (CTL), have been widely studied for the verification of industrial systems [103]. For example, in [74], it is shown that almost 90 percent of automotive embedded software requirements are formalisable using linear temporal logics (LTL).

Within safety engineering, there is no completely fault-free system because complete elimination of faults is impossible due to wear, aging and other effects [109]. Instead, safety-related systems are developed with additional measures to manage faults and future failures. These measures are sometimes referred to as "*safety mechanisms*" with the aim to detect, avoid, remove or tolerate faults and keep system failures at an acceptable level. Usually, a combination of these mechanisms is applied to achieve satisfactory results. It is usually the case that a system possesses a set of output states that are considered "*safe*". These can be for example disabling the airbag system in case of anomalies, if this is identified to be safer than an improper deployment of the airbag which may cause serious injuries. Therefore, safety mechanisms can put the system in a safe state until the fault is removed and the error is fixed.

However, a majority of highly critical systems do not possess a safe state but must tolerate the presence of faults and continue their operation properly. For example, a vehicle must maintain its ability to brake and steer even under the influence of faults. These "*fault-tolerant*" systems are designed to be *robust* (i.e., resilient against runtime errors) and be able to handle several possible temporary or permanent failures (e.g., hardware-related faults, I/O and memory errors) [109]. Validating the robustness of such systems requires considering two major factors, i.e., a set of requirements specifying the desired behavior of the system in all operating conditions and a method to analyze faulty scenarios and exercise the correctness of requirements under the influence of faults. One approach is to inject faults dynamically and/or simulate their effects to test error handling mechanisms as well as the robustness of the system under study.

Learning-based testing (LBT) [81] is a paradigm for fully automated black-box requirements testing that combines *machine-learning* with *model-checking*. What LBT offers, firstly, is to dynamically learn a state machine representation of the software under test (SUT). This reverse engineered behavioral model is then model-

checked against formalized safety-critical software requirements to evaluate their compliance or else construct counterexample test cases, violating the requirements. Using this technology, high test throughput, improved reliability, and high coverage can be achieved, while at the same time achieving a shorter testing time at a lower cost compared to manual testing.

In this thesis, these claims are evaluated on real-time safety-critical applications, such as can be found in the automotive sector, which typically have low test latency[1] [74]. Our work also gives insight into questions of technology transfer for the automotive sector. However, like any new testing technology, there is a need to consider how it can be integrated with existing software quality assurance (SQA) methods. Furthermore, for safety-critical systems, at least a semi-formal specification of safety requirements is highly recommended by ISO 26262. Given the rapid pace of technological and regulatory change within the automotive sector, the impact of research on behavioral modeling and testing is therefore potentially quite high.

1.1 Problem Formulation

According to ISO26262 [9], requirements testing and fault injection/insertion (FI) are highly motivated for ECU software, especially at automotive software integrity levels (ASILs) C and D. In practice, however, the automotive sector has faced several challenges to properly perform these tests. On the one hand, the rapid progress in the development of advanced automotive embedded systems brings more complexity to the verification of safety-critical functionality. This complexity represents a great bottleneck for manual validation and verification methods that still are common practice in the industry. What concerns the software quality assurance (SQA) community is that any test oracle needs to be fully automated and reliable. Hence, automating the process of requirements-based testing is necessary also for scalability to large test suits. These needs can be met by applying precise behavioral modeling formalisms to represent test requirements. Then, given an automated oracle and an executable version of the SUT, it is possible to conduct fully automated behavioral requirements testing. This not only saves a lot of time and effort to conduct the experiments but also increases the efficiency and the accuracy of results.

Therefore, the first question addressed in this thesis is: *how mature is the state-of-the-art in automated testing technology for the problem of behavioral requirements testing in the automotive sector*? In the context of this problem domain, we consider LBT as a potentially useful method to tackle these problems.

On the other hand, there is a large gap between the design stage of an embedded system and the stage in which FI is performed. FI tests are usually performed very late in final development stages with limited scope and coverage, e.g., in hardware-in-the-loop (HIL) testing and targeting only ECU pins. Injecting faults into embed-

[1] By test latency we mean the average time to execute a single test case on the SUT.

ded systems involves several challenges which could affect the corresponding testing results. One key aspect is the *representativeness* of the injected faults compared to reality. In other words, to what degree does an intentional corruption of hardware or software replicate a real fault scenario?

Another concern is whether FI is *intrusive*, which means, for instance, that fault insertion (injection) requires target software or hardware to be modified. Also in some cases, permanent damage to the hardware is inevitable, which is costly and therefore limits the scope and applicability to particular cases. In the automotive industry, realistic fault insertion is performed using HIL-rigs which control the ECU environment to avoid any damage to the hardware. Furthermore, an HIL rig, which includes a mock-up of electronic and electromechanical vehicle components, is an expensive and limited resource in the industry that becomes a bottleneck during testing.

Most of the techniques and tools proposed in the literature for FI usually focus on providing means and infrastructures to assist controlled occurrences of faults [69, 22, 95]. However, less effort has been devoted to the problem of automating the insertion process and measuring the coverage of generated tests [113]. In some cases, FI tests are manually created by a system test engineer to cover corresponding requirements [77, 119, 110]. Hence, this aspect is also open to further research.

Simulation-based fault injection (SFI) is an efficient approach in the literature to tackle representativeness, intrusiveness and the risk of damage problems [95, 48]. Furthermore, there is a growing trend towards more agile FI testing using modern hardware virtualization technology such as the Quick Emulator QEMU [20], as we will discuss in Section 7.2. This emerging paradigm offers initial support for the plausibility of a theory to introduce *virtualized HIL-rigs* (VHIL rig) as a new approach to perform FI testing during the development of ECU software. However, two further questions remain to be answered concerning: (1) *how to generate large test suites and test verdicts automatically and exploit the additional test throughput of a VHIL rig*; (2) *how to formalize safety requirements and systematically model-check their validity under the influence of a large combinatorial set of faults*?

Despite many positive results in recent works on FI (see Section 7), the problems of requirements modeling, test automation, and combinatorial FI are either not recognized or inadequately addressed. One practical solution to fill these gaps in the idea of a VHIL rig which integrates advanced technologies of automatic software requirements testing with hardware virtualization. Many virtualization-based (VFI) approaches (e.g., [52, 77, 55, 19]) rely on QEMU which seems to offer a promising technology for hardware virtualization. With the help of its strong dynamic binary translation (DBT) engine, QEMU can virtualize various guest hardware and run unmodified software on the host machine. Also, it provides an effective trade-off between simulation speed and realistic results.

1.2 Main Contributions of the Thesis

There is relatively little case study material published about black-box requirements modeling for the automotive sector. This thesis investigates the behavioral modeling languages and test functionality provided by LBT tools and proves these are a good match to the automotive domain. We strongly believe that requirements modeling is a subject which, just like coding, can be learned by looking over the shoulders of requirements engineers and reusing existing solutions on new products. To answer the central question about the usefulness of LBT in the automotive domain, a collection of automotive case studies with diverse ECU applications are tested using the LBT tool LBTest [89] and presented in [74]. In this preface, a summary of this approach is given which could also serve a pedagogical purpose.

For FI and test automation, LBT is potentially an interesting approach. In this thesis, we present a new methodology for VFI testing by combining hardware emulation with LBT. This represents a novel application of machine-learning to fault injection testing and hardware virtualization, requirements testing and model checking to manage FI campaigns. Moreover, using LBT in this context is attractive since it offers a promising automated technology to examine the behavior of the SUT even in *corner-cases* and manage the complexity of combinatorial faults.

In summary, the main contributions of the present work are:

1. an evaluation of LBT to automatically testing behavioral requirements on safety-critical applications;

2. a study of the ease of modeling informal behavioral requirements using temporal logic, and the success rate of LBT in finding known and unknown errors;

3. a new methodology for automated FI testing of safety-critical embedded systems based on formal requirements modeling, machine-learning and model-checking;

4. a toolchain based on integrating the QEMU hardware emulator with the GNU debugger GDB [1] and the LBTest requirements testing tool [89];

5. tool evaluation results on two industrial safety-critical ECU applications, where previously unknown FI errors were successfully found.

1.3 Thesis Outline

This thesis is organized into two parts. The first part presents the necessary framework to understand the results and provides a summary of the included papers. Part I includes Chapter 2 which presents the ideas of LBT; Chapter 3 discusses the principles of finite automata theory and the active automata learning used in LBT; Chapter 4 briefly addresses the model-checking problem for LBT; Chapter 5 investigates the viability of LBT for the automotive domain; Chapter 6 summarizes

the fault injection techniques for safety critical embedded systems and introduces a new approach to FI using LBT and a virtualized environment in QEMU; Chapter 7 gives a literature survey of the discussed material; Chapter 8 provides a summary of the two included papers; Chapter 9 presents concluding remarks. The second part includes two papers of the author.

Chapter 2

Learning-based Testing

In this chapter, we review some fundamental aspects and theoretical principles of learning-based testing (LBT) as these have been utilized in the research tool LBTest. After surveying the basic principles and a concrete implementation of the method, we emphasize some recent directions and applications of our approach.

2.1 Overview

Testing is a collection of processes or activities to *dynamically* verify the actual behavior of a program against its *expected behavior*, given a suitably *selected finite* set of test cases from the infinite execution domain [25]. From the above perspective, we could derive four main features of testing highlighted here:

- Dynamic. Unlike static verification methods such as symbolic execution or static program analysis, a significant feature of software testing is that it can give a record of the SUT's real-time dynamic behavior.
- Finite Selection. Exhaustive testing in practice means to consider all possible combinations of input values. This may be an infinite collection, and so complete testing is usually not feasible in any finite time. Therefore; tests have to be carefully and finitely selected to improve *coverage criteria* (see ahead).
- Expected behavior. Testing aims to reveal the undesired behavior of the SUT. The expected behavior has to be defined as a set of requirements, and an *oracle* is needed to judge whether a test is a failure or not at the point where test results are observed.

Some Testing Terms and Definitions

Consistent terminology is the key to avoid confusion of terms in the literature. In this work, we will follow the IEEE Standard Glossary of Software Engineering

Terminology [5].

- **Fault**. A fault is a defect in the system. It could be any mistake in the structural information such as source code, architecture or requirements. Sometimes also informally referred to as a *bug*.
- **Error**. A corrupted state or an incorrect result (e.g., a wrong variable value). Errors are prone to evolve and magnify during design, implementation and execution phases.
- **Failure**. When the component or the system fails to perform its expected function.
- **Test Case**. A set of carefully selected values from the input domain together with setup, tear-down and the expected test results to meet a particular test objective such as to cover a particular program path or to validate compliance with a specific requirement.
- **Test Suite**. A set of several test cases executed in a sequence with the intention to evaluate some predefined set of behaviours of a software program.
- **Verdict**. A verdict is a statement such as a *pass* or *fail* about the outcome of a test case execution relative to the expected results.
- **Oracle**. An external source to the system that makes verdict statements for the software under test.
- **Coverage**. Is the measured percentage of an entity or a property that has been tested by the test suite. Examples are code and requirements coverage.
- **State Space**. In a transition system or a program, state space refers to the set of states to be analyzed during a computation. This may be extremely large which can cause problems for exhaustive search algorithms such as model-checking that need to examine all possible paths (execution sequences) in search of a counterexample to the requirements.

Testing Opacity

Traditionally, testing techniques are classified based on the visibility of the source code when test cases are designed. *Glass(white)-box*, *black-box*, and *grey-box* are the three main classifications among the testing community [91]. Here, the "box" represents an SUT while the opacity, whether white, black or grey, determines the visibility level of the structural and internal details of a given system. From this perspective, the white-box approach (a.k.a., structural testing) evaluates the content (e.g., data structures, source code) of a software implementation, whereas black-box testing searches for problems in the behavior of the SUT to judge the quality and correctness of the box. This means that in black-box testing, the tester

requires no additional information about the structure of the SUT to generate test cases. Grey-box testing lies in between the two previously mentioned methods where to some extent there is some visibility of the SUT internal structure.

White-box testing has some issues. For example, it is hard to discover missing code (i.e., unimplemented requirements), since its methods are mainly concerned with existing program lines and paths. Combinatorial explosion is another common issue due to the exponential growth of reachable paths when loops are combined with conditional statements in the program.

On the other hand, black-box testing is not an ultimate stand-alone evaluation method but is complementary to white-box testing. When combined together, these approaches often intensively exercise use case or corner case scenarios, which requires a significant volume of test cases. Hence, they usually provide valuable quality assurance measures.

Since glass-box tests are derived from (complete) structural information about an SUT, test coverage can easily be inferred based on the identified portion of the software that has been evaluated. However, this information is not available in a black-box setting and the adequacy of a test suite, due to the unknown state space, cannot be precisely determined. In most cases, only an estimation of black-box test coverage can be obtained. This can be, for example, the degree to which a generated test suite covers and tests a design specification or, an equivalence checking measure of convergence between an approximated model and the SUT. Finally, a given test suite may demonstrate different black-box and white-box coverage figures because the amount of implemented source code for each functionality is not necessarily directly proportional to the demonstrated behaviors.

2.2 LBTest: from a black-box method to a test tool

Learning-based testing (LBT) [86] is a fully automated search heuristic for software errors guided by analogies between the processes of learning and testing. It is a novel black-box requirements testing approach that combines active machine learning with model-based testing [27]. For this reason, LBT can be termed *model-based testing without a model*.

In fact, the basic idea of LBT is to infer or reverse engineer a state machine representation of the SUT from test cases using deterministic model inference algorithms [40]. This process of inductive inference constructs a sequence of increasingly accurate models $M_1, M_2, ...$ that potentially contain unobserved errors of the SUT. Therefore, we need to statically analyze the models and generate extra test cases that discover potential SUT failure cases.

Model analysis can only identify potential system anomalies (e.g., counterexamples). Hence, to distinguish real test failures, the corresponding test cases have to be executed on the SUT. The result at this stage may either be: (i) a true negative (an SUT error), or (ii) a false negative (an artifact of an incomplete model). Hence, an equality test is required to compare the SUT response with the predicted faulty

behavior from the model and generate a *pass/fail* test verdict. Finally, each true negative can then be returned to the test engineer for post-mortem analysis.

Various approaches exist to analyze models of which model-checking (see, e.g., [36]) is a promising technique to simplify the process of test verdict generation. Model-checking usually requires an external reference model to evaluate the correctness of the SUT learned model against it. This is done by searching for any mismatch between the two models and reporting anomalies as counterexamples. A reference model can be, e.g., a test requirement formalized into a machine-readable format. For example, temporal logic (e.g., propositional linear temporal logic PLTL [43]) which is a widely used formal requirements language in many model-checkers. The advantage of PLTL is that test cases can be conveniently derived from generated counterexamples.

In LBT, observations are made at runtime either by a simulation environment (e.g., software-in-the-loop SIL, hardware-in-the-loop HIL) or field testing. For scalability to large problems, all testing processes are fully automated. In particular, test case and verdict generation activities are automated by interleaving active machine learning, model-checking, and equality testing queries [94]. Also, by means of equivalence checking, test progress or coverage are measured automatically. The latter can be quantified as an estimation of the convergence between the partial learned model and a behaviorally equivalent (complete) model of the SUT. This will be discussed in Sections 3.4 and 5.2 in more detail.

The soundness and completeness of LBT rely on the soundness and completeness of the utilized learning and model-checking methods. For example, the capability of an automata learning algorithm to converge is not always theoretically guaranteed, especially when the algorithm is based on some optimisation methods (see, e.g., [21]), but convergence is guaranteed for most automata learning algorithms, e.g., L* [15]. On the other hand, converging to a final complete model for a large complex system does not seem to be practical within a reasonable time frame (e.g., due to the need for infeasibly large datasets, etc.). To tackle these problems, LBT makes use of incremental learning algorithms where partial approximate models of the SUT evolve in small steps.

An efficient tool architecture to realize the ideas of LBT is shown in Figure 2.1. This architecture is a basis for the implementation of the LBTest tool which is used in several case studies [46, 88, 74, 84] to evaluate different features of an LBT approach. The inner feedback loop in Figure 2.1 between the SUT and learning process is to refine each approximated model M_i iteratively until it converges to a well-defined model. Another (outer) loop is formed around the model-checking, learning, equivalence checking and the SUT processes. This loop is used to analyse a partially converged model against a test requirement and execute a potentially generated counterexample on the SUT to construct pass or fail verdicts. Finally, the stochastic equivalence checker characterizes the coverage features of the final model M_{final} and reports the results for post-mortem analysis.

LBTest provides an automated test technology that is suitable for reactive systems, such as embedded software. It eliminates the need to maintain and up-

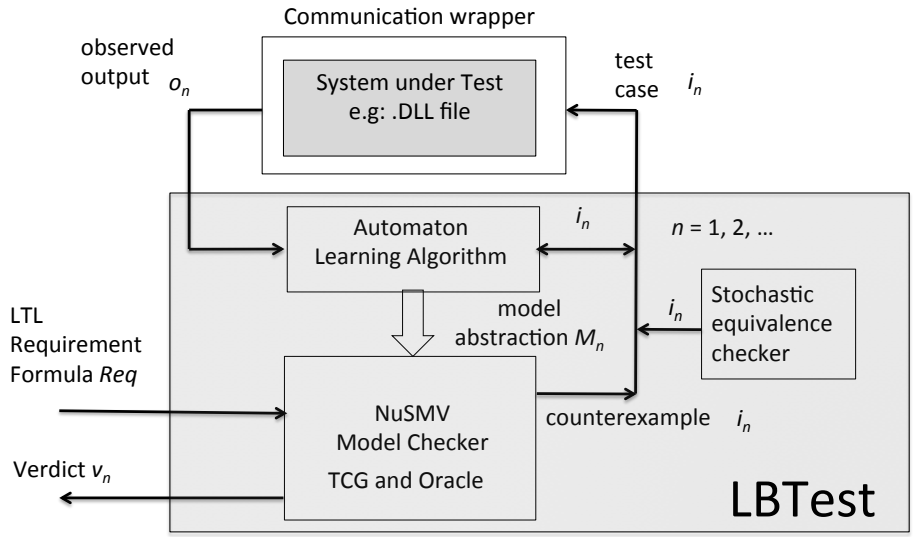

Figure 2.1: LBTest architecture

date models of software during development. LBTest generates abstract test cases based on user-specified symbolic data types defined in a particular configuration file. These abstract tests need to be mapped to concrete test cases with respect to the actual SUT API. The communication wrapper is the software component to handle this mapping task (see Figure 2.1). It acts as an adaptor to build a test harness around the SUT, sends test stimuli and returns SUT observations to LBTest.

The current version of the tool relies on the symbolic model verifier language (SMV) to represent the inferred models and a loosely integrated SMV checker NuSMV [32]. Two criteria let a user stop a test process, either using a bound on the test time or the number of partially converged learned models. These constrain the testing time while obtaining a model with desired size and convergence measures. However, LBTest may decide to terminate testing earlier when no further model improvements are made. In short, this means that LBTest assumes it has reached a complete model where all states are known.

Chapter 3

Algebraic Automata Theory

State machines (automata) are mathematical models that can be used to describe and analyze the dynamic behavior of a diverse group of reactive systems (e.g., embedded systems, control algorithms, digital circuits, telecom protocols, etc.) in an accurate and unambiguous way. They also have a connection to formal language theory by acting as language recognizers. What follows is an introduction to specific types of automata that attempts to discuss the essential concepts involved in LBT.

3.1 Overview

A state machine is a transition system (a graph) composed of sets of states ($Q = \{q_0, .., q_n\}$) and transitions ($q_i \rightarrow q_j$) that can occur between states. The states encode the status of the machine in each situation and transitions specify to which state it should switch when reacting to an input value. If the number of states and transitions is bounded, it is called a *finite* state machine (FSM). The opposite is an *infinite* state machine with infinitely many states and transitions. These state systems are usually used to deal with infinite-valued domains, e.g., integers, real numbers, etc. For example, an integer counter automaton is inherently infinite-state, since the values to be stored in the states are infinitely many integer numbers.

Visual representation of infinite state machines (e.g., in the form of directed graphs) in most cases is not practical due to memory and space constraints. However, such systems can be formalized mathematically using finite symbolic representations. Turing machines, Buchi automata, and Pushdown automata are examples of computational models for this task since they allow the specification of infinite structures.

An FSM, on the other hand, has finite memory fingerprint proportional to its state space size. Even though an FSM has less computational power compared to an infinite-state automaton, still it can represent many interesting problems in hardware and software. For example, an FSM with binary output is a language

recognizer for a *regular language*. When the output is multi-valued, the FSM is a *transducer*, e.g, a Mealy or Moore machine.

A couple of terms need to be defined before we go further into the study of relevant FSMs. (i) An input alphabet: $\Sigma = \{\sigma_1, ..., \sigma_k\}$ is a set of basic symbols for a language L. (ii) An output alphabet: $\Omega = \{\omega_1, ..., \omega_k\}$ is a set of output values. (iii) We let Σ^* denote the set of all finite strings over an input alphabet including the empty string ϵ. (iii) String length: Let $\alpha = \sigma_{i_1}...\sigma_{i_j} \in \Sigma^*$ be a finite string. $|\alpha|$ indicates its length j and $|\epsilon| = 0$. (iv) Concatenation: For any two strings $\alpha_1, \alpha_2 \in \Sigma^*$ the concatenation is indicated by $\alpha_1.\alpha_2$.

3.2 Types of Finite Automata

In a finite state machine, if there is exactly one start state and the transition relation for any state and input value is unique valued, it is said to be *deterministic*. The opposite is a *non-deterministic* state machine which is a more complex model of computation and, therefore, harder to learn[21]. Here, we begin with the simplest types of FSM, the deterministic finite automaton.

Deterministic Finite Automata

A *deterministic finite automaton* (DFA) is an FSM that acts as a language recognizer. The response to an input string is always unique. A DFA is defined as a five-tuple:

$$A = (Q, \Sigma, \delta : Q \times \Sigma \to Q, q_0, F),$$

where Q is a finite set of states, Σ is a finite alphabet of input symbols, $\delta : Q \times \Sigma \to Q$ is the state transition function, $q_0 \in Q$ denotes the start state, and $F \subseteq Q$ denotes the set of accepting or final states.

A DFA can accept or reject a sequence of input symbols. Hence, it has a binary output. An input string $x = a_1a_2...a_n$ is accepted if $\delta^*(q_0, x) \in F$ where δ^* is the iterated transition function defined as:

$$\delta^*(p, a.x) = \delta^*(\delta(p, a), x), \forall x \in \Sigma^* \text{ and } a \in \Sigma$$

The iterated transition function $\delta^*(p, x)$ receives an input string x and computes a sequence of transitions starting from state p to reach the last state which machine makes at the end of the computation process.

The language of a DFA, $L(A)$, denotes the set of all strings that the DFA accepts. Suppose an input string $x = a_1a_2...a_n$ is fed to a DFA. The transition function computes the next state for each input symbol, e.g., $\delta(q_0, a_1) = q_1$ and $\delta(q, \epsilon) = q$. After computing the sequence of state transitions $q_0, q_1, q_2, ..., q_n$ such that for each i, $\delta(q_i, a_{i+1}) = q_{i+1}$, x is in the language $x \in L(A)$ if $q_n \in F$, otherwise it is rejected by the DFA.

Even though the language recognized by a DFA (a *regular language*, see Appendix A) is constrained, it is expressive enough to model many of the current embedded systems behaviors. In general, DFA can even be used to deal with *non-deterministic* behavior of target systems and provide accurate approximated models for functional analysis. The Boolean output of a DFA is sometimes too restricted for more complex embedded systems with integer and float data types. Therefore, several extensions of DFA have been introduced with an additional parameter to indicate the output alphabet. Typical examples of such extensions are *Mealy* and *Moore machines*.

Mealy Machines

A deterministic Mealy machine is a six-tuple:

$$Mly = (Q, q_0, \Sigma, \Omega, \delta : Q \times \Sigma \to Q, \lambda : Q \times \Sigma \to \Omega)$$

where Q represents a finite set of states, $q_0 \in Q$ denotes the start state, Σ is a finite input alphabet, Ω is a finite output alphabet, $\delta : Q \times \Sigma \to Q$ is the state transition function, and $\lambda : Q \times \Sigma \to \Omega$ determines the output function computed from a state and an input symbol.

In Mealy machines, the output is associated with input values and is computed during each transition. This is the major difference between Mealy and Moore machines; otherwise, both can model equivalent classes of systems. In general, Mealy machines are more compact but somewhat harder as a learning problem for LBT.

Moore Machines

A deterministic Moore machine is defined as a six-tuple:

$$Mor = (Q, q_0, \Sigma, \Omega, \delta : Q \times \Sigma \to Q, \lambda : Q \to \Omega)$$

Compared with a Mealy machine where $\lambda : Q \times \Sigma \to \Omega$, the output in a Moore machine ($\lambda : Q \to \Omega$) is only associated with arriving at each target state q_i, i.e., if a transition occurs, the new output is observable in the next state.

In fact DFAs are a special case of Moore machines where the output alphabet is the binary set of $\Omega = \{accept, reject\}$. The application of DFAs in different fields of software analysis is a mature field and has been studied substantially in the literature [68]. However, the generalisation from a binary output to a multi-valued set for this purpose is not always trivial.

3.3 Algebraic Concepts for Machine Learning

A deterministic Moore machine is a universal algebraic structure.[1] Thus algebraic concepts such as isomorphism, congruence and quotient algebra are applicable. These concepts are important to have a deeper understanding of LBT principles, especially, machine learning.

Path

In automata theory, a *path* $\pi \in Q^*, \pi :=< q_0, q_1, ... >$ in an automaton A corresponds to a finite or infinite sequence of states starting from the initial state q_0 where there exists an infinite word $\omega = \sigma_0, \sigma_1, ... \in \Sigma^*$ such that $\forall i \geq 0 : q_{i+1} = \delta(q_i, \sigma_i)$.

Equivalent States

Equivalent states in a state machine are those that generate the same output sequence for all possible input strings regardless of the initial state. Otherwise they are *distinguishable* states with a string $x \in \Sigma^*$. Equivalence can be formally defined as:

$$\forall p, q \in Q, \quad p \approx q \text{ iff } \forall x \in \Sigma^*.\ \lambda(\delta^*(p, x)) = \lambda(\delta^*(q, x))$$

Note that the equivalence relation $\approx$ is *reflexive*, *symmetric* and *transitive*, i.e., $p \approx p$, $p \approx q \Rightarrow q \approx p$, and $p \approx q \wedge q \approx r \Rightarrow p \approx r$.

Equivalence Classes

An equivalence relation $\approx$ divides the underlying set of states of an automata A into disjoint *equivalence classes* (partitions). Each individual state belongs to exactly one partition. The equivalence class of $p \in Q$ is denoted by $[p] = \{q | p \approx q\}$. Further, if two states p, q are equivalent, their corresponding equivalence classes are the same: $(p \approx q)$ iff $([p] = [q])$.

State Minimization

A significant consequence of partitioning states into equivalence classes is that Moore machines can be minimized since equivalent states can be identified. Minimization involves the process of reducing the size of a Moore machine such that it still generates the same language. There exist algorithms to reduce the size of Moore machines (e.g., DFA) and construct a minimal equivalent one mechanically (e.g., Hopcroft's algorithm [68]).

[1] A universal algebraic structure is a first-order collection of data sets $A = \{A_1, ..., A_n\}$ and n-ary operations $f = \{f_1, ..., f_k\}$ that take n elements of A and return a single element of A, i.e., $f_k : A_{ik(1)} \times ... \times A_{ik(n)} \rightarrow A_{ik(n+1)}$ [82].

Congruence relation

A *congruence* is a special type of equivalence relation $\equiv$ on a Moore machine that is both closed and unambiguous *(well-defined)* under any algebraic functions. This definition can be summarized as follows:

For an n-ary function f, if $u_i \equiv v_i, 1 \leq i \leq n$, then

- $f(u_1, ..., u_n) \equiv f(v_1, ..., v_n)$

Under these substitutivity conditions, a congruence relation can be applied to a Moore machine. This leads to a deterministic (uniquely valued) automaton known as a quotient automaton. Here, the aforementioned conditions are modified to cover the transition relation $\delta_{A/\equiv}(q, \sigma)$ and output function $\lambda_{A/\equiv}(q, \sigma)$ of the state machine.

Again, for a given Moore automaton $A = (Q, \Sigma, \Omega, q_0, \delta, \lambda)$, $\equiv \subseteq Q \times Q$ on the state set Q is a congruence *iff*,

(i) $q \equiv q' \rightarrow \delta(q, a) \equiv \delta(q', a)$

(ii) $q \equiv q' \rightarrow \lambda(q, a) = \lambda(q', a)$

for any states $q, q' \in Q$ and input symbol $a \in \Sigma$. The first condition is termed a *state congruence* $\equiv_Q$ and the latter termed an *output congruence* $\equiv_\Omega$.

Quotient Automata

In automata theory, constructing a *quotient* automaton refers to partitioning states into equivalence classes and treating each partition as an individual state to build a new state machine. A mathematical definition is given here. If $A = (Q, \Sigma, \Omega, q_0, \delta, \lambda)$ is a Moore automaton, then $A/\approx = (Q^{'}, \Sigma, \Omega, q_0', \delta', \lambda')$ is a quotient Moore automaton where:

- $Q' = \{[q], q \in Q\}$, i.e., the set of equivalence classes of all states in Q,
- Σ is not changed,
- Ω is not changed,
- $q_0' = [q_0]$, i.e., the equivalence class of the initial state,
- $\delta'([q], a) = [\delta(q, a)]$, i.e., the equivalence class of the reached state after applying a as an input to $\delta(q, a)$ from a q state in the original state machine,
- $\lambda'([q]) = \lambda(q)$, i.e., the output of an equivalence classes is the same for all its states.

Conditions (i) and (ii) above ensure that δ' and λ' are mathematically well defined.

3.4 Automata Learning

Inductive inference [21] is the process of extracting general principles, laws or rules from a finite set of observations. In the context of *machine learning*, it refers to constructing a hypothesis model that typically contains a mixture of facts (from observations) and *conjectures* (i.e., extrapolations to unseen observations). In fact, it is the principles of inductive inference that allow us to infer an infinite set of model behaviors supported by just finite evidence. For example, a loop hypothesis derived from a finite repetition of specific observations.

In learning-based testing, models are inferred from a finite *dataset*, i.e, a set of queries (test cases) and observations (SUT responses). This inductive inference process can lead to discovering potential unforeseen errors in the SUT and also improving coverage criteria for testing. A learning algorithm can either predict the unknown behavior of a system $s \in S$ (S a set of systems) for future input sequences or summarize the available observations and make an approximation $h \in H$ of the actual system behavior.

Approximated models are not necessarily *exact*, meaning that they may diverge from the actual target behavior for the input sequences that are not covered in the observation data $d_i \in D$ ($D \subseteq \Sigma^*$ a subset of the input strings). However, for other subsets of input data, h and s usually closely or exactly agree. The approximation is *exact* if for all $d_i \in D : s(d_i) = h(d_i)$ where $s(d_i), h(d_i) \in \Omega^*$ (output values of s and h for an input d_i). Two types of questions (queries) are typically asked during a learning process, either what is the response of $s(d_i) =?$ (a.k.a., *membership query*) or whether $s(d_i) = h(d_i)$ for all $d_i \in \Sigma^*$ (a.k.a., *equivalence query*). These queries are usually sent to an *adequate teacher* (a.k.a., *oracle*) which returns accurate answers in response. Sometimes the answer to an equivalence query is negative. In these cases, an oracle may provide a *counterexample* from the set of $s(d_i) \setminus h(d_i)$ or $h(d_i) \setminus s(d_i)$, violating the query condition.

Model inference can be *active* or *passive*. It is active if the learning algorithm is able to interactively query the target system and obtain the desired information for new data points. In contrast, passive learning corresponds to an algorithm that has no interactions with the SUT. Instead, it tries to infer a model from a given and fixed set of (*query*, *response*) pairs provided to it.

Active Automata Learning

Incompleteness of a dataset is the main problem which an active automaton learning algorithm is trying to solve. A dataset is incomplete (for a given learning algorithm) if a behaviorally equivalent model of a system under learning (SUL) cannot be inferred unambiguously. Any identified incompleteness in the dataset results in a new active query that can generate a new observation. Hence a more complete dataset will be obtained. Iterating over this process, should (hopefully) eventually resolve the entire dataset incompleteness where complete learning is obtained.

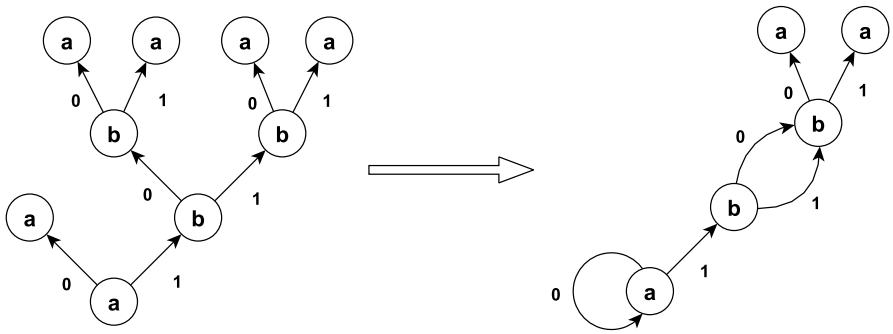

Figure 3.1: A sample prefix tree merging

Among computational learning algorithms, those that are suitable for inferring finite state machines are also known as *regular inference* algorithms. The principle of these techniques is rather simple. First, we can organize the input words into a *prefix tree* of branching degree size n, i.e., the cardinality of the input alphabet. This tree structure (as shown in Figure 3.1) consists of all input/output pairs of SUL samples stored in the tree. Then, a state machine model can be constructed by merging functionally equivalent nodes n_i and n_j with compatible sub-trees in the tree. Here the two merged nodes n_i and n_j can either form a *loop* in the resulting automaton or simply create a *joined/merged* path from the tree root depending on whether they lie along the same path from the root or not.

In Section 3.3, the conditions to define a congruence relation were studied. By inverting the rule (i) to its contrapositive form:

$$\delta(\overline{\alpha_1}, \sigma) \not\equiv \delta(\overline{\alpha_2}, \sigma) \rightarrow \overline{\alpha_1} \not\equiv \overline{\alpha_2}$$

for input strings $\overline{\alpha_1}, \overline{\alpha_2} \in \Sigma^*$ and then applying it to the prefix tree, a deterministic model can be obtained. Accordingly, any inconsistency between two nodes must be evaluated on all prefixes to build a congruence.

An essential principle of the prefix tree method is to assume any two nodes are behaviorally consistent unless any conflicting evidence is observed. This principle is known as the *closed-world* assumption, and the resulting learning process will be *non-monotonic*. Thus, early assumptions about mergeable nodes can be later falsified in the case of inconsistent observations.

Intuitively speaking, monotonicity means a new inferred hypothesis is never allowed to modify what a previously learned hypothesis already reflects. Non-monotonicity of regular inference methods makes it challenging to formulate whether a corresponding learning algorithm converges into a regular language. However, these inference algorithms are still provably convergent in most cases, e.g., L* [21]. Non-monotonicity may also result in both *under-* and *over-approximations* in the

inferred hypotheses (models). That is to say, either a path exists in the SUT which is not available in the model or vice-versa.

Angluin's L* and L* Mealy

Many algorithms have been proposed in the literature to learn automata from a sequence of input strings [53, 15, 73, 93, 83]. Most of these, e.g., Angluin's L* [15], are convergent and can be used to construct DFA recognizer for any target language.

L* is one of the classical algorithms also used for pedagogical purposes in active learning. It is easy to understand and implement, while covering most important aspects of more efficient algorithms. L* is proven to converge in polynomial time if there is a guarantee to detect behavior inconsistencies between the learned automaton and the SUL [21]. It is also provable that L* infers the minimum-sized automaton of the SUL, which is a nice feature of the algorithm.

Since LBT works with Moore machines, it is more appropriate to present a simple generalization of L* that can handle an arbitrary output alphabet Ω. This algorithm is termed L* Mealy and discussed in [21] in greater depth. Here, we can only sketch the basic principles due to lack of space and leave further details to the interested reader.

L* Mealy creates a two-dimensional observation table $OT = (P, S, T)$ of prefix, suffix and table entries to organize a finite set of observations into equivalence classes of states. This table dynamically expands over time, as incompleteness is detected in the table and new queries are generated to incrementally learn the SUL. Prefixes are divided into color-coded red and blue prefix sets of P_{red} and P_{blue} defined below. At any stage of the learning process and for a given input/output alphabet Σ/Ω,

- $P_{red} \subseteq \Sigma^*$ is the *red-prefix* set of input strings that is prefix-closed. i.e., if $p, u \in \Sigma^*$ then any prefix p of $\sigma = pu, \sigma \in P_{red}$ is also a member of $p \in P_{red}$.
- $P_{blue} = P_{red}.\Sigma$ is the prefix-closed *blue-prefix* set of input strings in P_{red} extended with one extra input symbol.
- $P = P_{red} \cup P_{blue}$ indexes the rows of OT divided into red and blue prefixes.
- $S \subseteq \Sigma^*$ is a non-empty suffix closed set which indexes the columns of OT.
- $T : ((P \cup P.\Sigma) \times S) \rightarrow \Omega$ models the table entries obtained from SUL query observations and is defined as $\forall p \in P \cup P.\Sigma$ and $\forall s \in S, T[p, s] = \lambda(\delta^*(q_0, ps))$.

Note that $q_0 \in \mathit{Queries}$ is the initial query to observe the initial output of the SUL and in most cases is just an empty string. δ^* and λ are previously defined in Section 3.2. The rows of the observation table OT are labeled with the strings in P, and the columns are labeled with S strings. Each cell of the table is mapped to the

Table 3.1: A sample L* Mealy observation table

(a) Closed: yes, consistent:no

	ϵ
ϵ	1
1	0
11	1
110	0
0	1
10	1
111	1
1100	1
1101	1

makeConsistent()

(b) Closed: yes, consistent: yes

	ϵ	0	1
ϵ	1	1	0
1	0	1	1
11	1	0	1
110	0	1	1
0	1	1	0
10	1	1	0
111	1	1	0
1100	1	1	0
1101	1	0	1

output set of T where its inputs are the corresponding row and column labels of the cell ps.

Table 3.1 shows an example OT and the way it is expanded by the L* algorithm to become *converged*, which will be defined later. To construct the table, first, the upper part is indexed by P_{red} elements, then the lower part by P_{blue} elements. The S elements index the columns of the observation table. Both the P and S sets are initialized to $\{\epsilon\}$. Membership queries for each $row(p)$, if missing an entry, are executed on SUL to store the outputs as $T[p, s]$ which in fact represents the tuples of the table entries. A successful convergence of the OT to a Moore machine requires the fulfillment of two conditions: (i) *closure*, and (ii) *consistency*. An observation table OT is said to be closed iff:

$$\forall p_1 \in P_{red}, \forall \sigma \in \Sigma, \exists p_2 \in P_{red} : row(p_1.\sigma) = row(p_2) \tag{3.1}$$

Similarly, OT is consistent iff:

$$\forall \sigma \in \Sigma, \forall p_1, p_2 \in P_{red} : row(p_1) = row(p_2) \rightarrow row(p_1.\sigma) = row(p_2.\sigma) \tag{3.2}$$

There are three ways to generate active queries. Either: (i) there is a missing entry in a table that should be filled in, or (ii) a filled table is not *closed*, or (iii) not *consistent* such that it should be expanded to fulfill these requirements. When all the cells of OT are filled in, the table is checked for consistency and closure. In the case of inconsistency, the table is extended by adding a new suffix αs to S where $p_1, p_2 \in P, \alpha \in \Sigma, s \in S : row(p_1) = row(p_2) \rightarrow T[p_1\alpha, s] \neq T[p_2\alpha, s]$, and then filling in the new cells with membership queries. If OT is not closed, then L* searches for a prefix $p \in P_{red}$ and an input symbol $\alpha \in \Sigma$ such that $\forall p' \in P_{red} : row(p\alpha) \neq row(p')$. Then, it appends $p\alpha$ to P_{red} and fills the subsequent missing fields in the table with membership queries.

When the observation table becomes converged (i.e., closed and consistent), it can be used to define a congruence structure on the red prefix set and thus construct

a quotient automaton. This hypothesis $H = (Q, q_0, \Sigma, \Omega, \delta : Q \times \Sigma \to Q, \lambda : Q \to \Omega)$ can be constructed over input alphabet Σ according to the subsequent rules:

1. $Q = \{[p] : p \in P_{red}\}$, i.e., the state set,
2. $q_0 = [\epsilon]$, i.e., the initial state,
3. $\delta([q], \alpha) = [q.\alpha]$, i.e., the transition relation,
4. $\lambda([p]) = T[p, \epsilon]$, i.e., the output function. $\forall p, p' \in [p], \lambda([p]) = \lambda([p'])$ because p and p' are row equivalent.

Once a model has been constructed, an equivalence oracle is needed to make equivalence queries in order to identify whether the inferred Moore automaton is behaviorally equivalent with the target SUL (M). L* Mealy terminates with a *yes* answer to this correctness test. However, if the Oracle provides a counterexample β such that $\beta \in L(M) \iff \beta \notin L(H)$, the learning algorithm extends the OT with all the prefixes of β and continues its membership queries.

Chapter 4

Model Checking

Even though LBT makes use of model checking as a black-box tool, it is helpful to have some understanding of the fundamentals of the *model checking* approach. Model checking is a technique to solve a satisfiability problem which is the question of whether a logical formula ϕ is true over a model M_{sys} for some assignment α. This is denoted by $M_{sys}, \alpha \models \phi$ where M is a model of the target system usually represented as an automaton (e.g., a Kripke structure) and ϕ is a specification that is formulated in a logic (e.g., temporal logic) describing the behavior of the target system.

To exemplify such a verification process, suppose that $L(M_\phi)$ and $L(M_{sys})$ are the formal languages that the automata of the specification M_ϕ and the system model M_{sys} capture. Intuitively, model checking means to check if $L(M_{sys}) \subseteq L(M_\phi)$. That is to say, whether the behavior of the model is within the limits of the specified requirement. One approach to do this check is to negate the specification formula to $\overline{\phi}$ and construct its automaton model to explore the interactions of $M_{\overline{\phi}}$ and M_{sys}. Hence, the *product automaton* of these two models can be built to search for any violations of the formula. If the language of the product automaton is empty, model checking generates a *true verdict* indicating that the model satisfies the formula:

$$L(M_{sys} \times M_{\overline{\phi}}) = \emptyset \rightarrow M_{sys} \models \phi$$

On the other hand, if the language $L(M_{sys} \times M_{\overline{\phi}})$ is non-empty, it means that there exists at least a witness to the violation of ϕ. This witness is in fact an input string $x \in \Sigma^*$ such that $x \in L(M_{sys})$ and $x \notin L(M_\phi)$. In this case, model checking results in an error and may or may not report a witness or a counterexample (e.g., a path or a state violating the requirement ϕ).

4.1 Formal Requirements Languages

Temporal Logic

Various formal notations have been proposed to describe the specifications of embedded systems during different phases of development. Some of these are considered in the survey [61]. These notations require precise syntax and semantics that can cover desired behaviors of target systems. For example, temporal logic can capture sequences of events over time. In general, temporal logic is a particular branch of modal logic with special modal operators to reason about time. Linear-time temporal logic (LTL) [96], computation tree logic (CTL) [34], and the superset CTL* [44] are the most common examples of these logics.

A significant application of temporal logic is in formal verification where it is utilized as a specification language for modeling behavioral requirements of embedded systems.

Linear-Time Temporal Logic (LTL)

LTL is the most widely used logic in the model checking community to express sequential behaviors of embedded systems over time. An LTL formula has a linear-time perspective that considers an infinite sequence of states where each state represents a unique point in time with a deterministic transition to only one successor. In LTL, time is discrete and can be extended infinitely to capture all desired past and future events. The time reference in LTL is *now*, which is the current point in which a formula is being evaluated. All the other modalities are regulated concerning their respective time difference to the current state.

Special syntax is required to translate a natural language property into LTL form. Each formula is composed of a finite set of atomic propositions (symbols) linked with boolean and temporal operators. These connectives help to specify in which state/states the property should hold.

Syntax of LTL

LTL can be divided into propositional linear-time (PLTL) and first-order linear-time (FOLTL). The most fundamental building blocks of PLTL are summarized in Table 4.1. Since temporal logic, in general, is an extension of propositional logic, it inherits the propositional operators that describe configurations of individual states or events. Note that PLTL only supports finite user-defined symbolic data types.[1]

PLTL formulas can also express the relative ordering of states, events, and properties in the past and future: e.g., *specification B occurs sometime after A*, *B never occurs after A*, etc. A very common pattern is a combination of *trigger*

[1]LTL does not support infinite data types such as integers or floating points. Such infinite data types are supported in first-order LTL (FOLTL). However, the model checking problem for FOLTL is generally undecidable.

Table 4.1: Propositional and Temporal Operators of PLTL

Proposition Forming Operators	Intended Meaning
`x = c`	variable `x` equals constant `c`
`x != c`	variable `x` does not equal constant `c`
`p & q`	`p` and `q`
`!p`	not `p`
`p | q`	`p` or `q`
`p -> q`	if `p` then `q`
`p <-> q`	`p` if, and only if `q`
`p XOR q`	`p` exclusive or `q`
Temporal Operators	**Intended Meaning**
`G(p)`	`p` is true in all future states
`F(p)`	`p` is true in some future state
`X(p)`	`p` is true in the next state
`(p U q)`	`p` is true until `q` is true
`Y(p)`	`p` was true in the previous state
`H(p)`	`p` was true in all previous states
`O(p)`	`p` was true in at least one previous state
`(p S q)`	`p` has been true since `q` was true

and *response* events in time that captures many time-invariant immediate-response specifications on a system [75, 42]:

```
G(trigger -> X(response))
```

This specification pattern concerns the generic requirement that: "*it is always the case that (G-operator) if the property* `trigger` *holds, then the property* `response` *must hold at the next time sample (X-operator).*" Both *safety* properties (i.e., something bad should not happen) and *liveness* properties (i.e., something good should happen eventually) can be represented in PLTL. For example, `G(!bad_property)` or `F(good_property)`. Also, formulas such as `GF(good_property)` exemplify the specification of *fairness* properties (i.e., something good should happen infinitely often) in PLTL.

A counterexample of a safety property is generically a finite path of length k from the initial state to the state which violates the property. On the other hand, a counterexample of a liveness property includes an infinite sequence of states where the desired condition does not hold at all. If the SUT is a finite-state model, an infinite counterexample implies some states repeatedly appear in the sequence infinitely often. This can be represented as a *lasso-shaped* graph[2] which loops over

[2]A lasso is a counterexample with a finite prefix leading to a bad cycle representing a non-terminating violation of a liveness property [103].

potential violating states. Since the execution of an infinite test case is not practical, the loop can be unrolled once or twice and be included in the test case such that it becomes finite. This *finite truncation* is essential to extract concrete test cases from these infinite counterexamples.

Semantics of LTL

The precise semantics of LTL expressions are defined based on a deterministic Kripke structure $K = \langle Q, q_0, \Sigma, \delta \subseteq Q \times Q, \lambda : Q \rightarrow 2^{AP} \rangle$ to model the target system, an LTL property $\phi \in LTL$ and a path $\pi = q_0 q_1 q_2 ...$ (a sequence of states) in K for which the property is to be evaluated. The satisfiability condition $K, \pi \models \phi$ denotes that ϕ holds for the path π in the model K whereas if it does not hold, we write if $K, \pi \not\models \phi$. We let, $p \in AP$ and $q \in Q$ denote a proposition and a state of K respectively. Let $\pi^i = q_i$, i.e., the ith state in the path π. The LTL satisfaction relation is inductively defined as follows:

$$K, \pi \not\models \bot \tag{4.1}$$
$$K, \pi \models \top \tag{4.2}$$
$$K, \pi \models p \iff p \in \delta^*(q_0, \sigma_0, ..., \sigma_i) \tag{4.3}$$
$$K, \pi \models \neg\phi \iff K, \pi \not\models \phi \tag{4.4}$$
$$K, \pi \models \phi_1 \wedge \phi_2 \iff K, \pi \models \phi_1 \wedge K, \pi \models \phi_2 \tag{4.5}$$
$$K, \pi \models \phi_1 \vee \phi_2 \iff K, \pi \models \phi_1 \vee K, \pi \models \phi_2 \tag{4.6}$$
$$K, \pi \models \phi_1 \rightarrow \phi_2 \iff K, \pi \not\models \phi_1 \vee K, \pi \models \phi_2 \tag{4.7}$$
$$K, \pi \models \bigcirc\phi \iff K, \pi^1 \models \phi \tag{4.8}$$
$$K, \pi \models \Box\phi \iff \forall i \in N : K, \pi^i \models \phi \tag{4.9}$$
$$K, \pi \models \Diamond\phi \iff \exists i \in N : K, \pi^i \models \phi \tag{4.10}$$
$$K, \pi \models \phi_1 \cup \phi_2 \iff \exists i \in N : K, \pi^i \models \phi_2 \wedge \forall k, 0 \leq k < i : K, \pi^k \models \phi_1 \tag{4.11}$$

4.2 Model Checking Techniques

Model checking is a large subject area that has been tackled by many scientists from different research domains throughout the years. Several model checking algorithms have been proposed within the classification of *symbolic* [80], *explicit* [78] and *bounded* [23].

The difference between symbolic and explicit-state model checking lies in the way the state space of a model and its requirement specifications are represented and also in the approach to search for a witness for falsifying the specifications. For example, symbolic checkers like NuSMV [32], and VIS [26], use *binary decision diagrams* (BDD, see e.g., [80]) to construct the state space and analyze it symbolically. On the other hand, explicit model checkers such as SPIN [67] or SPOT [41],

employ reachability analysis methods (e.g., depth-first search) to explore a graph representation of the state space and find potential violations.

Both approaches translate (TL) requirements into automata where the automata representation can be either symbolic (e.g., Boolean functions) [35] or explicit (e.g., Büchi automata) [39]. A symbolic model checker tries to encode transitions into labeled symbolic forms (e.g., a BDD), order variables and perform other optimizations to simplify the automata construction and save some space in memory. However, the nonemptiness tests are usually computationally expensive. In contrast, an explicit-state model-checker tries to construct a Büchi automaton from the negated LTL formula which can take up a large space in memory and even result in state-space explosion when combined with the SUT model in the model-checking phase.

A BDD is a rooted, directed, acyclic graph structure used to reason about Boolean functions representing transition systems [103]. In general, a BDD is a canonical Boolean encoding for a Boolean function. It is a compact representation given an ordered set of state variables. Model-checking with BDD is performed by transforming the model-automaton product into this structure using a *fixed-point* algorithm and then searching for a path from the root to the terminal vertex labeled 1 (i.e., a potential counterexample). Even though a BDD, in theory, can easily handle a large state space, its efficiency highly depends on finding an optimal order of the Boolean variables. This is due to the substantial impact of variable ordering on the size of a BDD graph such that it can become (not necessary always) a linear representation for the best ordering and exponential for the worst case.

On the other hand, in explicit model-checking, a requirement formula is complemented and then turned into an automaton representation to be composed with the model under verification (see also [115]). As long as this new model-automaton product does not accept any input string, the specification is correct concerning the model of the target system. Otherwise, the model checker finds a trace (a path) from the initial state to the state which violates the given requirement.

SMV Modeling Language

Symbolic model verifier (SMV) [80] is the modeling language of NuSMV to represent state machines. It is composed of signal definitions, structural declarations, and expressions. It is designed to support the modeling of finite state systems with a set of restricted data types including boolean, symbolic enumerated, bounded integer and bounded arrays of these. The language allows a complex system to be decomposed into modular units capable of being instantiated as many times as needed. At the moment, only the synchronous composition of modules is supported, which concerns the monotonic computation of all composed modules in a single step.

An SMV file describes both the model of a system and its specification. Deterministic and non-deterministic state systems are allowed to be defined in the language with the help of available constructs. For example, *MODULE* defines the main method of a system, *VAR* and *IVAR* are to express state and input variables

respectively, *ASSIGN* is to define transition relations of the states and variables, and *LTLSPEC* is the construct where an LTL specification can be specified.

Bounded model checking

Bounded model checking (BMC) is a *SAT-based* (i.e., *satisfiability solver*) approach with the aim to use *constraint solving* techniques to solve the model checking problem [37]. It is used in NuSMV as a complementary method to BDD-based symbolic model checking when the state space exceeds the capacity of the BDD approach. In BMC, a symbolic representation of the model and a requirement are jointly unwound for a given number of steps k to obtain a SAT formula. The formula is satisfiable by a SAT solver if there is a counterexample for the requirement up to length k. Unlike the BDD approach which treats the model and the property as a total boolean function to (totally) prove the absence of errors, BMC searches for a counterexample trace of the model whose length is bounded by k and, therefore, can only prove the existence of counterexamples, not their (complete) absence.

Model Checking with NuSMV

NuSMV [32] is a symbolic model checker that was first developed as a joint university project between the US and Europe. It supports both BDD and BMC techniques to check the satisfiability of the so-called symbolic model verifier (SMV) models. From Version 2, the software has become available to public researchers as *open source* which encourages their participation in its development. The current version of NuSMV is constrained to model-checking synchronous finite-state systems against LTL and CTL requirements formulas. The tool has provided the user with a set of data types, including Boolean, enumerated, bounded integers, words and arrays, to define various finite models. However, more recently, a new extension of the software is distributed under a separate tool called NuXMV [29]. This provides extensive support for infinite datatypes such as real numbers and unbounded integers.

Extracting counterexamples from NuSMV is straightforward as it provides the state trace and the input trace simultaneously in its output. These traces include the sequences of input and subsequent state and output changes that happen from the initial state to reach a property violating state. Hence, a counterexample can be conveniently obtained by simply filtering the NuSMV output using string manipulation features of any programming language.

Chapter 5

LBT for Automotive Software

This Chapter discusses learning-based testing for automotive software. The systems under test are black-box ECU applications where three relevant software testing questions are studied, namely the requirements modeling, learning efficiency and error discovery capabilities of the LBT approach.

5.1 Behavioral Modeling in PLTL

The effectiveness of LBT for a specific domain depends on many different objective measures such as test session length, achieved coverage and discovered errors, etc. In the automotive sector, safety-criticality, low test latency, and complexity are typical characteristics of embedded software. Hence, a mature testing strategy for this domain has to cope with these needs. This begs the question of a need for behavioral testing at all in this sector, and the maturity of technologies that are suited to behavioral modeling. However, these questions have already been addressed in Chapter 1.

LBT offers an automated technology that can achieve high test throughput for low-latency SUTs. Compared with manual testing, it can reach higher SUT coverage in shorter time at lower cost [74]. Furthermore, LBT supports agile development methods, since test suites are dynamically and adaptively matched to black-box SUT behavior at run-time. Thus, code refactoring does not affect the quality of the test suite. However, like any new testing technology, there is a need to consider how it can best be integrated with existing software quality assurance (SQA) methods. According to the emerging safety standards such as *ISO26262*, for safety critical systems, at least a semi-formal specification of safety requirements is highly recommended [9]. Given the rapid pace of technological and regulatory changes within the automotive sector, the impact of research on behavioral modeling and testing is therefore potentially quite high.

In LBTest, behavioral requirements are modeled in propositional linear temporal logic (PLTL) due to its decidability for model checking problems. Also,

extracting test cases from counterexamples in PLTL is much more straightforward. This is due to the fact that the LTL model of time has a simple linear structure, using the positive integers $0, 1, 2, ...$ Hence, counterexamples to behavioral requirements expressed in LTL are also linear and test cases can be extracted conveniently. Regarding the previous studies in the literature [74], only a small percentage of behavioral requirements cannot be directly modeled in PLTL.

Direct Modeling Issues

Not formalisable. One situation where this problem arises is where two or more traces of behaviors need to be compared. This is not feasible in a linear time model. For example, consider the requirement "*if* $x_1, x_2 \in \Sigma \ : x_2 > x_1$ *then* $t_2 < t_1$" which specifies that "*the greater the value applied to the input* x *is, the smaller will be the time it takes to finish a task* T." This requirement requires to compare the outcomes of two alternative execution scenarios, which is not directly feasible in LTL. Since all possible pairs of execution paths have to be quantified (for a model-checking problem), perhaps CTL is a more appropriate approach to take in this case.

Ambiguity. Another issue appears when the informal black-box requirements contain ambiguous descriptions that cannot be resolved without introducing new symbolic output variables. For example, when there is no concrete definition of a *verification period* and a *start-up time* in an ECU application documentation. Domain knowledge from test engineers turned out to be required to resolve the ambiguity of a corresponding requirement. Also, expressing such complex (sequential) timing guards in an LTL formula is not convenient. To reduce the complexity and resolve the ambiguity, two new symbolic outputs (namely `VTime` and `startup`) are defined based on the obtained information to represent the state of these timing guards and hide the actual computation in the wrapper. This way, such requirements *can* be expressed in LTL as simple stimulus-response patterns described in Section 4.1.

Conceptual parameters. Consider a requirement parameter that is not related to any actual internal or input/output variables of the SUT. This can be, e.g., the remaining fuel in a fuel tank "in reality" which cannot be known by any means other than emptying the tank and measuring by hand. This is a problem of conceptual modeling, where for testing purposes we can assume that the actual remaining fuel can be known by an external observer. In LBT, the communication wrapper (see Section 2.2), is assumed to be the external observer by knowing the initial fuel level in the tank and calculating all subsequent fuel levels using numerical integration of the consumption rates. Therefore, communication wrapper can hide ambiguous, complex parameters and computations from the informal requirements to simplify their corresponding LTL representations.

Requirement Redefinition

Sometimes, informal behavioral requirements are not directly formalisable into LTL, but they can be reformulated such that they become expressible in LTL. In general, reformulation can include rephrasing, merging requirements, hiding complex and unspecified computations in the wrapper and etc. One example of reformulation is when the informal requirements do not systematically model all the corner cases, names, and references necessary for an unambiguous translation into formal requirements. Table 5.1 shows a merging example of two informal requirements (namely **Req 1** and **Req 2**). What is usually desired in black-box testing, is a description of an input-output behavioral mapping (function). This is not completely available in the example since the informal requirements made use of internal (glass-box) signals (namely `variable1` and `variable2`). These internal variables should be removed by merging their known relationships to each other and the input and output variables. The result (i.e., **Merged Req 1**) is a pure black-box requirement.

Sometimes the description of the informal requirements includes connections between past and future events. To reformulate such requirements without using past-time LTL operators (c.f. Table 4.1) is a challenging task. For example, a specification may require certain events to happen when another event has occurred previously. Using a past-time LTL operator, in this case, would simplify the structure of the expressed formula and also avoid injecting unintended errors into the requirement.

5.2 Learning Efficiency and Error Discovery

For many ECU applications in automotive software, infinite combinatorial domains of input and output values give rise to infinite-state models, for which robust modeling and learning is needed, e.g., using *hybrid automata* [12]. Given that these applications inherit cyber-physical aspects, this adds all up to significant complexity in modeling and learning such SUTs.

In general, numerical modeling of these applications is a challenge for machine learning, since accurate models can be quite large (over 100,000 states). Besides the complexity of the SUT, this level of model detail is also influenced by the size of the input sample sets and the coarseness of the output value partitions. For output partitioning, the test engineer must abstract the infinite data types used by the SUT (e.g. integers, floating points) into finite symbolic data types used by the learner and model checker (e.g. negative, zero and positive integers).

In addition, learning a complete and detailed behavioral model has to be conducted in a reasonable amount of time. The performance of LBT mostly depends on the latency of the SUT, which can be a bottleneck for low-latency applications. Both low-resolution and high-resolution modeling are feasible by tuning the input sample intervals which can affect the time granularity and the size of a learned model. This is a question of providing the right level of detail when testing a specific LTL requirement.

Req 1
If *variable1* == true
  then **output1** = failure
else if *variable2* == true or **troublecode3** == on
  then **output1** = error
else **output1** = nofailure

Req 2
While input1 == on and input2 > 400
  if input3 == off for more than 1 second
    then *variable1* = true and **troublecode1** = on
  if input3 == on
    then *variable1* = false and **troublecode1** = off

Merged Req 1
While input1 == on and input2 > 400
  if input3 == off for more than 1 second
    then **output1** = failure and **troublecode1**
  if input3 == on
    if *variable2* == true or **troublecode3** == on
      then **output1** = error and **troublecode1**
    else **output1** = nofailure and **troublecode1**

Table 5.1: Requirements merging.

The convergence properties of a final model can be used to construct/define test coverage figures. In other words, the degree of convergence of the final model is a metric to show the accuracy of the converged model to repeat the actual behavior of the SUT for all possible input samples. The convergence metric currently used by LBT is the *divergent path metric* (DPM). This measures the percentage of input/output sequences produced by the learned model which agree with the input/output behavior of the SUT. Thus a DPM convergence value of 100% means that the SUT and the learned model have identical observed functional behavior. LBT uses a Monte Carlo approach to estimate DPM. A fixed number K of random input sequences (an evaluation set), of bounded length[1] b, is executed both on the

[1]The number K of random test sequences can be user-defined in the tool configuration file. The bound b is chosen to be twice the length of all active learner queries. In this way we evaluate the learned model outside the training set.

learned model and on the SUT. This evaluation set should not overlap with the training set used to infer the model. Let δ be the number of these input sequences for which the SUT and the learned model produce the same output sequence, then $0 \leq \delta \leq K$. The estimate value is simply $DPM = 100 * \delta / K$.

The maturity of an application is a significant factor when considering the likelihood of finding undiscovered faults using a new technology. For example, an application that is in production and has been maintained for many years is unlikely to contain any serious bugs violating the requirements that have been tested. In addition, the level of model detail needs to be sufficient for the generation of failing test cases, otherwise SUT coverage would be deemed inadequate. This can also be a model-checking problem when the model is too abstract to be able to express violating sequences.

Chapter 6

Fault Injection for Safety Critical Systems

In software engineering, dependability is a broad concept that covers many aspects such as reliability and availability. However, robustness is a more specific concept with focus on the ability of software to deal with unexpected usage or users' input. To evaluate the robustness of safety-related embedded software, various analytic and dynamic methods are available. Fault injection (FI), among other empirical approaches, helps to exercise corner cases that are not triggered during normal operation but only in the presence of faults. The three central questions an FI method should answer are: when, where and how to insert specific faults into software or its underlying hardware to observe realistic failure outcomes. This chapter reviews some basic principles of FI along with surveying related research on the thesis topic.

6.1 Hardware Faults in Embedded Systems

In general, faults[1] in embedded systems can be classified according to their hardware or software roots [22]. Software faults, also colloquially known as software bugs, are essentially related to design errors at various development steps. These dormant faults are hidden in the implementation, and their signature is only observable at runtime when the affected area is being executed. In fact, software faults are permanent in the code but transient in the behavior.

Hardware faults, on the other hand, have a broader spectrum due to their physical origins that can be affected by the environmental disturbances. These faults are not limited to design errors and can cover wear and tear, aging, and other external causes. One standard classification based on their temporal behavior (duration) is *permanent, transient* and *intermittent* faults. Permanent faults, as the name suggests, are those with perpetual lifetime that once they appear in the

[1] A fault is a triple consisting of a time, a place and a fault type.

system remain constant forever until the fault source is removed. Transient faults, however, have the opposite behavior. These arise for a relatively short time span and then disappear without any recurrence. Lastly, intermittent faults lie between these two extremes. They remain permanently in the system but have transient behavior. Intermittent faults may occur at irregular intervals, often with random distributions.

Against this background, an interesting subject of study is to investigate cause and effects for each category. For example, permanent faults may appear as stuck-at-values whereas transient faults usually arise in the form of inversion (e.g., bit-flips). In general, component damage, wear-out and unstable hardware (due to development errors) may result in permanent and intermittent faults. Furthermore, disturbances such as shock, heat, radiation, electromagnetic interference or power fluctuations can induce transient faults in the affected area.

Safety-critical embedded systems are susceptible to hardware faults in two ways; due to direct risk of physical damage to the system and indirect impact on the behavior of dependant safety-related software components. In both cases, the consequences of hardware faults can be modeled and simulated to investigate significant or insignificant impacts on the system's overall function [111].

Faults are usually latent in the code or circuitry but can surface during the operation and make incorrect changes in a system state also named errors. Although a single fault has limited scope in the affected area, it can introduce multiple propagated errors throughout the system. For example, depending on where a simple bit-flip occurs, **timing** (e.g., transition delay), **data** (change of memory value) or **control-flow** (corruption of a program counter value) errors might happen. FI experiments can examine error propagation from a fault location to other hardware or software elements. Mean time to failure (MTTF), mean time between failures (MTBF), fault latency and error latency[2]are examples of such measures. However, their traditional objective has been to determine reliability, availability, and other dependability parameters of a system under study (SUT).

FI experiments can be distinguished based on the following principles [99]:

- *Reachability*: the ability to induce a fault or its subsequent error in the desired area.
- *Controllability*: the capability to control fault parameters (time, location, type) in the reachable area.
- *Intrusiveness*: the degree of change in the target system required to implement an FI method.
- *Repeatability/reproducibility*: the possibility to replicate fault experiments.
- *Efficiency*: the required effort (time and labor) to perform FI testing.

[2]Fault latency is the time between the appearance of a fault and the respective error. Error latency is when a propagating error is detected by the necessary mechanisms after some delay.

- *Effectiveness*: the ability to trigger various error-handling mechanisms.
- *Observability*: the ability to precisely log system states under faults and capture all subsequent errors.

Among the above principles, being non-intrusive is an important property of an FI approach. More specifically, intrusiveness implies additional overhead due to instrumentation of the SUT that can manifest incorrect behavior after FI. Therefore, intrusiveness should be at its minimum possible level to reduce this undesired overhead and achieve accurate results comparable with real-world behaviors. In addition, reachability and controllability measures are tightly connected to effectiveness in the sense that the more fault sites and fault parameters that are covered by an FI approach, the more error-handling mechanisms can be examined. Observability is also very fundamental, since, without enough information about the SUT state changes due to faults, no further analysis will be accurate.

6.2 Conventional Fault Injection Methods

Various approaches to fault injection have been proposed in the literature over the years. These can be grouped into **hardware-implemented (HIFI)**, **emulation-based (EFI)**, **simulation-based (SFI)** and **software-implemented (SWIFI)** techniques. This classification is based on whether faults are injected into physical hardware, implemented software, or into models of the two [95].

Early FI techniques were experimental validation methods applied directly to the actual hardware implementations or prototypes. These HIFI techniques use external sources to introduce faults on the physical components of the system (e.g. by electromagnetic interference, short-circuiting connections on electronic boards and corrupting pin-level values of the circuit prototypes). HIFI often represents the best solution with respect to realistic results and performance. However, in addition to imperfect observability and controllability, HIFI methods tend to be costly and have a high risk of damaging the injected system also leading to low repeatability [33].

Emulation-based fault injection (EFI) has been introduced as a realistic alternative for HIFI to overcome the high cost and the associated risk of damage. In this method, often, field programmable gate arrays (FPGAs) are used to emulate the target system and synthesize faults in an emulation environment. Hence, EFI reduces execution time and speeds up fault simulation in comparison with SFI, while keeping FI experiments safe and reproducible compared to HIFI. Since EFI exploits FPGAs for effective circuit emulation, it is only feasible when there is a synthesizable model of the target hardware. Unfortunately, due to the growing complexity of embedded hardware, this is not always the case, and hence, EFI has limited coverage and applicability [95].

SFI [95] denotes techniques that utilize model artifacts to reproduce fault effects in a simulation environment *(fault simulator)*. It is a broad line of research that

includes both hardware modeling (e.g., VHDL models) and software modeling (e.g., Simulink models). Here, a fault is either injected into hardware models (i.e., model modifications) or software state (i.e., state change) to simulate a faulty behavior of the target system. SFI approaches can be further divided depending on when the injection occurs, either before runtime (compile-time) or after the simulation is being executed.

SFI methods can be used early in the development process since they are not reliant on the availability of hardware prototypes and the respective FI device. In addition, they impose no risk to damage the system in use. Also, they inherit high controllability and observability properties available in modeling and simulation tools. Most of the hardware modeling methods are not intrusive to software and benefit from high reachability to target elements. Apart from these advantages, SFI techniques, in general, suffer from significant development effort related to modeling complex hardware architectures (e.g., processor models). Nevertheless, due to trade-offs between the accuracy of models and simulation time, SFI may either lack precise fault models or have poor time resolution which can lead to fidelity problems.

SWIFI is a special case of SFI where the SUT is a complex software component running on a large microarchitecture system[3] with caches, memories and other peripherals. This approach is suitable for target applications and operating systems that may not be feasible in HIFI. SWIFI instruments production software such that faults can be inserted into the program or its state either before runtime or during the execution of the software [28, 62, 58]. Hence, a significant challenge of SWIFI is its undesired intrusion which may perturb the normal operation of the SUT and skew the FI results. Furthermore, it is limited to those locations reachable by the software (e.g., MMUs, registers).

More recently, **model-implemented (MIFI)** and **virtualized-based (VFI)** techniques have been proposed within SFI techniques. MIFI which is introduced by [112], extends the traditional FI classification by utilizing additional high-level model artifacts to simulate fault effects. In this view, high-level models (e.g., VHDL, Simulink) are extended with special mechanisms that when activated introduce the desired errors in the model behavior. MIFI is compatible with model-based development which is widely used for software development. It is applicable early in development stages when a detailed implementation of the SUT is not yet available. For example, a stuck-at fault in Simulink can be realized by adding a constant value and a switch to the FI location. This way, the switch output determines whether a faulty value should be connected to the next block.

Virtualization which is the technology behind virtual machines (VMs), allows a guest machine operating system or software to run on a host machine hardware [108]. This way the same physical hardware can be shared among many guest VMs

[3]A microarchitecture is a particular implementation of a certain processor architecture with details including the system/method/circuit designs. These could describe the complete operation/specification of the implemented architecture.

with efficient and flexible performance, and significant cost saving benefits. Xen, KVM, and QEMU [24, 20] are instances of machine virtualizers leveraged more frequently in literature to inject faults into virtual machines (VMs). QEMU seems to be a more attractive choice among researchers in this field. Further details about recent work in this context will be discussed in Chapter 7.

VFI is a new paradigm with the aim to overcome limitations of traditional fault injection methods. It basically adapts emulators capable of performing hardware virtualization to simulate hardware faults and observe their effects on the running software. In VFI, the emulation is mainly instruction accurate compared to cycle accurate simulation of SFI.[4] This results in significant performance improvement, while the simulation accuracy is still comparable to bare machine execution. There are even more benefits to the use of VFI approach as stated in [95]. For example, high observability, controllability, repeatability without having the real hardware and zero to negligible intrusiveness.

Despite significant improvements in FI campaigns resulting from employing virtual environments, it is not yet possible to precisely simulate some specific faults. Such faults include, for example, those that cannot be directly mapped to any emulated components (including memory, CPU registers, peripherals). Voltage glitches, clock drifts, over-currents due to short circuits, are among those that have known physical consequences but lack well-defined and deterministic translation in the simulation world to implement them [95].

6.3 Fault Injection through Active Learning Queries

This section provides an overview of our research on fault injection (FI) testing of safety-related embedded systems by combining hardware virtualization engine (QEMU) with learning-based requirements testing. The primary goals of this approach are to automate the FI testing in an architecture consisting of three components, i.e., (i) test case generation, (ii) fault injection, and (iii) formal requirements based verdict construction. First, the approach to emulate faults based on the combination of QEMU and the GNU debugger GDB are described. In the next step, an architecture is proposed to integrate QEMU-GDB with the learning-based testing tool LBTest, along with a discussion of how the toolchain manages fault injection processes.

SW/QEMU-GDB: an approach for Fault Emulation

Fault representativeness, with respect to physical reality, is a crucial concept of any FI testing approach with an influence on the effectiveness of the approach. For example, fault representativeness at register-transfer (RTL) or cycle-accurate levels cannot be fully preserved in an instruction-accurate simulator that abstracts

[4]The cycle or instruction accuracy of a simulator refers to the granularity of simulating a microarchitecture whether it executes clock cycle by clock cycle or instruction by instruction.

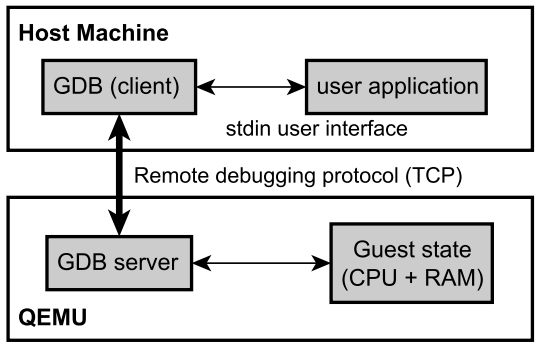

Figure 6.1: The QEMU-GDB architecture to emulate fault effects using user interface breakpoints

away many low-level details of a CPU micro-architecture. Instead, emulating the corresponding low-level fault models at CPU instruction-accurate level reduces the performance overhead due to the efficiency of such simulators. Instruction-accurate fault models usually try to emulate the consequence of lower-level faults. For instance, corruption of CMOS transistors can affect digital cells at the gate level. These can further influence more abstract components like multiplexers and registers at the RTL level. Finally, the propagation of faults can reach the instruction-accurate level where state variables (e.g., the program counter PC, processor GPRs) of a system are concerned. In other words, the faults at this abstraction level are the function of all possible faults at the gate or RTL levels. Simple hardware faults such as bit-flips and stuck-at faults with a direct impact on status variables can conveniently be represented by soft-errors. More advanced hardware faults with distributed outcomes are also expressable either by using a combination of the elementary fault models or by utilizing extra macros to introduce specific function manipulations to implement the fault.

In this thesis, the approach is to emulate hardware faults in QEMU, which is an open-source virtualization platform with diverse support for many target architectures including x86, PowerPC, and ARM. QEMU allows the execution of unmodified binary programs on a host machine by utilizing a two-step dynamic binary translation (DBT) engine. The target machine code is first translated into an intermediate representation (IR) for target-dependent optimizations and performance improvements. Then, this IR code is transformed into the host machine executable format with the help of a tiny code generator (TCG). Other hardware components outside the CPU core are emulated/simulated through C-level modules and functions that may access kernel level resources.

The integration of QEMU and GDB is a practical solution for fast and realistic fault emulation while a more inclusive coverage of the fault spectrum can also be obtained. In Figure 6.1, this QEMU-GDB architecture is depicted to provide more

insight into the integration. Here, QEMU is equipped with a serial interface to link and interact with generic developer tools such as GDB. This communication link provides detailed inspection and manipulation of internal states and boundary values of the emulated system (e.g., registers, MMUs, IOs, and peripherals). Also, it can be used to control program execution order and perform a stepwise simulation. Thus, desired fault models can be implemented without the intrusion of the SUT or QEMU.

Two somewhat different integration strategies can be considered to simulate the effect of hardware faults in embedded systems, both based on the Software GNU Debugger [1]. The first approach relies on the debugging capabilities of GDB to perform run-time manipulation of target program data and control the execution sequences. This, SW-GDB, method makes use of various software breakpoint types to introduce desired faults at a particular location within the embedded software and then inspect the behavior to discover potential errors. Such anomalies can be accurately determined by using precise user specification models of the LBTest tool.

The second method, namely QEMU-GDB, exploits emulation capabilities of QEMU to observe address and data buses of a processor and the emulated hardware peripherals. This is a mechanism to emulate hardware breakpoints similarly to stop the execution, inject faults and perform data profiling, memory access tracing, etc, [2, 70].

Comparison of SW-GDB and QEMU-GDB. The differences between SW-GDB and QEMU-GDB methods are related to their abstraction levels. In SW-GDB, a faulty behavior is emulated at the target software level. Here, debug information is augmented to the compiled program binary of the SUT. This provides the possibility to probe detailed execution of the source code. In other words, it provides the feasibility of inserting various breakpoints and accessing any data structures in the program during run-time. Hence, even complex multi-dimensional pointers can be manipulated at any desired time to inject faults and inspect the corresponding SUT response. In addition, the SW-GDB approach is not specific to a unique processor, since GDB is available for many kinds of different processor architectures. Furthermore, the intrusion is minimal and limited to the additional debugging macros. Despite these positive features, SW-GDB is, in general, slower than QEMU-GDB in terms of time overhead of its trap and exception handling mechanisms [1].

On the other hand, QEMU-GDB, which has access to the emulator processes, works at a lower abstraction level that is basically hardware-level simulation. Here, a fault can be inserted through macros which set memory access breakpoints (hardware watchpoints). Thus, this method similarly inherits the flexibility to manage fault types, locations and times dynamically with no intrusion in the QEMU or the SUT code at all. One obvious issue here seems to be the opacity of the SUT source code, which looks like a black box to QEMU-GDB. This might confine the

controllability of the program data structures to inject faults to some extent. However, these are not really needed for a very low-level FI technique where sufficient hardware domain parameters are still accessible to manipulate a hardware state. In addition, it is more convenient to implement hardware faults (other than soft errors) in QEMU-GDB, e.g., time delays, so that a broader fault type coverage can be achieved.

The Full FI Architecture of LBTest-QEMU-GDB

Figure 6.2 illustrates the architecture of the proposed FI technique containing five main components:

1. LBTest which acts as the FI manager, test generator, and the Oracle. This writes test results in separate log files.
2. An SUT-specific test harness which is basically a Python program as a communication wrapper to map faults and data types to concrete and symbolic values.
3. An FI-module to implement faults in the GDB environment.
4. An LBTest configuration file which captures all the necessary information to conduct testing (e.g., symbolic data types, specifications to be tested, the choice of learning and model checking algorithms, etc.).
5. A GDB API which is the interface to the debugger via QEMU serial terminal environment.

The main functionalities of LBTest to manage an FI test session can be summarized as follows: (i) It generates test cases that aim to discover a fault. (ii) It calls the main process of the communication wrapper to set up and initialize the SUT. (iii) It sends and receives symbolic test stimuli and the SUT observations to/from the wrapper. (iv) It reverse engineers a behavioral model of the SUT using a machine-learning algorithm specified in the configuration file, and finally (v) as an Oracle, LBTest constructs test verdicts according to the compatibility of the SUT observations with the formal specifications through model checking.

The purpose of using a configuration file in an FI campaign is to provide the option of adjusting various testing and FI parameters for the test engineer. For example, these parameters include the choice of machine learning and model-checking algorithms, some target-specific data models for the SUT API, and most importantly, test requirements, fault types, and values. Many other test parameters to handle a session, such as stop conditions, coverage measurements and directory paths, can also be defined in a configuration file.

At the LBTest level, the generated test cases are abstract in the sense that faults are encoded as symbolic values scheduled into symbolic input sequences. These abstract fault values are then resolved into concrete FI events using the

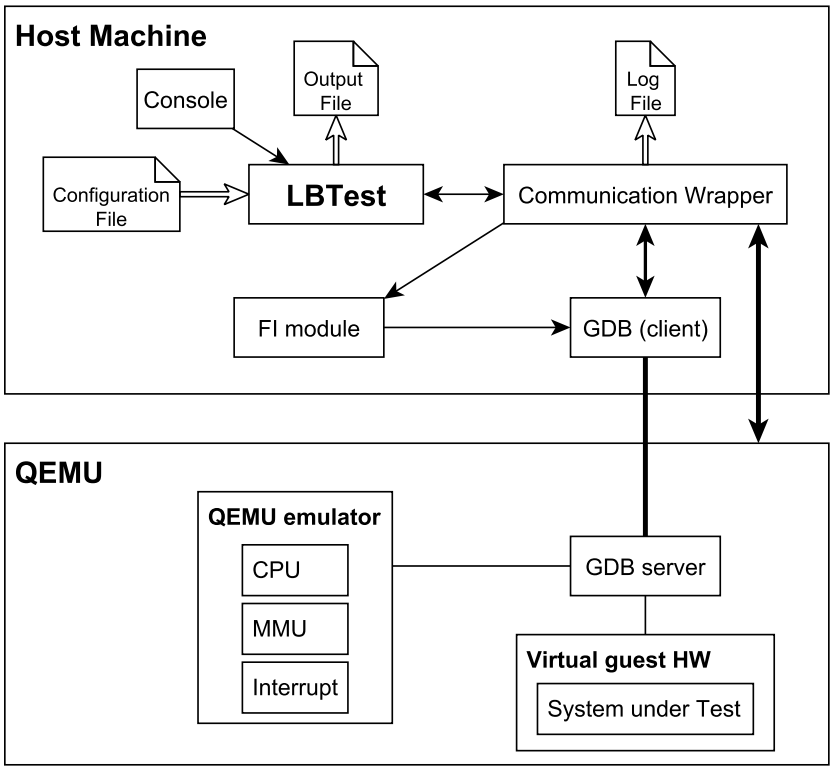

Figure 6.2: The integrated LBTest-QEMU-GDB tool chain with debugging from the GDB user interface

commmunication wrapper and according to the FI-module (see Section 6.3 for more details). Hence, by also relying on its platform independent feature, LBTest can turn into a multi-hardware level (e.g., gate, RTL and ISS levels) fault generator regardless of the kernel environment.

As mentioned above, before the communication wrapper can concretize the symbolic FI test cases into real ones, the FI-module has to define the corresponding SW- or QEMU-level breakpoints in GDB. These breakpoints are in fact similar to exception handling mechanisms that when called execute certain FI macros to implement the desired faults at a location. Then, the wrapper can activate/deactivate some of these breakpoints, depending on which are or are not needed according to the concrete fault scenarios.

Another important aspect of the proposed approach consists of the complex interactions of the communication wrapper and the FI-module. These are both user applications, where the latter is called inside the GDB process standard input (stdin) by the former. The FI-module configures the SUT and sets static data

and function breakpoints based on the configuration parameters. In case dynamic breakpoints are needed to deal with complex fault scenarios, specific Python-GDB methods are utilized. For convenience, these methods are used in a function to be called by the wrapper and manage breakpoints on the fly. With this flexibility, more fault types and locations can be covered in each test run. Finally, if any SUT exception or crash is detected as the result of FI, appropriate messages are reported to the wrapper based on a predefined contract in the FI-module.

The last two components of Figure 6.2 to be described are QEMU and the GDB client. QEMU is an individual process on top of the operating system in the host machine. Its primary task here is to emulate the hardware of the target platform and provide the necessary infrastructure to execute an unmodified binary program (i.e., SUT). QEMU can emulate the guest processor commands by converting its binary code into the corresponding host CPU executables, and also simulate target specific hardware (e.g., CAN controller) by means of user-defined C functions that exhibit the same behavior. Furthermore, QEMU comes with a lot of configuration options for its various modes, but the most relevant one for FI is the primitive support to interact with GDB. Hence, QEMU is launched with "-s" option by the communication wrapper so that it loads the binary image of the SUT and then waits for a GDB connection to start controlling the guest execution.

Finally, the GDB client in Figure 6.2 is a platform-specific program that implements commands and macros to step the SUT execution and access its internal data to modify the SUT state and inspect the subsequent effects.

LBTest: an Automatic FI Controller. Criteria that make LBTest an appealing tool choice for FI are: (i) a high degree of test automation, (ii) a high test throughput that can match the low latency of real-time embedded applications, (iii) the ability to systematically explore combinations of fault types, locations and times as test session parameters. To achieve the latter, LBTest supports useful combinatorial test techniques such as n-wise testing [13]. These are necessary to handle the combinatorial growth of both the SUT state space and the (time dependent) hardware fault space. Other important features of LBTest include scalability to large distributed systems as well as precise verdict construction from behavioral requirements. The latter is essential for FI testing the robustness of safety-critical embedded systems.

The role of Test Harness in Fault Injection. Traditionally, an LBTest communication wrapper functions as a test harness to encapsulate the SUT and its environment. It marshals each test stimulus and the corresponding SUT response back and forth. It also performs test set-up and tear-down activities necessary to isolate each test experiment. Alongside these basic functionalities, the wrapper has more sophisticated responsibilities, including the following.

1. The wrapper abstracts infinite-state systems into finite-state models. This abstraction is done through data partitioning.

2. Wrappers support real-time abstractions using synchronous (clock-based) or asynchronous (event-based) SUT response sampling methods.

3. The semantics of FI is implemented by using the wrapper to control fault injection and simulate desired failures (e.g., a sensor or communication failure). This principle is the gateway to robustness testing using LBT [88].

In our approach to FI, the wrapper is the second component of the FI chain. It calls QEMU and GDB in separate processes and instantiates their serial communication links (using standard-input/output). Through these links, the wrapper can send GDB/QEMU commands to configure test setup, start/stop the execution of the SUT, inject faults, wait for results, monitor variables and perform post-processing on the desired outputs.

The symbolic fault values within the test cases generated by LBTest are translated into GDB-executable queries. These commands, in turn, implement the fault semantics according to their specified attributes. Hence, each test case can assign its combination of fault parameters (test case fault values assigned to fault variables at specific times). Since our method is based on breakpoints to determine FI points (i.e., places where injection should occur), they are defined during test setup using the FI-module but are held deactivated before executing the SUT. The wrapper can then enable/disable the corresponding breakpoints according to every fault parameters.

Supported Fault Injection Techniques

As discussed previously, SW-GDB and QEMU-GDB emulate faults in two different abstraction layers where each fault can be described by three attributes of *space, type, duration*. Hence, the scope of the two interfaces in supporting the mentioned attributes is slightly different. Below, the extent of the fault properties are specified in each case.

Fault Triples. Each fault experiment controls three fundamental attributes.

- *Fault Space*: Fault space defines the time and the location in which the fault will affect the target object. It is a set of unique point pairs to distinguish fault trigger events.

- *Fault Type*: Fault types are symbolic values which determine how data or state is corrupted. Traditional fault types include stuck-at, bit-flip, etc.

- *Fault Duration*: Fault duration is the lifetime during which a fault is active. In other words, it defines the statistical distribution of fault appearances.

Fault Injection via SW-GDB.

1. *Fault Space*: To specify fault time and location parameters of SW-GDB is relatively easy, since here, a compiled SUT program provides access to the symbol information for tracing its execution. This way, event and source code line breakpoints can be set to trigger exceptions at run-time and inject faults by manipulating available program arguments. Furthermore, using conditional breakpoints here can bring more flexibility in creating and dealing with specific fault cases.

2. *Fault Type*: Thanks to a GDB-Python integrated API, any manipulation function can be performed at a fault point. With this in mind, not only simple fault types (e.g., inversion, value freezing) but also more advanced ones (e.g., transition delay, communication corruptions) can be realized on variables, registers, and memory cells.

3. *Fault Duration*: Permanent and low-frequency intermittent faults are convenient to be implemented in SW-GDB. However, the problem arises when the desired fault is a low-interval intermittent type. The main reason is that the breakpoint artifacts such as trap instruction, exception handling, halting the program execution, data manipulation, etc. are costly in terms of overhead that is implied during a test session.

Fault Injection via QEMU-GDB.

1. *Fault Space*: QEMU-GDB provides two breakpoint-based approaches to define fault time and location parameters in an FI experiment. First, by setting breakpoints in the QEMU source code it can control the emulation process and modify hardware state information that is available to QEMU (e.g., registers, virtual memory, timers, interrupts, etc.). Second, by setting hardware access breakpoints (access watchpoints) as a measure to determine where or when a hardware resource is accessed and perform the desired fault accordingly. A special case of the latter option is the program counter (*pc*) which identifies the address of the SUT program function that is being executed. This helps, for example, to access program points in a function by knowing the relative distance to a function start point and set a *pc* watchpoint there.

2. *Fault Type*: Correct manipulation of the target data after reaching an FI point results in the emulation of the desired fault. This includes, for example, using simple data masks and custom functions to implement traditional fault types (e.g., stuck-at, bit-flip). Furthermore, by adjusting the latency of individual QEMU hardware commands, delay faults can be implemented.

3. *Fault Duration*: Since hardware breakpoints are treated as exceptions in QEMU-GDB, they impose lower overhead on the program execution compared with other methods. Also, due to the simplicity of the available data

structures in hardware, data manipulations (e.g., masks) are implemented more conveniently. With these in mind, in addition to transient and permanent faults, low-latency intermittent faults are also accurately applicable.

Chapter 7

Related Work

In what follows in this section, we shall be investigating similar works and FI tools presented in the literature. This survey will highlight the basic ideas, advantages and disadvantages, and differences between the presented approaches and our work.

7.1 LBT related

A wide variety of formal notations have been proposed to support requirements testing of safety critical systems over the years, an extensive survey is [61]. In the context of embedded systems, considerable attention has been devoted to the use of *temporal logic* [50] for requirements modeling.

In [97] the authors present a large-scale study of modeling automotive behavioral requirements. For this they use the Real Time Specification Pattern System (RTSP) presented in [75]. This pattern system is an extension of the earlier Specification Pattern System (SPS) of [42]. RTSP requirements are semantically mapped into a variety of temporal logics, including linear temporal logic (LTL) [43], computation tree logic (CTL) [43] timed computation tree logic (TCTL) [11] and metric temporal logic (MTL) [11]. The intention of pattern languages such as SPS and RTSP is to allow the semantics of requirements to be easily understood and validated.

Drawing upon five informal requirements documents, [97] assembles a total of 245 separate requirements for 5 different ECU applications. Of these, just 14 requirements (3% of total) could not be modeled. The authors conclude that TCTL provides an appropriate temporal logic for behavioral modeling within the automotive domain. However, their proposal has the significant disadvantage that model checking is undecidable for TCTL (see [97]), and hence it cannot be used for automated test case generation. This contrasts with LTL, used in our own research, which has a decidable model checking problem. A later study [49] of RTSP at Scania CV has broadly confirmed these statistics. It estimated the relative frequency of successfully modeled behavioral requirements at about 70%. Just 6% of the requirements could not be modeled. Our own research here seems to suggest

that a large proportion of commonly encountered ECU requirements can already be modeled with LTL.

Methods to reverse engineer state machine models of automotive ECUs using similar machine learning techniques to our own have been considered in [104]. Here, the authors used an industry standard test runner PROVEtech [3] to perform hardware-in-the-loop testing of components, and reverse engineer SUT models using their RALT tool. These methods were shown to be effective for third-party software where source code was unavailable. However, in contrast to our own work, formal requirements and model checking were not used. In [117] the effectiveness of machine learning techniques for test case generation has been studied, and it has been shown that machine learning provides significantly better SUT coverage than random testing. This work concurs with results from our own previous investigation in LBT effectiveness [87].

7.2 Fault injection related

Hardware-based Approaches

HFI techniques require an external device to force/insert faulty values into physical components of the target system. This can be done with or without physical contact to the desired locations. For instance, pin level signals can be corrupted using probes and sockets (e.g. in Messaline [16], RIFLE [79] and FIST [57] tools). However, their drawbacks are limited reachability and high intrusiveness [69]. Contactless methods, like the ones implemented in MARS [72] and FIST tools, employ radiation flux or electromagnetic interference (EMF) to introduce single event upsets (SEUs) in VLSI circuits. Despite their good reachability, the latter tools have no direct control over the exact time and location where the induced faults occur. Other HFI methods can affect internal IC nodes by various physical means like disturbances in power supply, clock, temperature, light, etc. [95].

Simulation-based Approaches

Fault simulation can be performed at various abstraction levels including electronic-circuit, gate, RTL and even instruction-set levels. These techniques pose no damage risk to the system in use and are more affordable regarding time and effort than HFI methods. They also provide higher controllability and observability features to conduct FI tests.

Most common practiced SFIs work in the gate (logic) level by injecting faults into HDL models of SUTs. Such models are modified in two ways. Either using the so called *saboteurs* and *mutants* as in MEFISTO tool [51], or by manipulating HDL signals using built-in commands of the simulator as in VERIFY [106] tool. Saboteur is a dedicated module embedded into the simulator that targets value and timing properties, whereas mutants are faulty extensions of target components to emulate desired defective behavior. Despite their flexibilities, HDL-based SFIs suffer from

scalability problems due to bottlenecks in simulation speed. Especially in large and complex systems, the performance degrades dramatically, and simulation takes tens of times slower [95].

Other SFI methods can be implemented for instance in RTL level where existing available tools are HEARTLESS [102] and FTIs [45]. Moreover, tools like DEPEND [54] supply C++ libraries that satisfy design and FI testing needs for a wide range of fault-tolerant architectures.

Software-based Approaches

SWIFI techniques are more attractive to developers, especially when it comes to FI in software applications and operating systems (OSs). These methods do not require expensive hardware like HFI ones and provide higher observability and controllability of the SUT behavior under faulty operations. SWIFI tools can be grouped into *compile-time* and *runtime* mechanisms. The former, e.g., DOCTOR [58] and PROPANE [63], work by mutating source code or a data image of the program and therefore, are limited to memory fault models. The latter, however, are more widely used as in Xception [6], FERRARI [71], SAFE [38] and MAFALDA [100] tools. Their common feature is to use advanced debugging and monitoring facilities available for most processors to carry out FI campaigns. Finally, Enforcer [90] is an efficient comprehensive FI method for unit testing that uses both approaches and iterative abstraction refinement to identify potential failures of an application.

Model-implemented Approaches

In principle, MIFI techniques define FI mechanisms that are instantiated as model-blocks to be inserted into hardware, software, and system models. This is possible in two ways, either by mutating the original model or by replacing it with a faulty one. FISCADE[90] [116] is an MIFI plug-in to the well-known model-based development environment SCADE [10] which is used mostly in safety-critical applications. Another tool is MODIFI [113] which is dedicated to FI in Simulink/Matlab behavioral models. Its key advantage over other similar tools has been the capability to generate minimal cut sets as a method to mitigate the combinatorial explosion problem. In this work, simple faults such as bit-flip and stuck-at faults are extended to be able to simulate more advanced timing, control, and data flow error, thus, to study the fault propagation properties in the models.

Virtualization-based Approaches

VFI is a branch of SFI where instruction set simulators (ISS) and virtualization platforms are used for FI. This includes software emulation with dynamic binary translation (DBT) to simulate the response of micro-architecture systems. VFI is well-suited for early development validation without actual physical system still in place.

UMLinux [107] is one of the early works to link system virtualization and FI testing as a toolchain. This approach can examine dependability of networked VMs in Linux. Another distinguished work concerns the FAUmachine [98], offering more generic solution including other operating systems. The FAUmachine provides fast simulation speed even for complex embedded systems, and the user can determine fault specifications via VHDL scripts to automate FI testing. The tool also supports permanent, transient and intermittent faults in locations such as memory, network, disk and other peripherals. There are weaknesses in this approach, for example, the lack of fault models for CPU registers and limited availability of processor family models.

In [17], a checkpoint mechanism for VFI is introduced. Checkpoint-based FI (CFI) uses checkpointing to restore previously visited states of an application software in a virtual machine. This is an operating-system level approach for User-Mode Linux to inject faults and analyze the outcome. In recent years, a few FI works have been proposed based on the LLVM (Low Level Virtual Machine) [76]. This special compiler framework has additional support for analyzing a program transformation during various compile and build steps [95]. LLVM symbolizes a program code using IR (intermediate representation) for its optimizer and manages all mid-level transformation events. Hence, the tool and its derived FI techniques are independent of the source code language and the target machine. For example, in LLFI [114] and KULFI [105] faults are injected into an intermediate representation of the application source code. Thus, the subsequent errors can be traced back to program lines more conveniently. However, hardware faults are not supported as these have no noticeable effect at the application layer (e.g., registers and control flow faults).

Fault isolation among several VMs has been studied in [118]. This technique is based on the Xen virtual machine monitor (VMM) that can inject faults into kernel, memory, or registers of so-called para-virtualized (PV) and fully-virtualized (FV) VMs. One advantage of this approach is that non-virtualized systems can be characterized under the influence of faults. However, fidelity problems may arise due to the overhead of logging mechanisms.

Among commercial products, Relyzer [60] and CriticalFault [120] are Simics-based FI tools. To reduce FI experiments, the former uses equivalence-based pruning and the latter vulnerability analysis. With these static and dynamic techniques, application instructions are grouped into various equivalence classes including (among others) control, data and address instructions. Then soft errors are injected into different points to profile faulty behavior or detect SDCs (silent data corruptions).

QEMU-based Fault Injection

There is considerable prior work on QEMU-based FI frameworks in the literature. QEFI [4] is one of the first frameworks designed to examine system and kernel level robustness in the presence of faults. This work instruments TCG and integrates the

GDB to introduce errors at injection points. The injection locations include CPU, RAM, USB and some other external devices. In addition, faults can be triggered based on a user defined probability model, breakpoints or device activity measures. However, the fault model is only a combination of soft errors (e.g., bit-flips) and disturbances in packet communications. Moreover, QEFI is specifically designed for the Linux kernel and does not provide any support for x86 architectures. Another soft error injection approach based on QEMU is presented in [52] with the aim to evaluate the susceptibility of complex systems to certain fault types. However, this approach is limited to x86 architectures, and only bit-wise fault models on GPRs (general purpose registers) are considered. A more flexible approach is F-SEFI [55] which targets HPC (high performance computing) applications that are susceptible to silent data corruption (SDC). F-SEFI provides a broadly tunable injection granularity from the entire application space to specific functions. One shortcoming of the approach is that it only partially covers fault locations, limited to processor model variables, by tainting register values during operand execution.

A number of works have focused on mutation testing for embedded systems. XEMU [19] is an extension of QEMU integrated into its DBT with the ability to mutate disassembled binary code of an ARM instruction set at run-time. Hence, the approach does not assume access to the source code, and it can capture anomalies in the compiled target ISAs. Nevertheless, XEMU is mainly concerned with software faults, and its scope is not hardware fault injection.

Ping Xu et.al [119, 77] proposed BitVaSim with the initial aim of evaluating built in test (BIT) mechanisms in ARM and PowerPC platforms. The tool covers a variety of hardware faults not only in memory but also in I/O and CPU variables. Thus, it allows a good degree of user-mode fault configuration in any process thanks also to the XML technology. Furthermore, BitVaSim provides efficient feedback and monitoring features to control FI experiments and VM environment. One issue with this method is that it is not entirely deterministic. Hence, the reproducibility of an FI campaign is one concern.

Important work on FI by targeting the QEMU-DBT process is presented by Davide Ferraretto's team in [48, 56, 47]. In their approach, CPU GPRs and IRs are subjected to permanent, transient and intermittent faults with equivalent representativeness to RTL models. However, no memory or hardware peripheral faults are presented, and only x86 and ARM architectures are supported.

FIES, the framework described in [66], is the basis for applications in [65] and [64], where the fault models go beyond the low-level elementary bit flips and stuck at faults. Although the QEMU instrumentation remains within the DBT, more advanced memory and processor fault models are introduced to comply with IEC 61508 standard. This allows system level analysis and reliability aware software development and countermeasures for embedded and safety-critical domains.

Chapter 8

Summary of Papers

Paper I:

Hojat Khosrowjerdi, Karl Meinke, and Andreas Rasmusson. *"Learning-Based Testing for Safety Critical Automotive Applications."* In International Symposium on Model-Based Safety and Assessment, pp. 197-211. Springer, Cham, 2017.

Summary of the Paper

The paper discusses the usefulness of learning-based testing for two applications from the automotive domain. The systems under test were black-box systems, and a state machine model was learned for these. Four relevant accuracy questions were studied where the focus was on requirements modeling, learning efficiency and error discovery capabilities of the approach. It is notable that five faults in commercial SUTs were discovered with this methodology.

Author's Personal Contributions

- Formalizing natural language specifications into formal PLTL requirements formulas.
- The implementation of a test harnesses for the system under test and performing the experimental evaluation of the approach on the case study.
- Writing experimental sections of the paper and general discussions and comments on its improvement.

Paper II:

Hojat Khosrowjerdi, Karl Meinke, and Andreas Rasmusson. *"Virtualized-Fault Injection Testing: a Machine Learning Approach."* In Software Testing, Verifica-

tion and Validation (ICST), 2018 IEEE International Conference on, pp. 297-308. IEEE, 2018.

Summary of the Paper

The paper presents a proposal, implemented in a toolchain, for virtualized fault injection testing of safety-critical embedded systems. This approach tries to fully automate the processes of test case generation, fault injection, and verdict construction. It is based on machine learning to reverse engineer models of the system under test, and it adopts a model checking approach for generating the test verdicts. The suggested method combines the hardware simulator QEMU and the interactive debugger GDB to systematically induce predefined state changes and/or events that simulate hardware/software faults in the middle of binary program execution. To demonstrate the effectiveness, the authors report two studies of the application of the toolchain to two real-world automotive safety-related programs. However, the proposal can be extended to different embedded domains.

Author's Personal Contributions

- Architecture proposal, design and the integration of the fault injection toolchain.
- Background study and literature survey along with comparing the method with the most relevant approaches.
- The implementation of a test harnesses for the system under test and performing the experimental evaluation of the approach on the case study.
- Formalizing natural-language specifications into formal PLTL requirements formulas.
- Writing most sections of the paper. However, a few sections are written by Karl Meinke or edited based on Andreas Rasmusson's input.

Chapter 9

Conclusion and Future Work

9.1 Summary

This thesis overview has provided a background and literature survey on the relevant aspects of model-based requirements testing, machine learning, hardware emulation and fault injection. We have considered applying the ideas of LBT, mainly, in the automotive domain of embedded systems. For example, we have shown, using LBT, the benefits of MBT can be achieved without costly maintenance of SUT models, along with generating large black-box test suits on the fly. Also, as a side effect of machine-learning and model-checking (the oracle), the inferred behavioral models are guaranteed to conform with the SUT response function. Hence, a major source of false positives and negatives in MBT can be discarded.

One motivation for this licentiate research is the emergence of complex ADAS ECU applications in parallel with new standards for ECU safety and reliability such as ISO 26262. These developments suggest that automated testing will be a necessary complement to manual testing, even if automation does not fully replace manual methods. Another concern that is addressed is the need for a systematic fault injection of safety components at various development stages. The goal was to evaluate learning-based testing (LBT) as a solution to these problems.

One important aspect of LBT is the full adoption of simple propositional linear temporal logic (PLTL) requirements, e.g., to express low latency behavior of safety-related ECU applications. This approach is already proven in paper II to be a viable modeling method for many informal specifications of embedded systems (automotive). Furthermore, the presented case studies show that the rate of error discovery and convergence of accurately learning complex SUTs for LBT is quite high. With respect to the four objective measures defined in paper II to evaluate the utility of LBT, the following criteria has been assessed:

- Requirements formalization in PLTL is often a straightforward task for an ECU embedded software, since most behavioral specifications follow the sim-

ple invariant trigger-response pattern. However, there are exception cases where refactoring is necessary.

- Refactoring can be used to reduce the complexity of a requirement, resolve ambiguities and eliminate unnecessary details of informal texts. This is a quite efficient approach especially when combined with the capabilities of the SUT wrapper to hide complex computations.

- LBT is s scalable testing method based on active automata learning techniques, which is suitable for small and large embedded systems with hundred thousands of states and millions of transitions. The granularity of inferred models is adjustable by tuning input/output sampling/partitioning parameters in a configuration file.

- LBT is a successful black-box testing method at finding discrepancies between a requirement and its corresponding SUT. Its success rate depends on the incomplete or ambiguous requirements which can result in false verdicts or warnings.

Nevertheless, our case studies show that SUT latency issues need to be more carefully addressed. To this end, we have begun to investigate parallelised testing on multicore platforms. Initial efforts here have successfully increased test throughput by executing multiple copies of an SUT in parallel.

In this thesis, a new methodology for automating hardware fault injection testing of embedded systems have been also presented. This approach is based on the hypothesis to utilize the features of hardware virtualization and LBT to approach an integrated toolchain for virtualized hardware in the loop and FI called LBTest-QEMU-GDB. The main characteristics of such a toolchain are summarized as follows:

- It is non-intrusive because it requires zero to negligible modification to the SUT, the emulation platform and models in QEMU;

- Conventional fault models such as stuck-at, transient and coupling faults are supported at locations accessible to GDB and QEMU (e.g., registers, memory, I/Os, etc.).

- We use LBT technology to automate test case generation, FI, and the Oracle (verdict). This includes PLTL requirements modeling, machine learning, and model-checking techniques;

- We adopt traditional debugging technology to emulate faults in QEMU.

The successful application of this toolchain on two safety-critical industrial cases studies of ECU application is presented in the paper I. This work identified previously unknown anomalies concerning the tested safety requirements.

9.2 Future Work

The current technology for LBT can lead to complex models containing hundreds of thousands of states and millions of transitions. This size and complexity of the learned model has already reached the limits of many available model-checkers. Especially, NuSMV [32], which is currently used in LBTest, has difficulties to handle learned models of a distributed system-of-systems for a platooning scenario [85]. The problem is mainly around the approach (BDD representation) that such model-checkers take to construct a very large structure with all possible paths included for a given flat model and a requirement formula. Hence, model-checking of these systems can either lead to a full stack exception concerning the constraints on the available memory or will take an unreasonable time to be resolved.

To tackle this problem, we envisage to investigate the use of the SMT-based model-checking techniques such as the new NuXMV symbolic model-checker. Other longer-term future research areas include for example multi-core platforms both for hardware emulation and FI testing at integration and system levels. This is also useful for parallelized testing of more complicated case studies with high latency in order to improve the performance and scalability of the LBT technique.

Bibliography

[1] GDB Documentation. URL http://www.gnu.org/software/gdb/documentation/.

[2] Internals/Breakpoint Handling - GDB Wiki. URL https://sourceware.org/gdb/wiki/Internals/Breakpoint%20Handling.

[3] Mbtech. URL www.mbtech-group.com.

[4] QEMU-BASED FAULT INJECTION FRAMEWORK. URL https://www.researchgate.net/publication/260260503_QEMU-BASED_FAULT_INJECTION_FRAMEWORK.

[5] IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, pages 1–84, Dec 1990.

[6] *Xception: Software Fault Injection and Monitoring in Processor Functional Units*. 1995.

[7] Functional safety of electrical/electronic/programmable electronic safety-related systems. Technical Report IEC 61508, The International Electrotechnical Commission, 3, rue de Varembé, Case postale 131, CH-1211 Genève 20, Switzerland, 2010.

[8] Honda recalls 2.5 million vehicles on software issue. *Reuters*, August 2011. URL https://www.reuters.com/article/us-honda-recall/honda-recalls-1-5-million-vehicles-in-united-states-idUSTRE77432120110805.

[9] Supporting processes, baseline 17. ISO 26262:(2011)–Part-8 Road vehicles-functional safety, International Organization for Standardization, 2011. URL https://www.iso.org/obp/ui/#iso:std:iso:26262:-8:ed-1:v1:en.

[10] SCADE Suite®, June 2012. URL http://www.esterel-technologies.com/products/scade-suite/.

[11] Rajeev Alur. *Techniques for Automatic Verification of Real-time Systems*. PhD thesis, Stanford, CA, USA, 1992. UMI Order No. GAX92-06729.

[12] Rajeev Alur, Costas Courcoubetis, Thomas A. Henzinger, and Pei-Hsin Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, pages 209–229, 1992. URL https://doi.org/10.1007/3-540-57318-6_30.

[13] Paul Ammann and Jeff Offutt. *Introduction to software testing.* Cambridge University Press, 2nd Edition, 2016.

[14] Scott Andrews. Vehicle-to-vehicle (v2v) and vehicle-to-infrastructure (v2i) communications and cooperative driving. In Azim Eskandarian, editor, *Handbook of Intelligent Vehicles*, pages 1121–1144. Springer London. ISBN 978-0-85729-085-4. URL https://doi.org/10.1007/978-0-85729-085-4_46. DOI: 10.1007/978-0-85729-085-4_46.

[15] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.

[16] J. Arlat, Y. Crouzet, and J. C. Laprie. Fault injection for dependability validation of fault-tolerant computing systems. In *[1989] The Nineteenth International Symposium on Fault-Tolerant Computing. Digest of Papers*, pages 348–355, June 1989.

[17] Cyrille Artho, Kuniyasu Suzaki, Masami Hagiya, Watcharin Leungwattanakit, Richard Potter, Eric Platon, Yoshinori Tanabe, Franz Weitl, and Mitsuharu Yamamoto. Using checkpointing and virtualization for fault injection. *IJNC*, 5(2):347–372, 2015. URL http://www.ijnc.org/index.php/ijnc/article/view/112.

[18] Ricardo Barbosa, Nuno Silva, and João Mário Cunha. csxception®: First steps to provide fault injection for the development of safe systems in automotive industry. In *Dependable Computing - 14th European Workshop, EWDC 2013, Coimbra, Portugal, May 15-16, 2013. Proceedings*, pages 202–205, 2013. URL https://doi.org/10.1007/978-3-642-38789-0_21.

[19] Markus Becker, Daniel Baldin, Christoph Kuznik, Mabel Mary Joy, Tao Xie, and Wolfgang Mueller. XEMU: An Efficient QEMU Based Binary Mutation Testing Framework for Embedded Software. In *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT '12, pages 33–42, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1425-1. URL http://doi.acm.org/10.1145/2380356.2380368.

[20] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference*, pages 41–46, 2005.

[21] Amel Bennaceur, Reiner Hähnle, and Karl Meinke, editors. *Machine Learning for Dynamic Software Analysis: Potentials and Limits - International Dagstuhl Seminar 16172, Dagstuhl Castle, Germany, April 24-27,*

2016, Revised Papers, volume 11026 of *Lecture Notes in Computer Science*. Springer, 2018. ISBN 978-3-319-96561-1. URL https://doi.org/10.1007/978-3-319-96562-8.

[22] Alfredo Benso and Paolo Prinetto. *Fault injection techniques and tools for embedded systems reliability evaluation*, volume 23. Springer Science & Business Media, 2003.

[23] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*, pages 193–207, 1999. URL https://doi.org/10.1007/3-540-49059-0_14.

[24] A. Binu and G. Santhosh Kumar. Virtualization techniques: A methodical review of XEN and KVM. In *Advances in Computing and Communications - First International Conference, ACC 2011, Kochi, India, July 22-24, 2011. Proceedings, Part I*, pages 399–410, 2011. URL https://doi.org/10.1007/978-3-642-22709-7_40.

[25] P. Bourque, F. Robert, J. M. Lavoie, A. Lee, S. Trudel, and T. C. Lethbridge. Guide to the software engineering body of knowledge (swebok) and the software engineering education knowledge (seek) - a preliminary mapping. In *10th International Workshop on Software Technology and Engineering Practice*, pages 8–23, Oct 2002.

[26] Robert K. Brayton, Gary D. Hachtel, Alberto L. Sangiovanni-Vincentelli, Fabio Somenzi, Adnan Aziz, Szu-Tsung Cheng, Stephen A. Edwards, Sunil P. Khatri, Yuji Kukimoto, Abelardo Pardo, Shaz Qadeer, Rajeev K. Ranjan, Shaker Sarwary, Thomas R. Shiple, Gitanjali Swamy, and Tiziano Villa. VIS: A system for verification and synthesis. In *Computer Aided Verification, 8th International Conference, CAV '96, New Brunswick, NJ, USA, July 31 - August 3, 1996, Proceedings*, pages 428–432, 1996. URL https://doi.org/10.1007/3-540-61474-5_95.

[27] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner. *Model-Based Testing of Reactive Systems: Advanced Lectures (Lecture Notes in Computer Science)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005. ISBN 3540262784.

[28] J. Carreira, H. Madeira, and J. G. Silva. Xception: a technique for the experimental evaluation of dependability in modern computers. *IEEE Transactions on Software Engineering*, 24(2):125–136, Feb 1998. ISSN 0098-5589.

[29] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuxmv symbolic model checker. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 334–342, 2014. URL https://doi.org/10.1007/978-3-319-08867-9_22.

[30] Robert Charette. Jaguar Software Issue May Cause Cruise Control to Stay On, October 2011. URL https://spectrum.ieee.org/riskfactor/transportation/advanced-cars/jaguar-software-issue-may-cause-cruise-control-to-stay-on.

[31] Robert N. Charette. This Car Runs on Code, February 2009. URL http://spectrum.ieee.org/transportation/systems/this-car-runs-on-code.

[32] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. *NuSMV 2: An OpenSource Tool for Symbolic Model Checking*, pages 359–364. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002. ISBN 978-3-540-45657-5. URL http://dx.doi.org/10.1007/3-540-45657-0_29.

[33] J. A. Clark and D. K. Pradhan. Fault injection: a method for validating computer-system dependability. *Computer*, 28(6):47–56, June 1995. ISSN 0018-9162.

[34] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logics of Programs, Workshop, Yorktown Heights, New York, May 1981*, pages 52–71, 1981. URL https://doi.org/10.1007/BFb0025774.

[35] Edmund M. Clarke, Orna Grumberg, and Kiyoharu Hamaguchi. Another look at LTL model checking. *Formal Methods in System Design*, 10(1):47–71, 1997. URL https://doi.org/10.1023/A:1008615614281.

[36] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, MA, USA, January 1999. ISBN 0262032708. URL http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0262032708.

[37] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking*. Springer, 2018. ISBN 978-3-319-10574-1. URL https://doi.org/10.1007/978-3-319-10575-8.

[38] D. Cotroneo and R. Natella. Fault Injection for Software Certification. *IEEE Security Privacy*, 11(4):38–45, July 2013. ISSN 1540-7993.

[39] Jean-Michel Couvreur. On-the-fly verification of linear temporal logic. In *FM'99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 20-24, 1999, Proceedings, Volume I*, pages 253–271, 1999. URL https://doi.org/10.1007/3-540-48119-2_16.

[40] Colin De la Higuera. *Grammatical inference: learning automata and grammars*. Cambridge University Press, 2010.

[41] Alexandre Duret-Lutz and Denis Poitrenaud. SPOT: an extensible model checking library using transition-based generalized büchi automata. In *12th International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2004), 4-8 October 2004, Volendam, The Netherlands*, pages 76–83, 2004. URL https://doi.org/10.1109/MASCOT.2004.1348184.

[42] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Software Engineering, 1999. Proceedings of the 1999 International Conference on*, pages 411–420, May 1999.

[43] E Allen Emerson. Temporal and modal logic. *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, 995(1072):5, 1990.

[44] E. Allen Emerson and Joseph Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. *J. Comput. Syst. Sci.*, 30 (1):1–24, 1985. URL https://doi.org/10.1016/0022-0000(85)90001-7.

[45] Luis Entrena, Celia López, and Emilio Olías. Automatic generation of fault tolerant vhdl designs in rtl. In *FDL (Forum on Design Languages)*, 2001.

[46] Lei Feng, Simon Lundmark, Karl Meinke, Fei Niu, Muddassar A. Sindhu, and Peter Y. H. Wong. *Case Studies in Learning-Based Testing*, pages 164–179. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-41707-8. URL http://dx.doi.org/10.1007/978-3-642-41707-8_11.

[47] D. Ferraretto and G. Pravadelli. Efficient fault injection in QEMU. In *2015 16th Latin-American Test Symposium (LATS)*, pages 1–6, March 2015.

[48] Davide Ferraretto and Graziano Pravadelli. Simulation-based Fault Injection with QEMU for Speeding-up Dependability Analysis of Embedded Software. *Journal of Electronic Testing*, 32(1):43–57, February 2016. ISSN 0923-8174, 1573-0727. URL https://link.springer.com/article/10.1007/s10836-015-5555-z.

[49] P. Filipovikj, M. Nyberg, and G. Rodriguez-Navas. Reassessing the pattern-based approach for formalizing requirements in the automotive domain. In

2014 IEEE 22nd International Requirements Engineering Conference (RE), pages 444–450, Aug 2014.

[50] Michael Fisher. *An Introduction to Practical Formal Methods Using Temporal Logic*. Wiley Publishing, 1st edition, 2011. ISBN 0470027886, 9780470027882. URL http://cds.cern.ch/record/1518490.

[51] P. Folkesson, S. Svensson, and J. Karlsson. A comparison of simulation based and scan chain implemented fault injection. In *Digest of Papers. Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing (Cat. No.98CB36224)*, pages 284–293, June 1998.

[52] F. de Aguiar Geissler, F. L. Kastensmidt, and J. E. P. Souza. Soft error injection methodology based on QEMU software platform. In *2014 15th Latin American Test Workshop - LATW*, pages 1–5, March 2014.

[53] E Mark Gold *et al.* Language identification in the limit. *Information and control*, 10(5):447–474, 1967.

[54] K. K. Goswami. DEPEND: a simulation-based environment for system level dependability analysis. *IEEE Transactions on Computers*, 46(1):60–74, January 1997. ISSN 0018-9340.

[55] Q. Guan, N. Debardeleben, S. Blanchard, and S. Fu. F-SEFI: A Fine-Grained Soft Error Fault Injection Tool for Profiling Application Vulnerability. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 1245–1254, May 2014.

[56] G. Di Guglielmo, D. Ferraretto, F. Fummi, and G. Pravadelli. Efficient fault simulation through dynamic binary translation for dependability analysis of embedded software. In *2013 18th IEEE European Test Symposium (ETS)*, pages 1–6, May 2013.

[57] U. Gunneflo, J. Karlsson, and J. Torin. Evaluation of error detection schemes using fault injection by heavy-ion radiation. In *[1989] The Nineteenth International Symposium on Fault-Tolerant Computing. Digest of Papers*, pages 340–347, June 1989.

[58] Seungjae Han, K. G. Shin, and H. A. Rosenberg. DOCTOR: an integrated software fault injection environment for distributed real-time systems. In *Proceedings of 1995 IEEE International Computer Performance and Dependability Symposium*, pages 204–213, April 1995.

[59] David Harel. *Come, let's play - scenario-based programming using LSCs and the play-engine*. Springer, 2003. ISBN 978-3-540-00787-6. URL http://www.springer.com/computer/programming/book/978-3-540-00787-6.

[60] S. K. Sastry Hari, S. V. Adve, H. Naeimi, and P. Ramachandran. Relyzer: Application resiliency analyzer for transient faults. *IEEE Micro*, 33(3):58–66, May 2013. ISSN 0272-1732.

[61] Robert M. Hierons, Kirill Bogdanov, Jonathan P. Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, Gerald Lüttgen, Anthony J. H. Simons, Sergiy Vilkomir, Martin R. Woodward, and Hussein Zedan. Using formal specifications to support testing. *ACM Comput. Surv.*, 41(2):9:1–9:76, February 2009. ISSN 0360-0300. URL http://doi.acm.org/10.1145/1459352.1459354.

[62] Martin Hiller. *A software profiling methodology for design and assessment of dependable software*. Chalmers University of Technology, 2002.

[63] Martin Hiller, Arshad Jhumka, and Neeraj Suri. PROPANE: An Environment for Examining the Propagation of Errors in Software. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, pages 81–85, New York, NY, USA, 2002. ACM. ISBN 978-1-58113-562-6. URL http://doi.acm.org/10.1145/566172.566184.

[64] Andrea Höller, Armin Krieg, Tobias Rauter, Johannes Iber, and Christian Kreiner. Qemu-based fault injection for a system-level analysis of software countermeasures against fault attacks. In *2015 Euromicro Conference on Digital System Design, DSD 2015, Madeira, Portugal, August 26-28, 2015*, pages 530–533, 2015. URL https://doi.org/10.1109/DSD.2015.79.

[65] Andrea Höller, Georg Macher, Tobias Rauter, Johannes Iber, and Christian Kreiner. A virtual fault injection framework for reliability-aware software development. In *IEEE International Conference on Dependable Systems and Networks Workshops, DSN Workshops 2015, Rio de Janeiro, Brazil, June 22-25, 2015*, pages 69–74, 2015. URL https://doi.org/10.1109/DSN-W.2015.16.

[66] Andrea Höller, Gerhard Schonfelder, Nermin Kajtazovic, Tobias Rauter, and Christian Kreiner. FIES: A fault injection framework for the evaluation of self-tests for cots-based safety-critical systems. In *15th International Microprocessor Test and Verification Workshop, MTV 2014, Austin, TX, USA, December 15-16, 2014*, pages 105–110, 2014. URL https://doi.org/10.1109/MTV.2014.27.

[67] Gerard J. Holzmann. The model checker SPIN. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997. URL https://doi.org/10.1109/32.588521.

[68] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation - international edition (2. ed)*. Addison-Wesley, 2003. ISBN 978-0-321-21029-6.

[69] Mei-Chen Hsueh, T. K. Tsai, and R. K. Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, April 1997. ISSN 0018-9162.

[70] E. Jeong, N. Lee, J. Kim, D. Kang, and S. Ha. FIFA: A Kernel-Level Fault Injection Framework for ARM-Based Embedded Linux System. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 23–34, March 2017.

[71] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. FERRARI: a flexible software-based fault and error injection system. *IEEE Transactions on Computers*, 44(2):248–260, February 1995. ISSN 0018-9340.

[72] Johan Karlsson and Peter Folkesson. Application of three physical fault injection techniques to the experimental assessment of the MARS architecture. pages 267–287. IEEE Computer Society Press, 1995.

[73] Michael J Kearns and Umesh Virkumar Vazirani. *An introduction to computational learning theory*. MIT press, 1994.

[74] Hojat Khosrowjerdi, Karl Meinke, and Andreas Rasmusson. Learning-Based Testing for Safety Critical Automotive Applications. In *Model-Based Safety and Assessment*, Lecture Notes in Computer Science, pages 197–211. Springer, Cham, September 2017. ISBN 978-3-319-64118-8 978-3-319-64119-5. URL https://link.springer.com/chapter/10.1007/978-3-319-64119-5_13.

[75] Sascha Konrad and Betty H. C. Cheng. Real-time specification patterns. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 372–381, New York, NY, USA, 2005. ACM. ISBN 1-58113-963-2. URL http://doi.acm.org/10.1145/1062455.1062526.

[76] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2102-9. URL http://dl.acm.org/citation.cfm?id=977395.977673.

[77] Yi Li, Ping Xu, and Han Wan. A Fault Injection System Based on QEMU Simulator and Designed for BIT Software Testing. *Applied Mechanics and Materials*, 347-350:580–587, 2013. ISSN 1662-7482. URL https://www.scientific.net/AMM.347-350.580.

[78] Orna Lichtenstein and Amir Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages, New Orleans, Louisiana, USA, January 1985*, pages 97–107, 1985. URL http://doi.acm.org/10.1145/318593.318622.

[79] Henrique Madeira, Mário Rela, Francisco Moreira, and João Gabriel Silva. *RIFLE: A general purpose pin-level fault injector*, pages 197–216. Springer Berlin Heidelberg, Berlin, Heidelberg, 1994. ISBN 978-3-540-48785-2. URL https://doi.org/10.1007/3-540-58426-9_132.

[80] Kenneth L. McMillan. *Symbolic model checking*. Kluwer, 1993. ISBN 978-0-7923-9380-1.

[81] K. Meinke and M. A. Sindhu. Incremental learning-based testing for reactive systems. In Martin Gogolla and Burkhart Wolff, editors, *Tests and Proofs: 5th International Conference, TAP 2011, Proceedings*, pages 134–151. Springer, 2011.

[82] K. Meinke and J. V. Tucker. Handbook of logic in computer science (vol. 1). chapter Universal Algebra, pages 189–368. Oxford University Press, Inc., New York, NY, USA, 1992. ISBN 0-19-853735-2. URL http://dl.acm.org/citation.cfm?id=162573.162534.

[83] Karl Meinke. Cge: A sequential learning algorithm for mealy automata. In *International Colloquium on Grammatical Inference*, pages 148–162. Springer, 2010.

[84] Karl Meinke. Learning-Based Testing of Cyber-Physical Systems-of-Systems: A Platooning Study. In *Computer Performance Engineering*, Lecture Notes in Computer Science, pages 135–151. Springer, Cham, September 2017. ISBN 978-3-319-66582-5 978-3-319-66583-2. URL https://link.springer.com/chapter/10.1007/978-3-319-66583-2_9.

[85] Karl Meinke. Learning-based testing of cyber-physical systems-of-systems: A platooning study. In *Computer Performance Engineering - 14th European Workshop, EPEW 2017, Berlin, Germany, September 7-8, 2017, Proceedings*, pages 135–151, 2017.

[86] Karl Meinke, F. Niu, and M. Sindhu. *Learning-Based Software Testing: A Tutorial*, pages 200–219. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-34781-8. URL http://dx.doi.org/10.1007/978-3-642-34781-8_16.

[87] Karl Meinke and Fei Niu. A learning-based approach to unit testing of numerical software. In *Testing Software and Systems - 22nd IFIP WG 6.1 International Conference, ICTSS 2010, Natal, Brazil, November 8-10, 2010. Proceedings*, pages 221–235, 2010.

[88] Karl Meinke and Peter Nycander. *Learning-Based Testing of Distributed Microservice Architectures: Correctness and Fault Injection*, pages 3–10. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015. ISBN 978-3-662-49224-6. URL http://dx.doi.org/10.1007/978-3-662-49224-6_1.

[89] Karl Meinke and Muddassar A. Sindhu. Lbtest: A learning-based testing tool for reactive systems. In *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, ICST '13, pages 447–454, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-0-7695-4968-2. URL http://dx.doi.org/10.1109/ICST.2013.62.

[90] Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors. *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings*, volume 4085 of *Lecture Notes in Computer Science*. Springer, 2006. ISBN 3-540-37215-6. URL https://doi.org/10.1007/11813040.

[91] Srinivas Nidhra and Jagruthi Dondeti. Blackbox and whitebox testing techniques-a literature review. *International Journal of Embedded Systems and Applications (IJESA)*, 2(2):29–50, 2012.

[92] Ümit Özgüner, Tankut Acarman, and Keith Alan Redmill. *Autonomous ground vehicles*. Artech House, 2011.

[93] Rajesh G Parekh, Codrin Nichitiu, and Vasant Honavar. A polynomial time incremental algorithm for regular grammar inference. 1997.

[94] Doron A. Peled, Moshe Y. Vardi, and Mihalis Yannakakis. Black box checking. *Journal of Automata, Languages and Combinatorics*, 7(2):225–246, 2002.

[95] Roberta Piscitelli, Shivam Bhasin, and Francesco Regazzoni. Fault attacks, injection techniques and tools for simulation. In *Hardware Security and Trust*, pages 27–47. Springer, 2017. URL http://link.springer.com/chapter/10.1007/978-3-319-44318-8_2.

[96] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57, 1977. URL https://doi.org/10.1109/SFCS.1977.32.

[97] Amalinda Post, Igor Menzel, Jochen Hoenicke, and Andreas Podelski. Automotive behavioral requirements expressed in a specification pattern system: a case study at bosch. *Requirements Engineering*, 17(1):19–33, 2012. ISSN 1432-010X. URL http://dx.doi.org/10.1007/s00766-011-0145-9.

[98] S. Potyra, V. Sieh, and M. Dal Cin. Evaluating fault-tolerant system designs using faumachine. In *Proceedings of the 2007 Workshop on Engineering Fault Tolerant Systems*, EFTS '07, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-725-4. URL http://doi.acm.org/10.1145/1316550.1316559.

[99] Victor Reyes. Virtualized Fault Injection Methods in the Context of the ISO 26262 Standard. *SAE International Journal of Passenger Cars - Electronic*

and Electrical Systems, 5(1):9–16, April 2012. ISSN 1946-4622. URL http://papers.sae.org/2012-01-0001/.

[100] Manuel Rodríguez, Frédéric Salles, Jean-Charles Fabre, and Jean Arlat. MAFALDA: microkernel assessment by fault injection and design aid. In *Dependable Computing - EDCC-3, Third European Dependable Computing Conference, Prague, Czech Republic, September 15-17, 1999, Proceedings*, pages 143–160, 1999. URL https://doi.org/10.1007/3-540-48254-7_11.

[101] Francesca Rossi. Safecop. URL http://www.safecop.eu/.

[102] C. Rousselle, M. Pflanz, A. Behling, T. Mohaupt, and H. T. Vierhaus. A register-transfer-level fault simulator for permanent and transient faults in embedded processors. In *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*, pages 811–, 2001.

[103] Kristin Y. Rozier. Linear temporal logic symbolic model checking. *Computer Science Review*, 5(2):163–203, 2011. URL https://doi.org/10.1016/j.cosrev.2010.06.002.

[104] Muzammil Shahbaz, K. C. Shashidhar, and Robert Eschbach. Specification inference using systematic reverse-engineering methodologies: An automotive industry application. *IEEE Softw.*, 29(6):62–69, November 2012. ISSN 0740-7459. URL http://dx.doi.org/10.1109/MS.2011.159.

[105] V. C. Sharma, A. Haran, Z. Rakamaric, and G. Gopalakrishnan. Towards formal approaches to system resilience. In *2013 IEEE 19th Pacific Rim International Symposium on Dependable Computing*, pages 41–50, Dec 2013.

[106] V. Sieh, O. Tschache, and F. Balbach. VERIFY: evaluation of reliability using VHDL-models with embedded fault descriptions. In *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing*, pages 32–36, June 1997.

[107] Volkmar Sieh and Kerstin Buchacker. *UMLinux - A Versatile SWIFI Tool*, pages 159–171. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002. ISBN 978-3-540-36080-3. URL https://doi.org/10.1007/3-540-36080-8_16.

[108] J. E. Smith and Ravi Nair. The architecture of virtual machines. *Computer*, 38(5):32–38, May 2005. ISSN 0018-9162.

[109] Neil R. Storey. *Safety Critical Computer Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996. ISBN 0201427877.

[110] Ahyoung Sung, Byoungju Choi, W. Eric Wong, and Vidroha Debroy. Mutant generation for embedded systems using kernel-based software and hardware fault simulation. *Information and Software Technology*, 53(10):1153–1164, October 2011. ISSN 0950-5849. URL http://www.sciencedirect.com/science/article/pii/S0950584911000863.

[111] R. Svenningsson, H. Eriksson, J. Vinter, and M. Torngren. Model-Implemented Fault Injection for Hardware Fault Simulation. In *Workshop on Model-Driven and Validation 2010 Engineering, Verification*, pages 31–36, October 2010.

[112] Rickard Svenningsson. Model-Implemented Fault Injection for Robustness Assessment. 2011. URL http://kth.diva-portal.org/smash/record.jsf?pid=diva2:460561.

[113] Rickard Svenningsson, Jonny Vinter, Henrik Eriksson, and Martin Törngren. MODIFI: A model-implemented fault injection tool. In *Computer Safety, Reliability, and Security, 29th International Conference, SAFECOMP 2010, Vienna, Austria, September 14-17, 2010. Proceedings*, pages 210–222, 2010. URL https://doi.org/10.1007/978-3-642-15651-9_16.

[114] A. Thomas and K. Pattabiraman. Error detector placement for soft computation. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12, June 2013.

[115] Moshe Y. Vardi. Automata-theoretic model checking revisited. In *Verification, Model Checking, and Abstract Interpretation, 8th International Conference, VMCAI 2007, Nice, France, January 14-16, 2007, Proceedings*, pages 137–150, 2007. URL https://doi.org/10.1007/978-3-540-69738-1_10.

[116] J. Vinter, L. Bromander, P. Raistrick, and H. Edler. FISCADE - A Fault Injection Tool for SCADE Models. In *2007 3rd Institution of Engineering and Technology Conference on Automotive Electronics*, pages 1–9, June 2007.

[117] Neil Walkinshaw, Kirill Bogdanov, John Derrick, and Javier Paris. *Increasing Functional Coverage by Inductive Testing: A Case Study*, pages 126–141. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-16573-3. URL http://dx.doi.org/10.1007/978-3-642-16573-3_10.

[118] S. Winter, M. Tretter, B. Sattler, and N. Suri. simfi: From single to simultaneous software fault injections. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12, June 2013.

[119] J. Xu and P. Xu. The Research of Memory Fault Simulation and Fault Injection Method for BIT Software Test. In *2012 Second International Conference on Instrumentation, Measurement, Computer, Communication and Control*, pages 718–722, December 2012.

[120] Xin Xu and Man-Lap Li. Understanding soft error propagation using efficient vulnerability-driven fault injection. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pages 1–12, June 2012.

Part II

Appendices and Included Papers

Appendix A

Formal Language

In Section 3.2, a finite input alphabet and the set of all finite-length strings over it were defined as Σ and Σ^*. Now, the term *formal language* (L) refers to a subset of Σ^*. According to *Kleene's Theorem*, a *regular language* is a formal language that is accepted by a DFA. This particular language type has many interesting features including the *closure* of intersection, union and concatenation operators. In general, a regular language can be recursively generated by the following rule:

$$L_{reg} ::= \emptyset | \epsilon | a | L_{reg1} \cup L_{reg2} | L_{reg1} \cap L_{reg2} | L_{reg1}.L_{reg2} | L^*_{reg}$$

where $\emptyset$ denotes the empty language, ϵ represents the language of empty string, $a \in \Sigma^*$ is a singleton and L_{reg1} and L_{reg2} are each a separate regular language. Finally, L^*_{reg} is the *Kleene closure*, i.e., the infinite union of $\cup_{i \geq 0} L^i_{reg}$ where L^i_{reg} is the concatenation of i copies of L_{reg} [68].

Appendix B

Paper I (Learning-Based Testing for Safety Critical Automotive Applications)

Hojat Khosrowjerdi, Karl Meinke, and Andreas Rasmusson. 5th International Symposium on Model-Based Safety and Assessment (IMBSA), 2017, Trento, Italy.

Appendix C

Paper II (Virtualized-Fault Injection Testing: a Machine Learning Approach)

Hojat Khosrowjerdi, Karl Meinke, and Andreas Rasmusson. In Software Testing, Verification and Validation (ICST), 2018, Västerås, Sweden.