DI Hermann Felbinger, BSc

Characterizing Quality Assessment and Redundancy Elimination of Test Suites Without Execution

Doctoral Thesis

submitted to

Graz University of Technology

Institute of Softwaretechnology

Supervisor:

Univ.-Prof. Dipl.-Ing. Dr. techn. Franz Wotawa

Graz, 2020

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present PhD thesis.

Date

Signature

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Dissertation identisch.

Datum

Unterschrift

Abstract

Deciding whether a given test suite is effective enough is certainly a challenging task. To state whether a System Under Test is sufficiently tested requires an assessment of the test suite quality. Existing methods to assess the quality of a test suite either are based on the structure of an implementation or determine the quality by using fault injection, called mutation score. In this thesis we introduce a method, which is based on inductive inference to assess the quality of a test suite and propose methods to augment or reduce a test suite depending on the quality assessment result.
In general the idea to assess the quality of a test suite is to use the similarity or if possible equivalence between a model inferred from a test suite and the system under test as a measure of test suite adequacy, which is the ability of a test suite to expose errors in the system under test. Focusing on a software program's functionality, for assessing test suites of Boolean functions, we use machine learning in order to infer a special binary decision diagram from the considered test suites and extract a total variable order, if possible. Intuitively, if a reduced ordered binary decision diagram derived from the Boolean functions representing the program under test's specification actually coincides with that of the test suite (using the same variable order), we conclude that the test suite is effective enough. That is, any program that passes such a test suite should clearly show the desired input-output behavior. For general purpose systems under test we define similarity by using the root mean squared error computed from the differences of the system under test's output and the model output for certain inputs not used for model inference.

Also removing redundancies from test suites is an important task of software testing in order to keep test suites as small as possible, but not to harm the test suite's fault detection capabilities. A straightforward algorithm for test suite reduction would select elements of the test suite randomly

and remove them if and only if the reduced test suite fulfills the same or similar structural coverage or mutation score. Such algorithms rely on the execution of the program and the repeated computation of coverage or mutation score. In this thesis, we present an alternative approach that purely relies on a model inferred from the original test suite without requiring the execution of the program under test. The idea is to remove those tests that do not change the inferred model. The equivalence relation underlying the comparisons plays obviously a significant role for the effectiveness achieved and efficiency experienced. We explore five such relations that take different aspects into account and investigate their impact on test suite reduction, their effectiveness in fault detection, and computation time. We report corresponding results, and show as well as prove that the equivalence relations build a taxonomy. In order to evaluate the approach we carried out an experimental study showing that reductions of 60-99% are possible while still keeping coverage and mutation score almost the same.

Kurzfassung

Ein Urteil zur Entscheidung zu finden, ob eine gegebene Menge an Testfällen ausreichend wirkungsvoll ist, ist eine anspruchsvolle Aufgabe. Dieses Urteil erfordert eine Bewertung der Qualität dieser Testfälle. Bestehende Methoden zur Bewertung der Qualität von Testfällen basieren entweder auf der Struktur oder Mutationen der Implementierung. Die Methode, die wir in dieser Arbeit vorschlagen, basiert auf induktiver Ableitung um die Qualität von Testfällen zu bewerten, sowie, abhängig von den Bewertungsergebnissen, Methoden zur Erweiterung und Reduktion von Testfällen.
Die Methode zur Bewertung der Qualität von Testfällen basiert auf der Idee, die Ähnlichkeit bzw. wenn möglich die Äquivalenz zwischen einem Modell, abgeleitet von Testfällen, und dem zu testended System als Maß für die Qualtiät anzuwenden. Als Qualität wird hier bezeichnet, wie geeignet die Testfälle insgesamt zum Finden von Fehlern im zu testenden System sind. Mit dem Fokus auf Testen der Funktionalität benutzen wir zum Testen von Bool'schen Funktionen Maschienen Lernen, um ein binäres Entscheidungsdiagramm aus den betrachteten Testfällen abzuleiten und, wenn möglich, eine Anordnung der Variablen. Wenn dann ein reduziertes geordnetes binäres Entscheidungsdiagramm abgeleitet aus den Bool'schen Funktionen, die die zu testenden Systemspezifikationen darstellen, mit jenem abgeleitet aus den Testfällen übereinstimmt (mit der selben Anordnung der Variablen), schließen wir daraus, dass die Testfälle ausreichend sind. Wenn die Testfälle ausreichend sind, zeigt jedes Programm, das alle Testfälle besteht, das gewünschte Verhalten von Eingaben und entsprechenden Ausgaben. Allgemein für zu testende Systeme definieren wir Ähnlichkeit als radizierten mittleren quadratischen Fehler, berechnet aus der Differenz der Ausgaben des zu testenden Systems und jenen des abgeleiteten Models für alle Eingaben, die zum Ableiten eines Models in Betracht gezogen wurden.

Auch das Entfernen von redundanten Testfällen ist eine wichtige Aufgabe

im Bereicht Softwaretesten. Dabei wird die Anzahl der Testfälle durch Entfernen redundanter Testfälle so gering wie möglich gehalten, während die Qualität der Testfälle insgesamt gleich bleibt. Ein simpler Algorithmus zur Reduktion von Testfällen wäre randomisiert Testfälle auszuwählen und zu entfernen solange die übgrigen Testfälle die gleiche bzw. ähnliche strukturelle Abdeckung oder Mutationswert erreichen. Derartige Algorithmen erfordern die mehrmalige Ausführung des zu testenden Systems und die mehrmalige Berechnung der Abdeckung der des Mutationswerts. Hier wird ein alternativer Ansatz beschrieben, der auf der Ableitung von Modellen aus den initialen Testfällen basiert, während das zu testende System nicht ausgeführt werden muss. Dabei werden nur Testfälle entfernt, die das abgeleitete Modell im Vergleich zum Modell, abgeleitet aus den originalen Testfällen, nicht verändern. Die dem Vergleich unterliegende Äquivalenzrelation spielt eine signifikante Rolle für die erreichte Effizienz bei der Reduktion. In dieser Arbeit wurden fünf Äquivalenzrelationen, die verschiedene Aspekte und deren Einflüsse auf die Testfallreduktion und Rechenzeit betrachten, untersucht. Dazu beschreiben wir entsprechende experimentelle Ergebnisse und beweisen, dass diese Äquivalenzrelationen eine Taxonomie bilden. Um den Ansatz zu evaluieren führten wir eine experimentelle Studie durch, die zeigt, dass Reduktionen von 60-99% möglich sind, während die Abdeckung und der Mutationswert nahezu unverändert bleiben.

Acknowledgements

First, I would like to express my sincere gratitude to my advisor Prof. Franz Wotawa for the continuous support of my PhD study and research, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my PhD study. I thank Radu Mateescu for reviewing the thesis, his support and commitment to act as a referee assessing the thesis and as an examiner for the defense of the thesis.
My sincere thanks also goes to Alexandre Petrenko for the ideas and fruitful discussions we had and also the opportunity to visit him and his research group, which was a great experience and a very productive time.

I thank my colleagues and collaborators for the stimulating discussions, for enlightening me the first glance of research, and for all the fun we have had in the last years. I would also like to thank my friends for their seemingly inexhaustible well of support. I thank my brother and sister and their families, and my parents for providing me with an exceptionally stimulating childhood.

Foremost I would like to thank the love of my life Nadja and our incredible children for their dedication, patience and support in every matter.

Contents

List of Figures

Part I.

Introduction

1. Introduction

In the past 35 years software applications have spread out from the original data processing and scientific computing domains into our daily lives, such as phones, cars, kitchen appliances, and services which transferred to the internet. In general software products and products driven by software must meet two challenges. First, software development cost should be low and software should be deployed in a short time to stay competitive. Second, software must adhere quality attributes like usability, dependability, and safety.
McCall et al. [1] gave in their report, a concept of software quality in terms of quality factors and quality criteria:

- **Factor:** a condition or characteristic which actively contributes to the quality of the software. For standardization purposes, all factors will be related to a normalized cost to either perform the activity characterized by the factor or to operate with that degree of quality. For example, maintainability is the effort required to locate and fix an error in an operational program. This effort required may be expressed in units such as time, money, or manpower. The following rules were used to determine the prime set of quality factors:
  - A condition or characteristic which contributes to software quality,
  - a user-related characteristic,
  - related to cost either to perform the activity characterized by the function or to operate with that degree of quality,
  - relative characteristic between software products.
- **Criteria:** Attributes of the software or software production process by which the factors can be judged and defined. The following rules are applied to the determination of criteria:

Figure 1.1.: ISO/IEC 25010 quality model [2].

  - Attributes of the software or software products of the development process; i.e., criteria are software oriented while factors are user oriented,
  - may display a hierarchical relationship with subcriteria,
  - may affect more than one factor.
- **Metrics:** Measures of the criteria or subcriteria related to the quality factors. The measures may be objective or subjective. The units of the metrics are chosen as the ratio of actual occurrences to the possible number of occurrences.

Correctness, reliability, efficiency, testability, maintainability, and reusability are examples of quality factors whereas a quality criterion is an attribute of a quality factor that is related to software development. The ISO/IEC 25000 series of standards provide a quality model with eight independent quality characteristics and sub-characteristics, as shown in Figure 1.1, which are related to the initial software quality concept of McCall et al. [1].

To assess the achieved software quality testing plays an important role. On the one hand, software quality is improved during development by continuously repeating the $\rightarrow$ *test* $\rightarrow$ *find defects* $\rightarrow$ *fix* $\rightarrow$ loop. On the other hand, system level tests have to be executed to assess the software quality before releasing the software. As explained in Osterweil et al. [3] software quality assessment can be divided into two broad categories, namely, static analysis and dynamic testing.

- **Static Analysis**: A systematic examination of program structure for the purpose of showing that certain properties are true. Static analysis

does not require to execute the software under test.
- **Dynamic Testing**: Entails executing the software under test and examining the outcomes. The behavior of the software under test is reflected by the relation of provided inputs and given outcomes after executing the software.

Both, static analysis and dynamic testing are complementary used to identify as many faults as possible. A software failure occurs when the executed software does not perform as required and expected. Therefore, a failure is a deviation of the behavior of an executed program from its intended behavior. A software fault is a malformation whose execution causes a failure, and an error is a flaw in human reasoning and performance that leads to the creation of the fault [3].

If the developed software is a simple web application, a smartphone app, or a safety critical system in the automotive industry, quality assurance in terms of robustness, performance, correct functionality, etc., for these products requires testing. In general, the more critical the developed software product is in terms of safety and security, the more effort is put into quality assurance. This effort is limited by resources which are mainly computational power and time. Therefore, exhaustive testing by providing all possible input variations is in general not feasible in finite time. Structured methods to create test cases tackle this issue. Numerous of these methods already exist and still new methods are developed, especially for new software based and software supported technologies. The most popular methods are stochastic based random input selection, model based test creation, where a reference model of the developed software product supports the test creation.

1.1. Thesis Statement

The focus of this thesis is on improving our understanding and characterization of quality assessment and redundancy elimination of test suites without execution. As mentioned earlier, software testing requires the execution of the software under test. For complex systems a huge number of tests causes impractical execution time. Existing quality metrics, such as code coverage

and mutation score require the source code of the software under test to be available, where also drawbacks such as changes in the behavior due to code instrumentation, or not manageable execution effort due to high number of mutants exist. Another metric, which is also applicable without having the source code of the program under test, but only the binaries is combinatorial coverage. Combinatorial coverage is focusing on the input space of the software under test.

This thesis begins with analyzing combinatorial testing and continues with definitions and empirical investigations of model inference based quality assessment, where the correlation to existing quality assessment methods is analyzed. The test suites were generated automatically with various techniques, such as random, property based, and combinatorial testing. Another approach to assess the quality of test suites, applicable for Boolean functions, is based on reduced ordered binary decision diagrams for which the equivalence problem is decidable. The definition of these methods and the reduced execution time to obtain results allows to adapt these methods also for redundancy elimination.

We thus formulate the following thesis statement:

> *Developing an understanding of software and software quality requires testing. The inevitable part of dynamic testing requires test input definition, execution, and analyzing the result. Specifically, quantifying the confidence that the executed test cases are able to identify all deviations of the software under test to the intended behavior and the elimination of redundancies requires a metric whose calculation is feasible within limited time.*

1.2. Contributions and Outline

In this section we present an overview of the research problems investigated within this thesis. Following the above thesis statement this thesis makes the following contributions:

- In *Part 2* we study combinatorial testing, which is used throughout this thesis. As a first investigation we extend manually created test

cases by combinatorial testing and show how this affects the test suite quality. In this chapter we generalized conventional unit tests into parameterized unit tests for which we generated the input values by using combinatorial testing. For the combinatorial test case generation we used the input domains extracted from the conventional unit tests. The empirical evaluation also provides results showing the combinatorial coverage of the existing, manually created test cases. We also show how adapting the existing unit tests affects the mutation score. This chapter is based on the work "Adapting Unit Tests by Generating Combinatorial Test Data" [4].

- In *Part 3* we introduce a model inference based quality assessment approach. In this chapter we start with giving an overview of existing test suite quality assessment methods. Then we define how to infer a model from a test suite. We show two different approaches how to quantify the quality. First we show for programs with discrete outcomes (number of possible outcomes > 2) and state based systems, where it is possible to monitor the current state, how model inference is related to the actual outcomes. In a second approach we classify the test suite effectiveness for Boolean functions by inferring reduced ordered binary decision diagrams from the Boolean functions and the test suite, and decide on their equivalence whether the actual test suite is effective or not. In the empirical evaluation we show how the model inference approaches relate to existing quality assessment approaches, such as code coverage, mutation score, and combinatorial coverage. This chapter is based on the papers "Test Suite Quality Assessment Using Model Inference Techniques" [5], "Empirical Study of Correlation Between Mutation Score and Model Inference Based Test Suite Adequacy Assessment" [6], "Classifying Test Suite Effectiveness via Model Inference and ROBBDs" [7], and "Mutation Score, Coverage, Model Inference: Quality Assessment for T-Way Combinatorial Test-Suites" [8].
- We introduce in *Part 4* an approach to reduce the size of a test suite based on the previously introduced model inference based quality assessment approach. This reduction approach allows to eliminate redundancies in a test suite without executing the software under test. Also we define various relations how to compare inferred models from a test suite. We show how these relations build a taxonomy. In an

empirical evaluation we compare our approach with a conventional approach by time and effectiveness of the reduction, and show how the defined relations affect the reduction. This chapter is based on the papers "Test-Suite Reduction Does Not Necessarily Require Executing the Program under Test" [9] and "A "strength of decision tree equivalence"-taxonomy and its impact on test suite reduction" [10].

We give an overview of related literature within the respective chapters. Finally, in Chapter 5 we conclude the thesis and give some directions for model inference based quality assessment of test suites without execution.

Part II.

Combinatorial Testing

2. t-way Combinatorial Testing

In the last years one of the main topics of software quality assurance research is automatic test case generation. When choosing an automatic test generation approach, the quality of the generated test suites has to be ensured. Different test generation approaches target different types of fault [11], [12]. One approach to generate test cases is t-way combinatorial testing. This approach has been well studied in the last 20 years [12]. Combinatorial testing is supposed to detect interaction triggered faults. Combinatorial testing includes modeling of the inputs, constraints [13], failure diagnosis, prioritization, and test generation. Additionally combinatorial testing serves to reduce the full Cartesian product space of a set of values from the input parameters, which may be extremely large in real-world applications.

Combinatorial testing was introduced to detect faults triggered by interactions of parameters in the program under test, and is therefore also called Combinatorial Interaction Testing. Combinatorial testing tests a program with covering arrays as test suite. The covering arrays test a subset of the exhaustive set of parameter value combinations. Parameters can be configuration parameters, internal or external events. We assume the program under test has a set of n parameters $P = \{p_1, ..., p_n\}$ and each parameter $p_i \in P$ has a set of discrete values V_i. An exhaustive test suite is the product of $V_1 \times ... \times V_n$. In t-way combinatorial testing we use the binomial coefficient $\binom{n}{t}$, which result is the number of ways to choose t parameters from n parameters, to calculate the number of possible combinations within P. A combination is defined as:

Definition 1 (Combination) *A combination $c \in C$ is a set of parameters, where C contains all possible distinct subsets of P with size t.*

In t-way combinatorial testing the value of t is named strength. E.g. 2-way combinatorial testing has strength 2 and for each of the $\binom{n}{2}$ pairs $(p_i, p_j) \in P$ all cartesian products of $V_i \times V_j$ exist in the generated test suite.

The tool which we used in this work to generate t-way combinatorial tests uses the IPOG algorithm. The IPOG algorithm is an extension of the IPO-algorithm (In Parameter Order) [14]. IPO covers "one-parameter-at-a-time" through horizontal and vertical extension mechanisms. As a t-way strategy IPO has been extended into IPOG [15], because IPO only supports pairwise or 2-way combinatorial test generation. The combinations of the first t parameters are initially generated as a partial test suite, that is based on the values of the parameters in these combinations. The test suite is then extended with the values of the next parameters using horizontal and vertical extension mechanisms. Horizontal extension extends the partial test suite with values of the next parameter to cover the maximum number of interactions. Upon completion of horizontal extension, vertical extension may be summoned to generate additional test cases that cover all of the uncovered interactions. More recently, a number of variants have been developed to improve IPOG performance (IPOG-D [16], IPOF and IPOF2 [17]).

A test case and a test suite in combinatorial testing are defined as:

Definition 2 (Test Case) *A test case is a tuple $(v_1, ..., v_n, o) \in V_1 \times ... \times V_n \times O$, where $V_1, ..., V_n$ are the sets of possible values for the parameters in P and $v_i \in V_i$. O is the set of possible outcomes of the program under test when executing the program with input values from $V_1 \times ... \times V_n$, and $o \in O$.*

Definition 3 (Test Suite) *A test suite T_t is a set of test cases that cover all interactions of strength t.*

In combinatorial testing also constraints over parameters can be used [13], which prevent the generation of certain valuations of combinations. In addition to the resulting combinations, additional interaction relations can be specified. An interaction relation is a set of parameters which requires that all value combinations of these parameters have to be in the test suite, because they can affect the program under test and therefore may trigger a

failure. The interaction relations can be viewed as covering requirements, specifying which combinations should be covered while testing.

2.1. Input Modeling

In combinatorial testing an input model consists of parameters, values, interaction relations and constraints. More precisely: parameters that may affect the program under test and values that should be selected for each parameter, interaction relations that exist between parameters, and constraints that exist between values of the different parameters, which are used to exclude combinations that are not meaningful from the domain semantics. In this thesis we created the input models manually by investigating the source code.

2.2. Oracle for test case generation

Because there exists no oracle in combinatorial testing by default, if we do not have a formal model of the program under test, we had to choose a different strategy to obtain the expected outcomes for the generated test cases in this work. Here we executed the original program with the parameter values of a covering array from $V_1 \times ... \times V_n$ as generated, and added the outcome of the original program as expected outcome to each of the test cases.

3. Adapting Unit Tests by Generating Combinatorial Test Data

This chapter is based on the work "Adapting Unit Tests by Generating Combinatorial Test Data" [4].

Conventional unit tests are still mainly handcrafted. Generalizing conventional unit tests to parameterized unit tests supports automatic test data generation. Parameterized unit tests accept parameters and describe the behavior of the method under test for all test arguments. Methods that were introduced to instantiate parameterized unit tests with concrete values as test data are based on search based approaches, dynamic symbolic execution, or property based testing. In this chapter, we introduce an approach that retrofits existing conventional unit tests into parameterized unit tests by generalization, and generate test data by combinatorial valuation to adapt existing conventional unit test suites.

3.1. Introduction

In software development, projects should have a suite of unit tests. These unit tests often have been collected over time, some of them checking specific and subtle cases. According to [18] writing Parameterized Unit Tests (PUTs) is more challenging than writing CUTs (also named closed traditional unit tests in [18]). Because of this challenge and the fact that PUTs came up in the year 2005 [19] where CUTs already existed, CUTs are mainly used in software development projects.

As introduced in [19] the input data for PUTs can be generated automatically. In their work Tillmann et al. used dynamic symbolic execution to create path constraints from which they derived input values for the variables by constraint satisfaction. In our work we made use of the existing input data within CUTs and generate t-way combinatorial valuations from these input data. Consequently our approach is limited to methods under test with two or more input parameters. We provide an algorithm that is used to manually retrofit CUTs into PUTs. This algorithm extracts the input parameters from the CUT that are necessary to create a PUT. Furthermore the algorithm extracts the existing input data within the CUTs. These input data are used as a basis to generate t-way combinatorial valuations as inputs of the PUTs. With this approach we can adjust an existing test suite, meaning that its size can either decrease or grow while maintaining or even increasing the existing test suite quality. To assess a test suite's quality we calculate its mutation score and code coverage. This approach is very appealing to apply in all variants of software development projects and domains, because with feasible effort you can increase the quality of existing test suites without knowing any details of the program under test. Combinatorial test suites are usually of high quality in terms of fault detection [8, 11, 20–25]. Therefore the developer/tester only has to make sure that the existing input values in the CUTs are valuable e.g. they represent all possible boundary values.

Adapting a test suite can be either extending or reducing it. There exist multiple test suite reduction approaches which are based on coverage, mutation score, or model inference [9, 26–31]. Extending a test suite usually requires more input data which can be added manually, generated randomly, or derived from symbolic execution. In this chapter we adapt the existing test suites without additional input data by generating *t*-way combinatorial valuations from the existing data.

For our empirical study we use four example applications from the Defects4J collection [32]. Defects4J's purpose is to enable controlled testing studies for Java. It is a collection of reproducible bugs and supporting infrastructure for several open source projects. From these example applications we generalized CUTs into PUTs, created input models for the generator of the *t*-way combinatorial valuations, derived the expected outcomes for the generated valuations, executed the tests and collected code coverage and mutation score results by using the Defects4J framework.

3.2. Preliminaries

In this section we introduce the necessary preliminaries for our unit test adaption approach. These preliminaries are the conventional and parameterized unit tests and how to generalize from conventional to parameterized unit tests.

3.2.1. Conventional and Parameterized Unit Tests

Conventional Unit Test (CUT): In this thesis we define CUTs as test methods within a test class. These test methods are parameterless and e.g. in the Java unit testing framework JUnit [33] decorated with an annotation like `@Test`. A CUT only explores a particular aspect of the behavior of the unit under test. CUTs also contain oracles which are represented as assertions that compare the observed behavior with expected results after executing the test. An example for a CUT is shown in Listing 3.1.

Listing 3.1: Conventional Unit Test

```
@Test
public void testAdd() {
  Calculator calc = new Calculator();
  int result = calc.add(2,3);
  assertEquals(5, result);
}
```

Parameterized Unit Test (PUT): In this thesis we define PUTs as a generalization of CUTs by allowing parameters. With a PUT the behavior of a method is tested for an entire set of input values. An example for a PUT is shown in Listing 3.2.

Listing 3.2: Parameterized Unit Test

```
@Test
public void testAdd(int a, int b,
    int expected) {
  Calculator calc = new Calculator();
```

```
    int result = calc.add(a, b);
    assertEquals(expected, result);
}
```

3.2.2. Test Generalization

To achieve test generalization from CUTs to PUTs we lean on a methodology introduced in [34]. The methodology from [34] is based on Algorithm 1. The algorithm requires a set of CUTs for a method under test *M* as input. Procedure *Parameterize(cut)* (Line 7) identifies concrete values in the CUTs ($cut \in CUTs$) and promotes them as parameters for a PUT *put*. Then in *GeneralizeTestOracle(cut, put)* (Line 8) the assertions from the CUT *cut* are generalized into generalized test oracles in the PUT *put*. The remainder of the algorithm describes four additional steps which are: the generation of CUTs based on dynamic symbolic execution, adding assumptions to guide the generation, handling mock objects to interact with external resources, and adding factory methods to generate objects for non-primitive parameters. In this work we do not apply these steps and therefore adapt the algorithm from Figure 1 for our needs as introduced in Section 3.3.2.
In practice generalizing a test oracle can be a complex task, since determining the expected outcomes for the generated input values is not trivial. Therefore the developers of the Microsoft Pex test generation tool proposed 15 PUT patterns which can be used to analyze existing CUTs and generalize the contained oracles. These 15 patterns can be found in [35]. In this thesis mainly the *Arrange, Act, Assert*-pattern was used, which is based on the assumption that a CUT is organized in 3 sections:

- **Arrange:** set up the unit under test.
- **Act:** exercise the unit under test, capturing any resulting state.
- **Assert:** verify the behavior through assertions.

As an example we can use the CUT in Listing 3.1 and the PUT in Listing 3.2. In this example as an arrangement we have to create an object from *Calculator*. To act means to exercise the method under test which is the method *add* in this example. And finally we have to add the assertions from the CUT to the PUT.

Algorithm 1 Test Generalization Algorithm [34]

Require: *CUTs* for an *m*
Ensure: *PUTs*

```
1: Set PUTs = ϕ, gAllCUTs = ϕ
2: for all cut ∈ CUTs do
3:     if gAllCUTs.contains(cut) then
4:         continue
5:     end if
6:     Set put = ϕ, gCUTs = ϕ, break = false
7:     put = Parameterize(cut)
8:     put = GeneralizeTestOracle(cut, put)
9:     gCUTs = GenerateCUTs(put)
10:    repeat
11:        while !Execute(gCUTs) do
12:            if LegalValueIssue(gCUTs) then
13:                put = AddAssumption(put)
14:            else
15:                ReportDefect()
16:                continue
17:            end if
18:        end while
19:        if !Cov(m, gCUTs) ⊇ Cov(m, cut) then
20:            if NPTypeParam(put) then
21:                put = AddFactoryMethod(put)
22:            end if
23:            if EnviInteractionIssue(m) then
24:                put = AddMockObj(put)
25:            end if
26:        else
27:            break = true
28:        end if
29:    until break
30:    PUTs.Add(put), gAllCUTs.Add(gCUTs)
31: end forreturn PUTs
```

3.3. Test Suite Adaption

In this section we first introduce our test generalization approach where we retrofit existing CUTs into PUTs. Then we show how to generate input data for these PUTs using the *t*-way combinatorial testing generation technique and give an example for our test suite adaption approach.

3.3.1. Overview

Our test suite adaption approach includes two major steps, which are first the generalization of CUTs into PUTs and second the generation of combinatorial valuations of existing input data from the CUTs. These major steps are divided into smaller substeps. An overview of these substeps is shown in Figure 3.1. On the left in Figure 3.1 we have a set of CUTs and a method under test which serve as inputs for the first step of our adaption approach. In this first step where we generalize the CUTs, we extract the input parameters from the method under test and the CUTs, and we also extract the input values from the CUTs. Next we create a PUT, where we provide the extracted input parameters and transfer the arrangements, e.g. the initialization of some objects required to execute a test, to the PUT. In the second step where we generate the combinatorial valuations from the extracted input values, we also execute the generated combinatorial valuations on the method under test and investigate manually whether the output value is correct and can be used as the expected output for the PUT. Then we provide the combinatorial valuations and the corresponding expected output values to the PUT.

3.3.2. Test Generalization

To generalize CUTs into PUTs we provide a manual approach. This approach is based on our test generalization Algorithm 2 which is a modified version of Algorithm 1. Our test suite adaption approach is limited to CUTs for a method under test which is defined as:

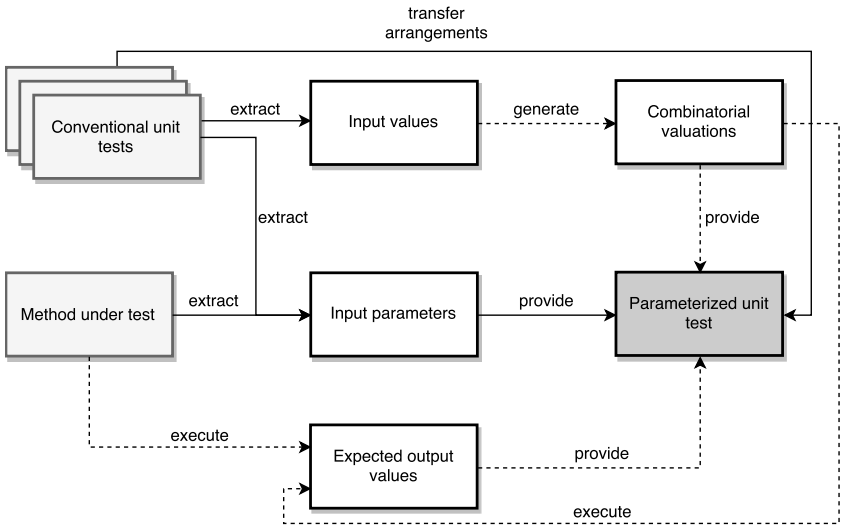

Figure 3.1.: Overview of the Test Suite Adaption

Definition 4 (Method Under Test) *A method under test m is a method from the set of methods under test M', which is a subset of the set M. M contains all methods under test from the current system under test, where each method has a different name or a distinct signature. $M' \subseteq M$ contains all methods under test that contain two or more input parameters.*

Algorithm 2 iterates through all CUTs and extracts the input parameters (line 3) and the input values for these parameters (line 4). Then we have to check if an extracted value for a certain parameter already exists in the set of input data and if not, add the value to the input data (lines 5 to 15). Finally we have to generalize the test oracle from the so far generalized test oracle in the PUT, the current retrofitted CUT, and the input parameters (line 17). The algorithm returns the PUT and the extracted input data from the CUTs.

3.3.3. Test Generation

In this work we generate combinatorial input valuations for a PUT. Therefore a PUT and the corresponding combinatorial input valuations represent

Algorithm 2 Test Generalization Algorithm for Test Suite Adaption

Require: *CUTs* for an $m \in M'$
Ensure: *put, InputData*

```
1: Set InputData = ϕ, addValue = True
2: for all cut ∈ CUTs do
3:     InputParams = Parameterize(m)
4:     data = getValues(cut)
5:     for all idx ∈ range(0, size(data)) do
6:         for all tmpData ∈ InputData do
7:             if data[idx] == tmpData[idx] then
8:                 addValue = False
9:             end if
10:        end for
11:        if addValue then
12:            InputData.add(idx, data[idx])
13:        else
14:            addValue = True
15:        end if
16:    end for
17:    put = GeneralizeTestOracle(put, cut, InputParams)
18: end for
19: return put, InputData
```

a test suite T_t as introduced in Definition 3. In this work we generate combinatorial valuations for strength t=2. As shown in Algorithm 2 we use only input data of existing CUTs for the generation of combinatorial valuations.

3.3.4. Example

In this section we give an example that shows the test suite adaption approach. As example we use a CUT from the *NumberUtilsTests* class within the Apache Commons Lang application. The CUT is shown in Listing 3.3.

From the test case in Listing 3.3 we can see that the method under test has three input parameters. The test also contains four assertions. Each assertion contains a call of the method under test with a different combination of the input values. This test contains the values `low` and `mid` for the first input parameter of the method under test, `low`, `mid`, and `high` for the second input parameter, and `low` and `high` for the third parameter. These combinations provide 68.75% 2-way combinatorial coverage and 33.33% 3-way combinatorial coverage as shown in Figure 3.2.

Listing 3.3: Conventional Unit Test

```
@Test
public void testMinimumByte() {
    final byte low = 123;
    final byte mid = 123 + 1;
    final byte high = 123 + 2;
    assertEquals(
      "minimum(byte,byte,byte) 1 failed",
      low,NumberUtils.min(low, mid, high));
    assertEquals(
      "minimum(byte,byte,byte) 1 failed",
      low,NumberUtils.min(mid, low, high));
    assertEquals(
      "minimum(byte,byte,byte) 1 failed",
      low,NumberUtils.min(mid, high, low));
    assertEquals(
```

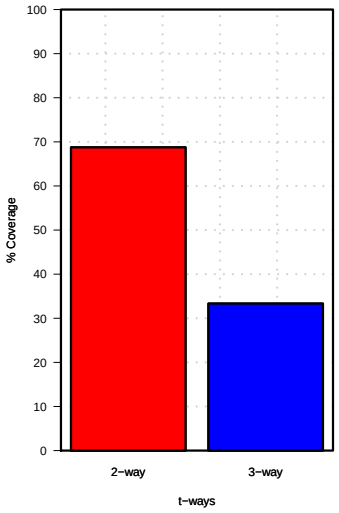

Figure 3.2.: Combinatorial Coverage of Existing CUT.

```
        "minimum(byte,byte,byte) 1 failed",
        low,NumberUtils.min(low, mid, low));
}
```

Listing 3.4: Parameterized Unit Test

```
@Test
public void testMinimumByte(
        byte low, byte mid, byte high,
        byte expected) {
    assertEquals(expected,
        NumberUtils.min(low, mid, high));
}
```

After generalizing the CUT from Listing 3.3 into the PUT in 3.4 we generate the combinatorial valuations as shown in Table 3.1 which serve as inputs for the PUT and obtain the expected output values manually.

Table 3.1.: 2-way Combinatorial Valuations for the Input Values Extracted from the CUT.

low	low	mid
low	mid	high
low	high	low
mid	low	high
mid	mid	low
mid	high	mid
high	low	low
high	mid	mid
high	high	high

3.4. Empirical Study

In this section we introduce the subject applications and tools which we chose to answer the following research questions:

- **RQ1:** Does a 2-way combinatorial valuation of existing data in a test suite of CUTs increase code coverage and mutation score?
- **RQ2:** Does a manually written test suite of CUTs in general grow when replacing the CUTs with PUTs where the parameter valuations represent 2-way combinatorial valuations of existing values in the test suite?

To answer these questions we provide data about combinatorial coverage in existing test suites of CUTs and give results of code coverage and mutation score for both CUTs and PUTs in this section.

3.4.1. Subject Applications

For our empirical study we selected four Java applications from the Defects4J [32] repository [36]. Defects4J provides a framework to checkout, compile, and test several open source Java applications. For each of these applications multiple versions are available where each version is either a buggy version for a certain bug or a fixed version for that particular bug. To assess the quality of the provided test suites the Defects4J framework also

Table 3.2.: Basic Information About Selected Applications for Empirical Study

Name	Versions	Version	Modified Class	Mutants
Lang	65	fixed 1	NumberUtils	913
Math	106	fixed 4	euclidean.threed.SubLine	74
JFreeChart	26	fixed 8	data.time.Week	165
Joda-Time	27	fixed 12	LocalDate	610

comes integrated with the code coverage tool Cobertura [37] and the Major mutation framework [38]. With Cobertura we measure line and branch coverage of the provided test suites and with Major we measure the mutation score of the test suites.

The four Java applications which we selected are Apache Commons Lang[1], Apache Commons Math[2], JFreeChart[3], and Joda-Time[4].

3.4.2. Study Setup

For the selected applications in our study we give the number of existing versions, the version we applied in this study, the class that was modified by a patch to fix a bug, and the number of generated mutants in Table 3.2. For each version a buggy one and a fixed one exists. In each buggy version only a single bug is implemented. In the fixed versions the patch that fixes the bug is already integrated.

For each of the four applications we selected a class that was modified to fix a certain bug. Defects4J contains a list of relevant test classes for each version of the applications. These relevant test classes contain CUTs which we retrofitted manually into PUTs by applying Algorithm 2. Since we adapt the existing tests by generating combinatorial valuations for the PUTs we retrofitted only CUTs which used more than two different input variables for the test and at least two different values for these input variables are present

[1] https://commons.apache.org/proper/commons-lang/
[2] http://commons.apache.org/proper/commons-math/
[3] http://www.jfree.org/jfreechart/
[4] http://www.joda.org/joda-time/

Table 3.3.: Example test suite with four combinatorial valuations, four input parameters and two input values for each parameter. [40]

a	**b**	**c**	**d**
0	0	0	0
0	1	1	0
1	0	0	1
0	1	1	1

in the test class. From these values we generated combinatorial valuations by using the combinatorial testing tool ACTS 3.0[5] (Automated Combinatorial Testing for Software)[6]. Then we generated 2-way combinatorial valuations for the PUTs. We manually added the expected values for the test oracles to each combinatorial valuation.

To show the combinatorial coverage of the existing CUTs which we retrofitted into PUTs we used the combinatorial coverage measurement tool CCM [39]. Combinatorial coverage or t-way combinatorial coverage for a method under test with n input parameters is the proportion of t-way combinations of these n input parameters for which all input values are covered. E.g. if we have a test suite as shown in Table 3.3 with four binary variables **a, b, c, d** and each row represents a combinatorial valuation then there should be six 2-way combinations **ab, ac, ad, bc, bd, cd** in the test suite. In Table 3.3 only **bd** and **cd** have all possible combinations (00, 01, 10, 11) covered. So the combinatorial coverage is 2/3=66.6%.

3.4.3. Combinatorial Coverage of Conventional Unit Tests

In this section we show the combinatorial coverage of the existing CUTs. Since we do not use constraints in our example applications for the generation of combinatorial valuations with ACTS, we automatically obtain

[5] http://csrc.nist.gov/groups/SNS/acts

[6] The input models are freely available for download from https://bitbucket.org/hfelbinger/iwct2018

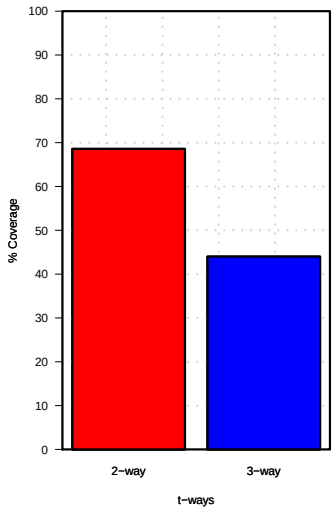

Figure 3.3.: Combinatorial Coverage of CUTs from Apache Commons Lang.

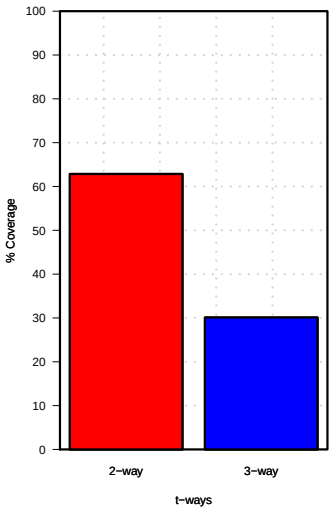

Figure 3.4.: Combinatorial Coverage of CUTs from Apache Commons Math.

100% combinatorial coverage for the newly generated 2-way combinatorial valuations.

As shown in Figure 3.3 the combinatorial coverage of the CUTs from the Apache Commons Lang example is 68.57% for 2-way combinations and 44% for 3-way combinations. To achieve 100% 2-way combinatorial coverage we had to increase the number of test cases from 56 CUTs to 111 combinatorial valuations for the PUT.

Figure 3.4 shows that the combinatorial coverage of the CUTs from the Apache Commons Math example is 62.85% for 2-way combinations and 30.1% for 3-way combinations. There are 7 CUTs, which we increased to 81 combinatorial valuations to achieve 100% 2-way combinatorial coverage for Apache Commons Math.

The combinatorial coverage for 2-way combinations is 57.68% and for 3-way combinations is 25% for the CUTs of the JFreeChart example, as shown in Figure 3.5. We increased the 7 CUTs to 12 combinatorial valuations to achieve 100% 2-way combinatorial coverage for JFreeChart.

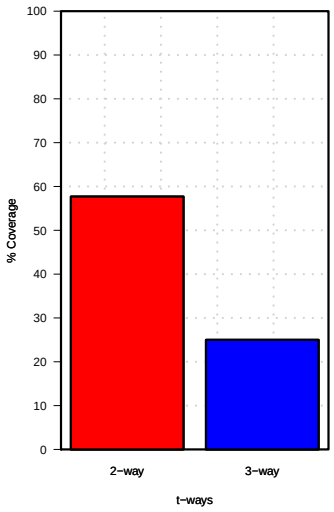

Figure 3.5.: Combinatorial Coverage of CUTs from JFreeChart.

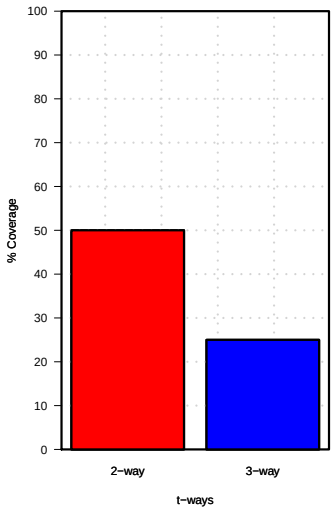

Figure 3.6.: Combinatorial Coverage of CUTs from Joda-Time.

As shown in Figure 3.6 the 2-way combinatorial coverage of the CUTs is 50%, 3-way is 25% for the Joda-Time example. Increasing the number of CUTs from 3 to 6 combinatorial valuations for the PUT resulted in 100% 2-way combinatorial coverage.

3.4.4. Coverage

The results for line coverage and condition coverage are shown in Table 3.4. These results show that for 3 examples line and condition coverage are equivalent for CUTs and PUTs. Only for the JFreeChart example, both line and condition coverage, increase by about 5%.

3.4.5. Mutation Score

The mutation scores for the CUTs and PUTs are listed in Table 3.5. Additionally the sizes of the test suites are shown. The test suite size is either the

Table 3.4.: Code Coverage results for CUTs and PUTs.

name	test type	line cov. (%)	cond. cov. (%)
Lang	CUT	10.6	4.6
	PUT	10.6	4.6
Math	CUT	20.8	25
	PUT	20.8	25
JFreeChart	CUT	31.9	17.6
	PUT	36.2	22.1
Joda-Time	CUT	4.3	0
	PUT	4.3	0

Table 3.5.: Mutation score results for CUTs and PUTs and the respective test suite sizes.

name	test type	ms (%)	test suite size
Lang	CUT	37.9	56
	PUT	41.4	111
Math	CUT	80	7
	PUT	100	81
JFreeChart	CUT	28	7
	PUT	42.1	12
Joda-Time	CUT	39.1	3
	PUT	39.1	6

number of CUTs or the number of combinatorial valuations for a PUT. For the Apache Commons Lang example the mutation score increases by about 5% whereas the test suite size increases by almost 100% from CUTs to PUTs. The mutation score for the Apache Commons Math example increases by 20% and the test suite size by more than 1000%. For JFreeChart the test suite size also increases by about 100% and the mutation score increases by 14%. Finally for Joda-Time the mutation scores are equivalent but the test suite size increases by 100%.

3.5. Evaluation and Threats to Validity

In this work we did not change or add any input values when generalizing CUTs into PUTs. If the programmer/tester missed to create CUTs with input values that are boundary values or equivalence classes, we also did not have these values for the generation of combinatorial valuations. Therefore the growth of code coverage and mutation score are not that high compared to the growth of the test suite size. As an answer for RQ1 we derive from our results, that a 2-way combinatorial valuation of existing data in a test suite of CUTs increases code coverage and mutation score.
Answering RQ2 depends on the extent of the existing test suite. If the programmer/test would have created a test suite that has 100% 3-way combinatorial coverage it should be possible to reduce the test suite size for 2-way combinations. Since this is usually not the case as also shown in our empirical results, we assume that generally the test suite size grows.

In our empirical evaluation we did not eliminate equivalent mutants. When investigating the examples manually we figured out that there were several equivalent mutants for each example application except for Apache Commons Math.
Also the level of abstraction when selecting the input data for the combinatorial valuation might affect the results. E.g. in one case we used every single integer value in the CUTs and for another case abstracted the integer values in different calls of a constructor to input values where all these values are equivalent:

- **Case 1:** x.call(1,2,3) and x.call(2,3,4) result in 3 different input parameters with 2 values each.
- **Case 2:** x.call(1,2,3) and x.call(2,3,4) result in 1 input parameter with 2 values where the values are (1,2,3) and (2,3,4).

It is also very important to say that our approach only works for methods under test with at least two input variables.

3.6. Related Work

When Tillmann et al. [19] introduced PUTs, their idea was to use symbolic computation to derive test inputs. They use the path conditions within the program under test and find covering input values for each path. PUTs have their origins in testing algebraic specifications. Work on testing algebraic specifications was started by Bernot et al. [41]. They use axioms to describe the test purpose, to obtain concrete data, and to derive new theorems. In this work we use PUTs just as a generalization from existing CUTs and generate the test inputs with a combinatorial testing tool. The input values used for the generation of the test inputs, are already included in the CUTs. We assume that the CUTs were created manually and therefore the programmer that created the tests also selected the input values very wisely.

Thummalapenta et al. [34] investigated the costs and benefits of retrofitting CUTs as PUTs. They propose a methodology that helps to systematically retrofit existing CUTs into PUTs. Furthermore they provide an empirical analysis which shows that with PUTs the fault detection capabilities and code coverage increase with feasible effort. In their work they used the test generation tool Pex [42] which accepts PUTs and uses dynamic symbolic execution to generate test inputs. In contrast we use the combinatorial testing tool ACTS 3.0 to generate the test inputs, but we have to find the expected output values manually, because there is no suitable technique to generate test oracles automatically in combinatorial testing.

In [43] Fraser and Zeller present an approach to generate PUTs with symbolic pre- and postconditions to characterize test inputs and test results. Their empirical evaluation results show that the PUTs are more expressive, need fewer computation steps, and achieve higher code coverage than CUTs. In their work Fraser and Zeller use a search based approach to iteratively derive new test inputs and mutation testing to identify possible oracles [44].

Saff et al. [45] use the term theory-based testing for their generalization approach of example based testing. Theories are closely related to PUTs. They use an input generator that tries to achieve path coverage, while attempting to execute every possible outcome. The input generator uses a combination of static and dynamic analyses to iteratively explore a certain

unit of code. Therefore they use constraint solving and some heuristics that allow to explore otherwise unreachable paths e.g. for Integer values a heuristic would be to use the constant values 0, 1, and -1.

In their very recent work Peleg et al. [46] present a framework that synthesizes property based tests from existing CUTs. Since property based testing combines parametric testing with value generators, property based testing is closely related to parametric unit testing. The difference is that property based testing uses value generators that generate randomly large numbers of inputs defined by a generator to check whether the assertions hold. They implemented the tool called JARVIS to automatically generalize CUTs. In contrast we generalized the CUTs into PUTs manually.

In [18] Xie et al. propose a mutation analysis approach for analyzing PUTs. In this work we compare the mutation scores of CUTs and PUTs where both use the same input data, but with different or more combinations. Xie et al. propose a set of mutation operators for systematically mutating PUTs written by developers.

3.7. Summary

Each software under development should contain unit tests. These unit tests are mainly manually written CUTs. PUTs are a generalized form of CUTs. In this chapter we present an approach to manually generalize from CUTs to PUTs, extract the input data from the CUTs, generate combinatorial valuations from these input data, and derive the expected outcomes for the combinatorial valuations.
We provide an empirical study using 4 Java examples from the Defects4J collection. The results show that 2-way combinatorial coverage for the CUTs in these examples is in the range of 50 to 70%. Nevertheless our approach facilitates a growth of mutation score and code coverage.

Part III.

Model Inference Based Quality Assessment

4. Test Suite Quality Assessment

When testing programs or systems, in practice the question when to stop testing naturally arises. To answer this question, it is significant to assess the quality of the underlying test suite in terms of its adequacy with respect to extract faults for a particular program or system under test. In testing practice we are either using certain coverage criteria or mutation score that a test suite has to ensure to serve this purpose. Where coverage gives an indication whether a test suite causes the execution of certain parts of the system under test, alone or in combination, mutation score is a measure for the fault detection capabilities of a test suite directly. Measuring coverage causes a small computational overhead during execution, whereas the mutation score requires the execution of a certain amount of program variants, i.e., the mutants. Therefore, computing the mutation score might be too time consuming in practice.

In this chapter, we follow an alternative approach for assuring the fault detection capabilities of test suites. The approach is based on machine learning. The underlying idea is to extract a model from the test suite and to compare the obtained model with the original program or a reference test suite. In case both the model and a model from the Software Under Test (SUT) are equivalent or very similar, it is evident that the test suite captures the behavior of the SUT in sufficient detail. Otherwise, there are test cases missing in the test suite. The question now is whether the difference between the learned model and the SUT is a good measure for assuring test suite quality?

4.1. Existing Quality Assessment Methods

In this section we describe the three assessment methods mutation score, code coverage, and combinatorial coverage.

4.1.1. Mutation Score

The mutation score is the result of mutation testing. Mutation testing [47] is a fault-based testing technique where modifications of the SUT lead to faulty versions of the SUT. These faulty versions are called mutants. A mutant is said to be a killed mutant, if a test suite can distinguish the mutant from the original program. Mutation score is defined as:

Definition 5 (Mutation score) *Mutation score μ is the proportion between killed mutants k and the number of existing mutants n.*

$$\mu = \frac{k}{n}$$

In this chapter we used mutants for Java source code from the following categories, which are described in detail in [48]:

- Operator Replacement Binary (ORB): Replace all occurrences of arithmetic (AOR), logical (LOR), shift (SOR), conditional (COR), and relational (ROR) operators with all valid alternatives.
- Literal Value Replacement (LVR): Force literals to take a positive value, a negative value, and zero. Additionally, all reference initializations are replaced by null.

Mutation score is in [0,1]. We assess a test suite to be of maximum adequacy if the mutation score is 1 and a test suite to be inadequate if the mutation score is 0.

4.1.2. Code Coverage

To assess code coverage the degree to which test cases exercise the source code of the program is measured. There exist several different metrics to assess code coverage [49], e.g.:

- Statement Coverage: To reach full statement coverage, each statement of the program code has to be executed at least once.
- Decision Coverage: To reach that 100% decision coverage a test suite has to be provided s.t. each decision has a true and a false outcome at least once. In other words, each branch direction must be traversed at least once.
- MC/DC Coverage: To achieve 100% modified condition/decision coverage (MC/DC) requires a test suite where each condition within a decision is shown by execution to independently and correctly affect the outcome of the decision [50].

4.1.3. Combinatorial Coverage

Combinatorial testing is a black box testing technique using valuations for input parameters or configuration parameters. Therefore, for a given test set for n parameters, simple t-way combinatorial coverage is the proportion of t-way combinations of n variables for which all valuations are fully covered [40].

4.2. Model Inference Based Quality Assessment

In this thesis we investigate a new method for test suite evaluation that is based on an inferred model from the test suite. The idea is to use the similarity between the inferred model and the system under test as a measure of test suite adequacy, which is the ability of a test suite to expose errors in the system under test. Therefore, we infer a model from a test suite by learning a decision tree. As shown in Figure 4.1 we need a test suite T which is divided into two distinct subsets T_{equiv} and T_{dt}. For the empirical

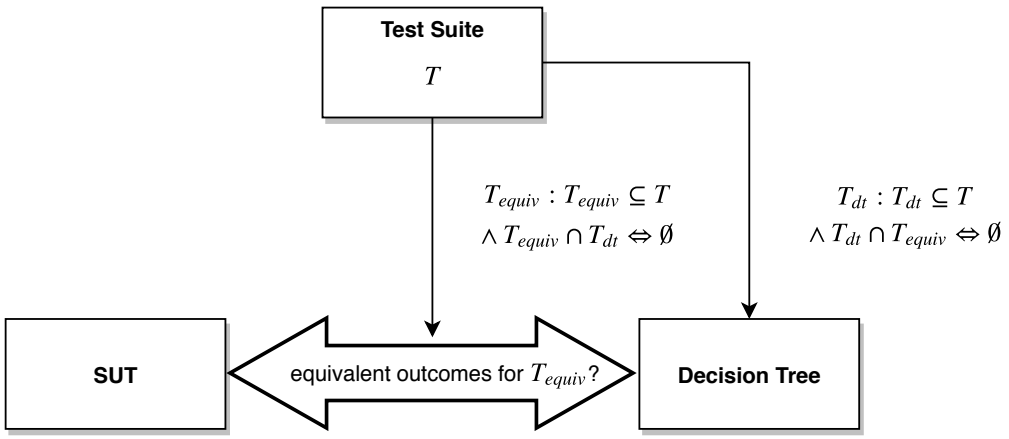

Figure 4.1.: Schema of model inference based quality assessment.

evaluation we decided to use a relative size ratio of 9:1 for subsets T_{equiv} and T_{dt} respectively. The test suite for which we assess its adequacy is T_{dt}. Therefore we infer a model from T_{dt}. The second subset T_{equiv} is used to assess the adequacy by comparing the outcomes from the SUT and from the inferred model when executing the test suite. If all outcomes are equivalent, we assess a test suite T_{dt} to be adequate. If different outcomes appear from the SUT and the inferred model when executing the same test case from T_{equiv}, we calculate the root mean squared error to assess the adequacy defined as follows:

Definition 6 (root mean squared error) *The root mean squared error rmse measures the differences from the outcomes of an SUT and a decision tree of n test cases.*

$$rmse = \sqrt{\frac{(p_1 - a_1)^2 + ... + (p_n - a_n)^2}{n}}$$

$p_1, p_2, ..., p_n$ are the outcomes from the inferred model
$a_1, a_2, ..., a_n$ are the outcomes from the SUT

The root mean squared error is in [0,1] where 0 indicates maximum adequacy and 1 indicates an inadequate test suite.
If the outcomes of the learned decision tree and the SUT are different for the ith test case, the difference of $(p_i - a_i)$ depends on the number of different categories the decision tree and the SUT categorize and on possible

misclassifications while learning the decision tree. The difference of $(p_i - a_i)$ is

$$(p_i - a_i) = \begin{cases} \frac{2}{\#categories} & \text{if no misclassifications exist,} \\ \frac{P(p_i|leaf)}{\#categories} & \text{if misclassifications exist,} \\ 0 & \text{if } p_i \text{ equals } a_i. \end{cases}$$

Wrongly classified, or misclassified test cases cause probability distributions $P(p_i|leaf)$ for the classified test cases at the leaf nodes. This probability distribution affects the root mean squared error, but depends on the number of misclassifications and is calculated for each learned decision tree individually.

The decision tree learning method we used in this work was the C4.5 algorithm as introduced in [51]. In real world applications only a test suite T_{dt} exists, which requires in addition to create a set T_{equiv} to assess the adequacy of T_{dt}.

5. Empirical Study Of Correlation Between Mutation Score And Model Inference Based Test Suite Adequacy Assessment

This chapter is based on the work "Empirical Study of Correlation Between Mutation Score and Model Inference Based Test Suite Adequacy Assessment" [6].

In this chapter we address the question whether the difference between an inferred model from a test suite and the SUT is a good measure for assuring test suite quality. To give an answer to this question, we carried out an experimental evaluation that is based on one specific setting.

The setting comprises a set of Java programs together with a corresponding test suite from which we learn decision trees. Checking program equivalence is an undecidable problem of computer science in general. However, in order to check for equivalence or at least to establish reasons about equivalence or similarity of programs, someone might use statistical methods. When taking a sufficient number of test inputs and executing both, the program and the potentially equivalent program, leads to always the same output, the evidence of equivalence raises. We make use of the same idea to define similarity as a measure based on behavioral differences when applying a random set of test cases to both, the model and the SUT as introduced in Section 4.2. To measure the behavioral differences we make use of the root

mean squared error between the outputs of the model and the SUT for the tests.

In order to judge whether the model inferred from the test suite can be a measure for the fault detection capabilities, the carried out experimental evaluation compares the mutation score of the test suite with the similarity between the inferred model and the test suite. In case both measures correlate, model inference and mutation score can be assumed as equivalent approaches for assuring fault detection capabilities of test suites. The contributions of this chapter are as follows:

- Introduction of an alternative approach for test suite adequacy assessment based on machine learning, where we infer a model from the test suite and compute the similarity between the model and the SUT.
- An experimental evaluation of the model inference approach in order to answer the question whether model similarity and mutation score are equally good in assuring fault detection capabilities of test suites.

5.1. Pearson Correlation

In this work we investigate whether a relationship between mutation score and a model inference based adequacy assessment exists. To quantify this relationship we calculate the Pearson Correlation coefficient [52] between mutation score and the root mean squared error of the model inference based assessment. The Pearson Correlation coefficient indicates the extent to which two variables have a linear relationship and is defined as:

Definition 7 (Pearson Correlation Coefficient) *The Pearson Correlation coefficient r measures the linear correlation between two variables.*

$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

n is the number of test cases,

Table 5.1.: Strength interpretations of the Pearson Correlation coefficient r

r	strength
0.00-0.19	very weak
0.20-0.39	weak
0.40-0.59	moderate
0.60-0.79	strong
0.80-1.0	very strong

$x_1, x_2, ..., x_n$ are the results for the root mean squared error of T_{dt},
$y_1, y_2, ..., y_n$ are the results for the mutation score of T_{dt},
$\bar{x}$ is the sample mean for $x_1, ..., x_n$ (and analogously for $\bar{y}$)

The resulting Pearson Correlation coefficient is in [-1,1]. The linear relationship is indicated by the direction (sign +/-) and the strength of r. According to Evans [53] the result for the Pearson Correlation coefficient can be interpreted in strength by the value of r as shown in Table 5.1.

When interpreting the direction of the Pearson Correlation coefficient a value of $r = 0$ indicates no correlation, a value $r < 0$ indicates negative correlation, and a value $r > 0$ indicates positive correlation.
For the empirical evaluation we created a sample of 100 test suites T_{dt} where all test suites in the sample have the same size. The test cases for T_{dt} are selected randomly. For each member of the sample we calculated mutation score and the root mean squared error.
To obtain a reasonable result from the calculation of the Pearson Correlation coefficient, the data should be sampled from populations with normal distributions. To test whether the 100 mutation score results and the 100 root mean squared error results for each example are normally distributed, we used the Shapiro-Wilk test [54]. From the Shapiro-Wilk test we conclude that our input data to calculate the Pearson Correlation coefficient most likely come from a normally distributed population.
Since we assume that a growing mutation score indicates a reduction of the root mean squared error, we expect the Pearson Correlation coefficient to be the right tool to confirm this assumption.

5.2. Experimental Results

For the empirical evaluation of the Pearson Correlation coefficient between mutation score and root mean squared error we used six example programs and their corresponding test suites. As depicted in Figure 4.1 we divided the given test suite T of each program into a subset $T_{dt} \subset T$ and a subset $T_{equiv} \subset T$. We take T_{dt} as the test suite for which adequacy is assessed on the one hand by mutation score and on the other hand by model inference. To obtain the mutation score we have to execute the test cases in T_{dt} for each mutant. To get the root mean squared error an additional set of test cases T_{equiv} is required, but these additional test cases have to be executed only once. We presume that all test cases in T pass if they would be executed on the original SUT.

5.2.1. Examples

For the experimental evaluation we make use of six Java programs:

1. TCAS: For the traffic collision avoidance system
   (TCAS) an implementation in C with a test suite and mutants can be found in [55]. Here we use a Java implementation. Also the mutants were translated into Java.
2. BMI: The body mass index (BMI) example is a program that categorizes a person by means of weight and height into one of five different categories. For this example we used the source code published in [56].
3. Triangle: The triangle example program [57] determines whether the inputs correspond to a valid triangle and if so, whether the valid triangle is equilateral, scalene, or isosceles.
4. POP3: The POP3 example is a state machine. A graphical representation of the implemented state machine is shown in Figure 5.1.
5. CAS: The car alarm system (CAS) example is a state machine. A graphical representation of the implemented state machine is shown in Figure 5.2.

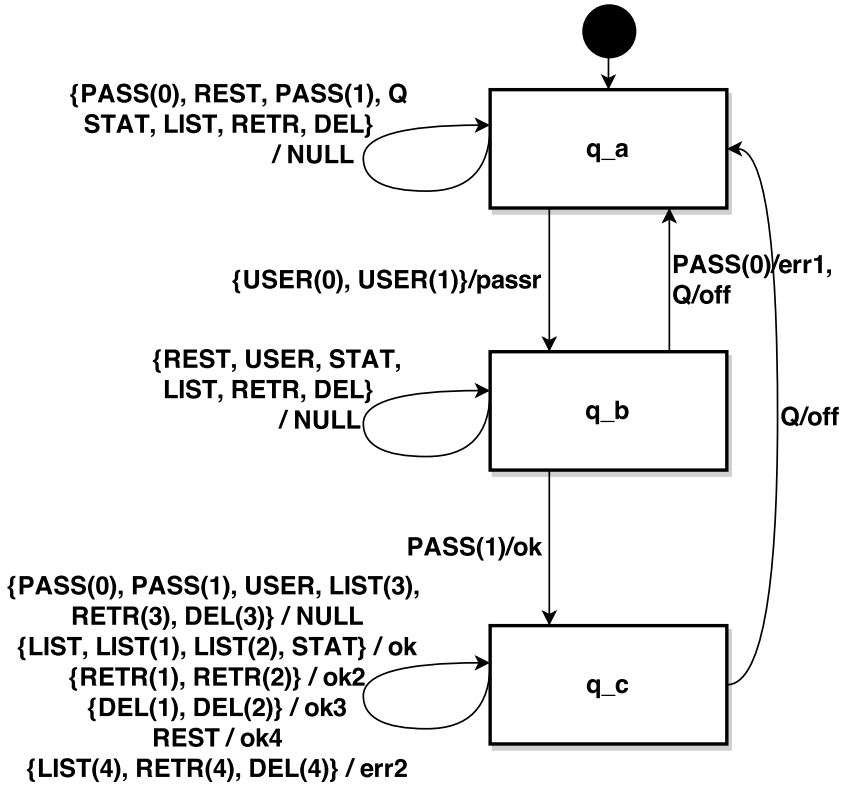

Figure 5.1.: State machine showing the POP3 example.

6. UTF8: The Guava UTF8 example is a part of Google's Guava library, which checks if an input sequence of bytes is a well formed UTF8 encoded input. The source code of the UTF8 function can be found at [58].

The input types for the examples are either continuous or enumerated, where an enumerated input is limited by a finite number of different values. The outputs for all six examples in this work are categorical.
The examples TCAS, BMI, UTF8 and Triangle appear as single transition functions, without any externally observable states, that process input vectors of continuous type values. The input for the UTF8 example is a vector of four bytes where validity is checked for the input bytes consecutively. There-

fore the UTF8 example represents a state machine, without any externally observable states, with transitions after a valid input byte was processed. If the input vector for the UTF8 example contains less than four bytes, the other bytes remain unknown (indicated by '?'). Decision tree learning can be used even when some input values are unknown. Missing input values for decision tree learning are estimated. For estimation either the value that is most common among the test vectors for the input value is used, or probabilities are estimated for the input values. For the four examples, namely TCAS, BMI, Triangle, and UTF8 we consider a test case tc to be a test vector of k input values and an expected output value, e.g., $tc = (in_1, .., in_k, out)$. For the POP3 and CAS examples a test case is a sequence s of n test vectors, e.g., $s = \langle tc_1, .., tc_n \rangle$ where n varies. Each test vector in s consists of four values where the first value is the input value, the second value is the expected outcome after executing the previous test vector, the third value is the current state, and the fourth value is the expected outcome after executing the current test vector. The inputs for the POP3 and the CAS examples are enumerated.

The attributes source lines of code *(SLOC)*, number of output categories *(categories)*, size of the test suite T *(*$|T|$*)*, and number of mutants *(# mutants)* for the six examples are given in Table 5.2. Despite the mutants for the TCAS example the mutants for the remaining examples use only a single mutation per mutant. For the TCAS example we used the existing test suite and generated random test cases for the other examples. The size of a test suite is the number of test cases it contains. Table 5.3 displays different properties for T, the learned decision tree from T, and the control flow graph for the six examples. The least number of inputs (k) is two for the BMI example and the maximum is 12 for the TCAS example. As mentioned before k is 3 for POP3 and CAS, which are the input value, the previous expected outcome, and the current state. Because for CAS and POP3 each test case is a sequence of test vectors, the number of test vectors n is much higher than the size of the test suite. For the examples BMI, Triangle, and UTF8 misclassifications (*misc.*) appear even when learning the decision tree from T. The decision tree with the most nodes is the decision tree for TCAS with a *size* of 372 nodes and 256 leaves. Also for TCAS the maximum number of edges which can be counted when traversing the decision tree from the root node to a leaf node is the highest with *len.* 12 whereas *len.* is just 2 for

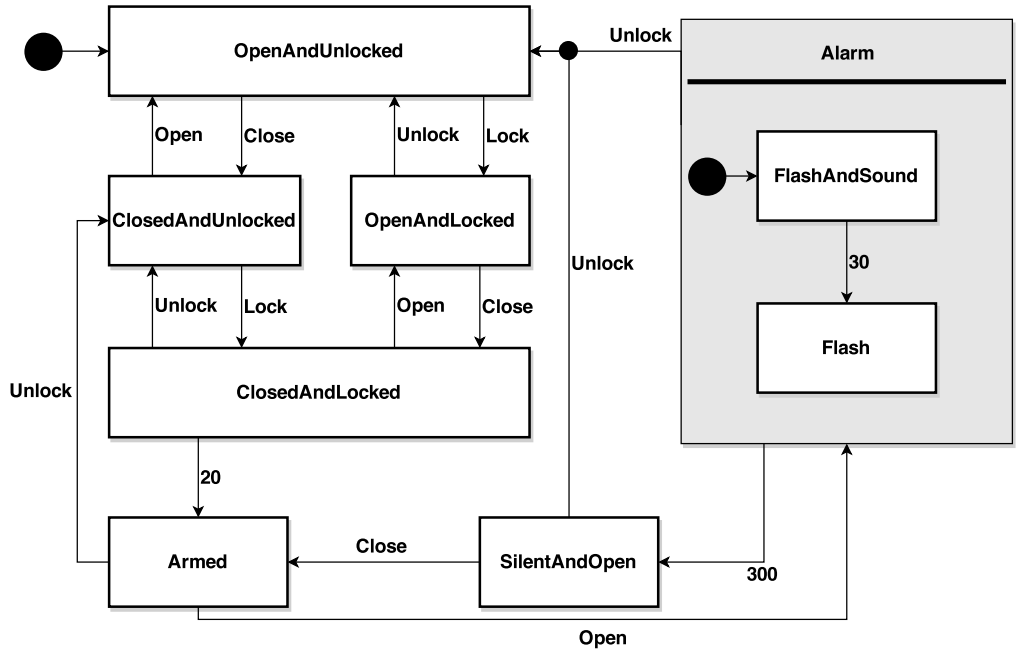

Figure 5.2.: State machine showing the CAS example [59].

Table 5.2.: Attributes SLOC (source lines of code), size of the test suite TS, and number of mutants of the example programs.

Name	*SLOC*	*categories*	$\|T\|$	*#mutants*
TCAS	100	3	1,545	41
BMI	19	5	1,000	28
Triangle	30	4	1,000	35
POP3	122	10	1,000	167
CAS	110	5	1,000	167
UTF8	56	2	1,000	147

Table 5.3.: The properties of the initial test suite, the learned tree, and the control flow graph for the 6 examples.

	test suite		learned tree				cfg
ex.	*k*	*n*	*misc.*	*size*	*leaves*	*len.*	*size*
TCAS	12	1545	0	372	256	12	431
BMI	2	1000	9	107	54	9	67
Trian.	3	1000	246	57	29	14	99
POP3	3	25117	0	165	123	2	695
CAS	3	54437	0	33	28	2	511
UTF8	4	1000	83	33	17	11	229

POP3 and CAS. Further we extracted the control flow graphs (cfg) of the six examples and counted the nodes in the graphs where the highest numbers of nodes are given by POP3 and CAS with *size* 695 and 511 respectively.

5.2.2. Results

As introduced in Sections 4.1.1 and 4.2 we obtained the values for mutation score and root mean squared error to calculate the Pearson Correlation coefficient for the six examples described in Section 5.2.1. To calculate the values for the root mean squared error we used the Java library Weka 3.7 [60].

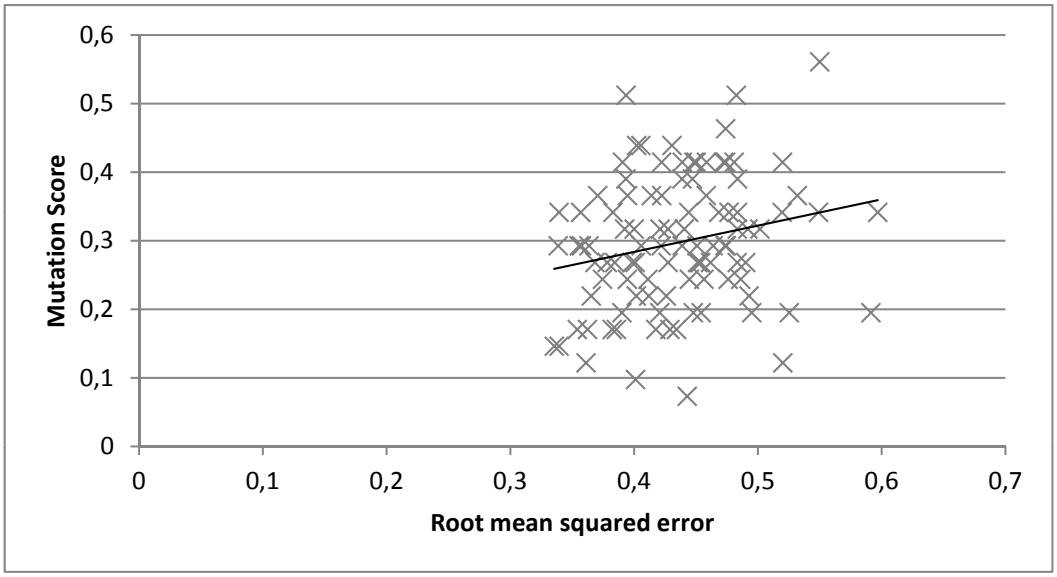

Figure 5.3.: Correlation of mutation score and root mean squared error for the TCAS example.

We selected 100 random subsets for T_{dt} from each of the six example test suites and acquired mutation score and root mean squared error for each of the T_{dt}s.

Figure 5.3 shows the results for mutation score and root mean squared error for the TCAS example. The Pearson Correlation coefficient for the TCAS example is $r_{TCAS} = 0.22$. The mutation score results are in the interval [0.07, 0.57] and the root mean squared error results are in the interval [0.33, 0.6].

In Figure 5.4 the results for the Triangle example are shown. The mutation score results are in the interval [0.33, 0.78] and the root mean squared error results are in the interval [0.45, 0.65]. For the Triangle example the Pearson Correlation coefficient is $r_{Triangle} = 0.332$.

For the BMI example the resulting mutation scores and root mean squared errors are shown in Figure 5.5. The mutation score results are in the interval [0.17, 0.86] and the root mean squared error results are in the interval [0.32,

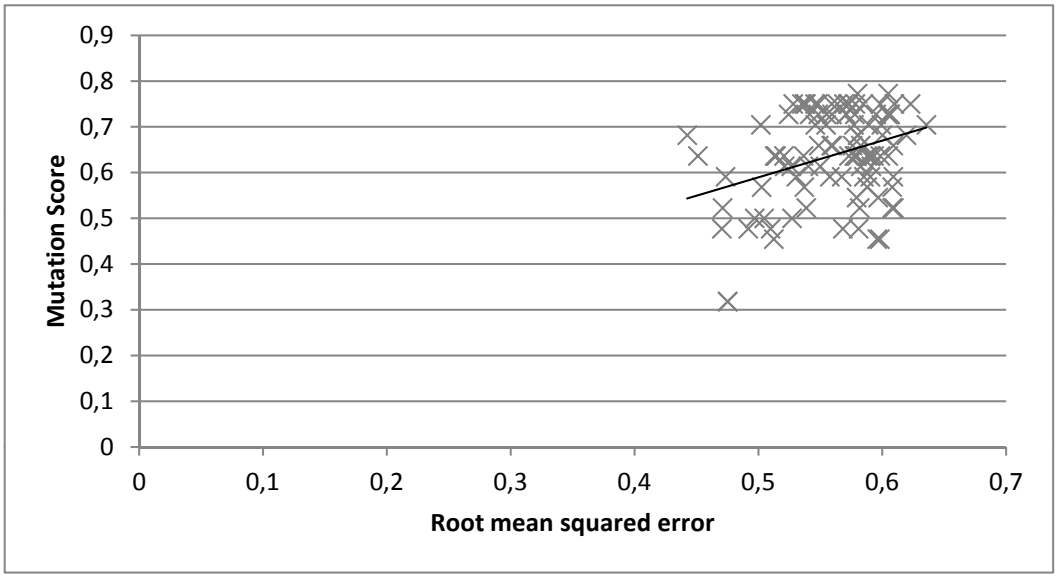

Figure 5.4.: Correlation of mutation score and root mean squared error for the Triangle example.

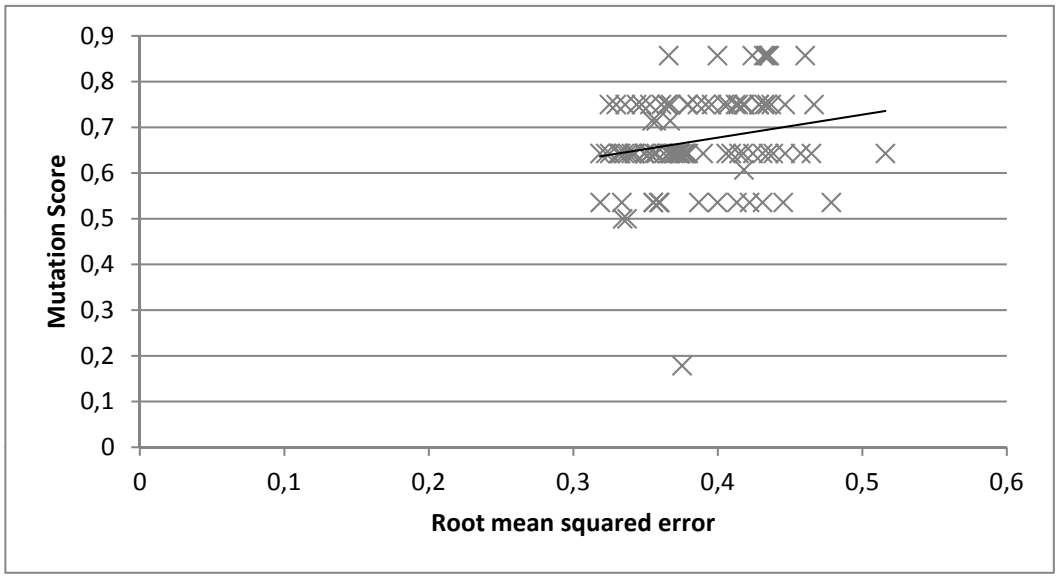

Figure 5.5.: Correlation of mutation score and root mean squared error for the BMI example.

0.53]. The Pearson Correlation coefficient for the BMI example is $r_{BMI} = 0.21$.

The results of mutation score and root mean squared error for the UTF8 example are shown in Figure 5.6. The mutation score results are in the interval [0.11, 0.42] and the root mean squared error results are in the interval [0.32, 0.72]. The resulting Pearson Correlation coefficient for the UTF8 example is $r_{UTF8} = 0.202$.

Figure 5.7 shows the 100 mutation score and root mean squared error results for the POP3 example. The mutation score results are in the interval [0.24, 1.0] and the root mean squared error results are in the interval [0.11, 0.22]. For the POP3 example the Pearson Correlation coefficient is $r_{POP3} = -0.632$.

In Figure 5.8 the results for the CAS example are shown. The mutation score results are in the interval [0.1, 0.38] and the root mean squared error results

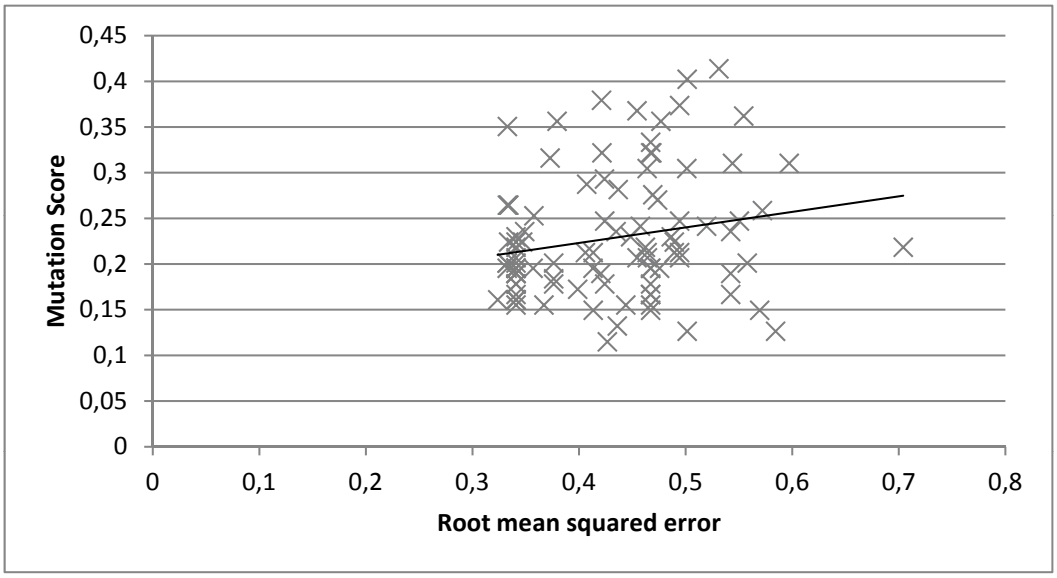

Figure 5.6.: Correlation of mutation score and root mean squared error for the UTF8 example.

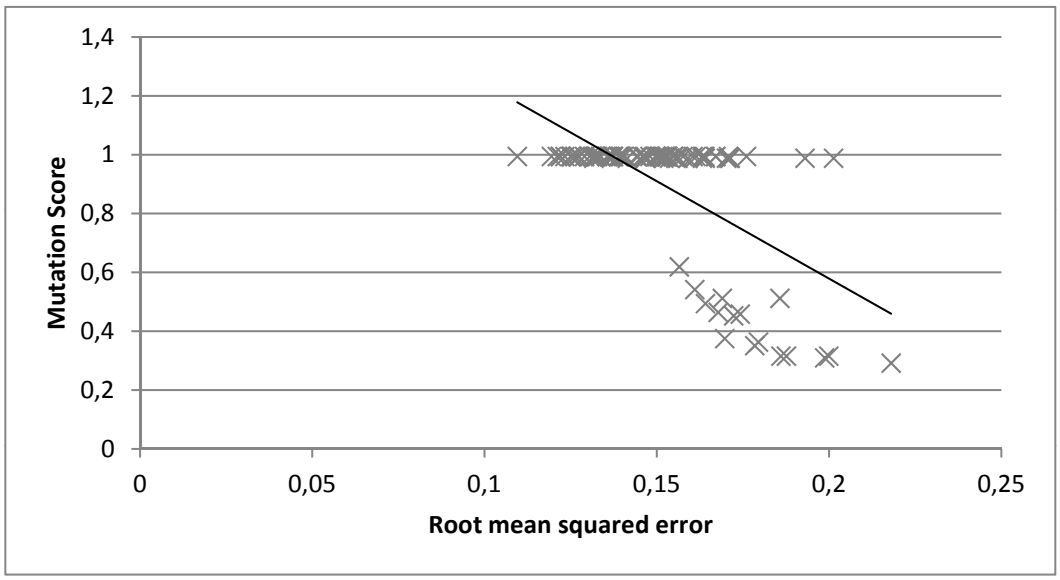

Figure 5.7.: Correlation of mutation score and root mean squared error for the POP3 example.

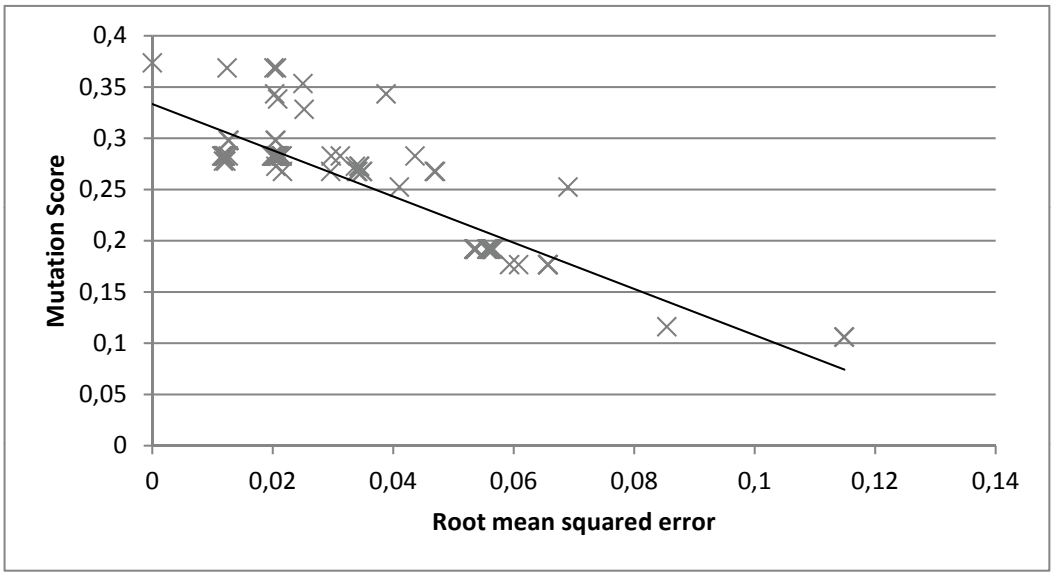

Figure 5.8.: Correlation of mutation score and root mean squared error for the CAS example.

are in the interval [0.0, 0.12]. For the CAS example the Pearson Correlation coefficient is $r_{CAS} = -0.871$.

The six figures showing the results for mutation score and root mean squared error also contain a trendline each which was created by linear regression as explained in [61]. This trendline supports the visualization of the Pearson Correlation coefficient.

5.3. Evaluation

We interpret the Pearson Correlation coefficients obtained from our results as explained in Table 5.1. For the examples TCAS, Triangle, BMI, and UTF8 the correlation is weak. The correlation for the POP3 example is strong and for the CAS example the correlation is very strong. The optimal Pearson correlation coefficient, which could indicate that we can use model inference

based adequacy assessment instead of mutation score is -1. This follows from the assumption that if the mutation score of a test suite is high, the root mean squared error is low and vice versa.
The examples resulting in a weak correlation indicate that mutation score could not be replaced by model inference based adequacy assessment. To verify this claim we ran two additional experiments for TCAS, Triangle, BMI, and UTF8. First we changed the size ratio of T_{dt} and T_{equal} from 1:9 to 1:4, 3:7, 2:3, and 1:1. Second we divided the mutants into categories as explained in Section 4.1.1 and analyzed whether a correlation between model inference based quality assessment and the mutation score of certain categories of mutants exists.
Changing the sizes of T_{dt} and T_{equal} did not affect the Pearson correlation coefficient significantly. We were able to categorize the mutants into the six categories: AOR, LVR, ROR, COR, SOR, and LOR. To obtain the Pearson Correlation coefficient we created again 100 test suites T_{dt} for each mutant category, for each example, and processed mutation score and root mean squared error.

For the four examples TCAS, Triangle, BMI, and UTF8 no sign of linear correlation was obtained for any category of mutants. The examples POP3 and CAS result in a strong and very strong correlation respectively. This originates from the facts that POP3 and CAS use enumerated inputs and that the test sequences from which the models are inferred contain additionally to the inputs and outputs also information about the current state and the preceding output. As shown in Table 5.3 the learned decision trees for POP3 and CAS have at most two edges from the root to any leaf node. For all other examples the inputs are continuous values which ranges can be split multiple times while growing a decision tree. The algorithm to learn the decision tree is essential for the obtained results. For examples with continuous types we obtain binary decision trees which split value ranges. For the examples with enumerated input types we obtain decision trees with an outgoing edge from a node for each possible value the variable represented by the node can be assigned to. The sizes of the control flow graphs for POP3 and CAS are highest, but to obtain similar coverage results for all examples also the number of test vectors is highest for these two examples.

5.4. Related Work

In their recent work Fraser and Walkinshaw [56] answer the question whether there exists a relationship between their adequacy score and the ability of a test suite to detect defects. The authors also use the term adequacy, which was defined in [62] where a test suite is adequate if it implies no errors in the SUT if it executes correctly. In [56] the authors do not obtain a positive correlation at all for their examples with categorical outcomes when using a similar machine learning based test suite adequacy assessment. The approach introduced in this work to assess the adequacy of a test suite by model inference is similar to the idea of probably approximately correct (PAC) learning [63]. PAC provides a theoretical framework for evaluating model accuracy.
The same method of learning a decision tree from a test suite using the C4.5 algorithm is applied by Papadopoulos and Walkinshaw in [64]. In [64] the authors use the decision tree as input model for test case generation to extend an existing test suite.
In [65] the authors demonstrate empirically that there is a low to moderate correlation between coverage and effectiveness of a test suite. To assess effectiveness they also use the mutation score of a test suite. The authors calculate two different correlation coefficients. First they calculate the Pearson Correlation coefficient as we did in this work. Second they calculate the Kendall Correlation coefficient [66], which yields similar results as obtained by the Pearson Correlation coefficient.
Just et al. investigate in [67] whether mutants are a valid substitute for real faults. Their results show a statistically significant correlation between mutant detection and real fault detection, independent of coverage.

In [68] the author introduces a definition of test data adequacy. She points out that if the behavior of a system can be inferred from a test suite, it can be concluded, that the SUT's behavior is adequately tested. She also describes practical limitations, which origin in the fact that equivalence is not decidable. In [69] Zhu et al. review the theories of inductive inference and their relevance to software testing.

5.5. Summary

We conclude that there can be a linear correlation between mutation score and model inference based test suite adequacy assessment. However, this relationship depends on the data within a test suite and the input types of the SUT. In contrast to related work we added information about the current state and the previous output of the SUT to each test case in a test suite for certain examples, that resulted in a clear linear correlation.
The examples resulting in a weak positive correlation, are the examples with continuous input types and without information about internal states of an SUT within a test suite. The types of the inputs and the information about internal states within the test suite seem to be the essential reasons why for some examples a linear correlation exists and for some not. We address the open research on the detailed differences of the examples and the test suites in future work.

Generally we obtained results for the root mean squared error of an inferred model with ranges from 0.11 up to 0.4. These results, and the related work, clearly show us, that it is possible to assess the adequacy of a test suite by model inference. In future work we will prepare some real world examples and also investigate whether there exist linear correlations between model inference based test suite adequacy assessment and various coverage assessment methods. Further we will apply different machine learning methods to infer models from a test suite and adopt other similarity measures to assess the approximations of the inferred model and the SUT. From these improvements we will get better insights when to use a model inference based method, which only depends on the inputs and outputs of a program, and when to use test suite adequacy assessment methods, which depend on source code modifications.

6. Classifying Test Suite Effectiveness via Model Inference and ROBBDs

This chapter is based on the work "Classifying Test Suite Effectiveness via Model Inference and ROBBDs" [7].

Test suite generation and in turn the decision whether a test suite is good enough are certainly quite complex tasks. In this chapter, we focus on the latter and in particular on the functional aspect. That is, we would like to know whether a given test suite examines the functionality of our program to the best of our knowledge. Certainly, a test suite examining all possible input combinations would be an assuring approach, but exhaustive testing of the entire I/O space is certainly impossible in most cases due to the sheer number of required tests.

Now, let us consider the example of a Boolean function as illustrated in Figure 6.1. The function has $n = 3$ input variables, and let us assume that we have an exhaustive test suite TS containing 2^n test cases s.t. all input combinations are tested. While TS certainly is effective due to checking the entire I/O space, for a subset $T \subseteq TS$, T's effectiveness is unclear. If we assume there to be, e.g., 40 inputs, exhaustive testing with 2^{40} test cases is however certainly impossible. Our idea now is to learn a canonical representation from T that we then compare to a corresponding representation derived from the program under test's specification, in order to check whether they are equivalent. In case of equivalence, we then assume that the test suite captures the same behavior as the program. This leads us to the immediate and important question: What is an attractive canonical representation suitable for a test suite *and* the tested program?

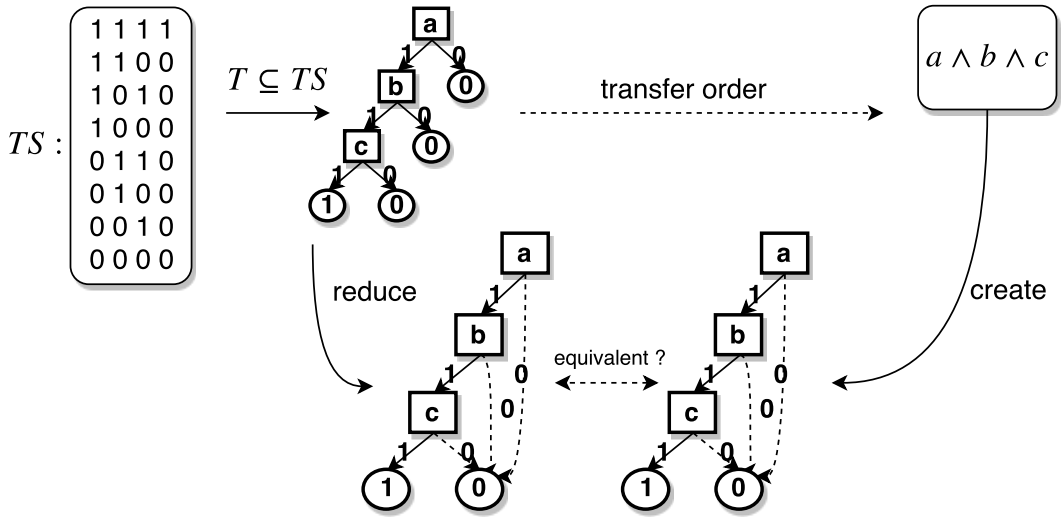

Figure 6.1.: An illustration of our approach for a simple Boolean function.

As described in Section 6.2, we use reduced ordered binary decision diagrams (ROBDDs) as our canonical format. In particular, like illustrated in Figure 6.1, we infer a binary decision tree *DT* from the considered test suite *T* via machine learning. In the learning algorithm proposed in Section 6.2.1, similar to the algorithm C4.5 (see Section 6.1) we take the local information gain into account when choosing the next decision variable. Consequently, the variable order in *DT*'s individual paths is determined via the entropy of the local situations in order to come up with a "good" order. Our next step is to derive an ordering graph *O* for *DT* as described in Section 6.2.2, and if it is cycle-free, we extract a total variable order and reduce *DT* to an ROBDD (for the latter see Section 6.2.3). Afterwards we derive an ROBDD for the system under test (SUT) from its Boolean functions' specification with the concept described in Section 6.2.4.

Our argument is that if the two ROBDDs coincide, then a program satisfying *T* should implement the desired I/O behavior. Our experiments reported in Section 6.3 show an excellent correlation between our classification and fault identification capabilities. Note that if *O* features cycles, then there are contradicting variable orders in *DT*'s paths (see Sec. 6.2.2) that we would need to resolve by reordering the variables in some paths (this is not implemented yet).

Table 6.1.: A table reporting on the performance of several test suites $T \subseteq TS$ for the example shown in Figure 6.1.

$\|T\|$	# subsets T	eff. T	ratio [%]
8	1	1	100.00
7	8	4	50.00
6	28	7	25.00
5	56	5	8.93
4	70	1	1.43
3	56	0	0.00

Now let us come back to our example. If we consider Table 6.1, we can see, e.g., that for a subset size of 4 $(= n + 1)$, out of the $\binom{2^n}{n+1} = 70$ subsets, there is a single one that we would consider to be effective, and that the likelihood of achieving this with some $T \subseteq TS$ increases with a higher subset size. While our approach cannot yet be used to derive *missing* test cases, it can serve for adding a quality label to some existing test suite T.

When using only a subset of (n+1) test cases of the test suite to infer a model and classifying the efficiency of each of the $\binom{2^n}{n+1} = 70$ different subsets of size (n+1), only one subset can be classified as an efficient test suite. From this efficient test suite we infer a model from which we extract a variable order that is transferred to the creation of the ROBDD from the Boolean function. Only for the efficient test suite the created ROBDD and the inferred model, reduced to an ROBDD, are equivalent. For different subset sizes we obtained the results as shown in Table 6.1.

6.1. Preliminaries

We use Boolean functions as abstract representation format for a program (or also a combinatorial circuit) in our approach. Static Single Assignment form as discussed in [70] and digitizing non-Boolean variables (via predicates) would help to use our work also for more complex programs.

An n-ary **Boolean function** $f : \mathbb{B}^n \to \mathbb{B}$ with n inputs maps n Boolean ($\mathbb{B} = \{0, 1\}$ represents the Boolean values *False* and *True* as usual) input

values to a single Boolean output value. A test case t in a test suite T thus is a vector $(x_1, .., x_{n+1})$ defining n input values and f's expected output when executing t. For our implementation, we consider the following unary and binary standard operators in a Boolean function: $\neg$ (not), $\wedge$ (and), $\vee$ (or), and $\oplus$ (xor).

An essential ingredient of our approach is that we **learn a decision tree** from a test suite. As depicted in more detail in Section 6.2.1, we use an altered version of the widely used **C4.5 algorithm** [51] for this. The algorithm computes the *entropy* and *information gain* when selecting nodes/variables while growing the tree. So let us first define these two terms.

In information theory, entropy is commonly used as a measure of purity or impurity of an arbitrary set of examples, and we would like to choose an optimal next decision variable in this respect. In our case, for a test suite T, we take a Boolean output variable (if there are more, we have to choose one) and can derive the entropy $E(T)$ with respect to this variable according to Equation 6.1. In this equation, p_t represents the proportion of test cases s.t. the considered output variable's value is *True*, and p_f gives the same proportion for *False* ($p_t + p_f = 1$). If all test cases result in the same output value, then entropy is 0, while an entropy of 1 indicates $p_t = p_f = 0.5$. In general, we have $0 \leq E(T) \leq 1$.

$$E(T) \equiv -p_t \log_2 p_t - p_f \log_2 p_f. \tag{6.1}$$

Due to base 2, entropy is a measure for the expected encoding length in bits. That is, entropy can be interpreted as the minimum number of bits needed to encode the classification of an arbitrary test case from a test suite. If not an integer, the value represents the average number of required bits.

When growing the tree, the variables for nodes $n \in N$ are selected by investigating the effectiveness of the various input variables in classifying the test cases. Informally, the *information gain* of a variable reports the expected reduction in entropy caused by partitioning the test suite according to this variable. Formally, the information gain $Gain(T, v)$ of variable v for a set of test cases T is computed as of Equation 6.2, where $Values(v)$ returns the domain of variable v (which is $\mathbb{B}$ in our case) and $T_b \subseteq T$ denotes T's subset s.t. v has value $b \in Values(v)$. Coming back to the learning algorithm, it selects a variable with the highest information gain. If there is no variable

v s.t. $Gain(T, v) > 0$, the current node becomes a leaf labeled by $b \in \mathbb{B}$ s.t. b is the most probable classification of the inputs leading to this leaf. Consequently, there can still be misclassifications.

$$Gain(T, v) \equiv E(T) - \sum_{b \in Values(v)} \frac{|T_b|}{|T|} E(T_b). \tag{6.2}$$

We use special **binary decision diagrams (BDDs)** [71] as canonical format for our comparison. A BDD describes a Boolean function f via a rooted, directed acyclic graph. In particular, the BDD consists of nodes N for representing the consideration of a Boolean variable, two nodes L representing f's two possible outcomes *False* and *True*, and directed edges E connecting the nodes as follows. Each node $n \in N$ is labeled by a variable $var(n)$ and has *a pair* of outgoing edges $e_0(n)$ and $e_1(n)$ leading to a child node for the corresponding evaluation (the i of $e_i(n)$) of variable $var(n)$. Nodes in L have no outgoing edge.

A canonical variant of BDDs are reduced ordered binary decision diagrams (ROBDD)s [72]. A specific feature of an ROBDD is that it implements a certain variable order in the graph, i.e., there is a total order s.t. for $n, n' \in N$ we have that if $v_1 = var(n)$ appears before $v_2 = var(n')$ in some path, then this is the case for all paths Π^k in the BDD (the BDD is an OBDD then) s.t. both variables appear in them. Furthermore, an ROBDD is reduced, which means that each node in the OBDD represents a different Boolean function. Due to the resulting canonicity of an ROBBD, we can easily decide **equivalence**:

Definition 8 *Two ROBDDs A and B are* equivalent *iff their root nodes are equivalent. Two nodes $n_1 \in A$ and $n_2 \in B$ are equivalent if they are either*

1. *leaf nodes with the same label, or*
2. *non-leaf nodes that have the same label and the same pairs of outgoing edges $(e_0(n_1), e_0(n_2))$ and $(e_1(n_1), e_1(n_2))$ lead to equivalent nodes respectively.*

Determining two ROBDDs' equivalence is linear in the size of the smaller (if not equivalent) ROBDD where its size is given by the number of its nodes.

6.1.1. Reduced Ordered Binary Decision Diagrams

As explained in [71], binary decision diagrams (BDD)s represent Boolean functions as rooted, directed acyclic graphs. A BDD consists of nodes N representing Boolean variables, two nodes L representing the possible outcomes of the Boolean functions, and edges E connecting the nodes. Each node $n \in N$ is labeled by a variable name and has a pair of outgoing edges $e0(n)$ and $e1(n)$ each directing to a child node where $e0(n)$ is labeled 0 and $e1(n)$ is labeled 1 for the valuation of the variable represented by n.
In this work we use reduced ordered binary decision diagrams (ROBDD)s [72]. An ROBDD follows a certain order of the nodes in the graph. Within the graph exists a total ordering $v_1 \in V < v_2 \in V$ over the set of variables V of the Boolean function represented by the root node. A total order ensures that for each path from the root node to a leaf node a certain order is kept. These graphs are canonical representations of the respective Boolean functions. The order which variable to select while growing the tree is transferred from the inferred decision tree as shown in Figure 6.2 in this work.

6.2. Classifying Test Suite Effectiveness

Via comparing the canonical representations of the test suite and the program under test, we aim to decide whether a test suite is effective enough. The basic steps of our approach at achieving this are illustrated in Figure 6.2.

Our initial step is to learn a decision tree *DT* from the considered test suite *T*, where we report on the details of this step in Subsection 6.2.1. An important prerequisite here is that the test cases in *T* are unique, since duplicates affect the information gain as computed for selecting the next decision variable in a tree. Like we mentioned in the introduction, furthermore the output variables' values have to be deterministic for the input values (as provided by some $t \in T$).

As depicted in Subsection 6.2.2, we then investigate the variable orders in the individual paths in *DT* and derive a total variable order if possible, i.e.,

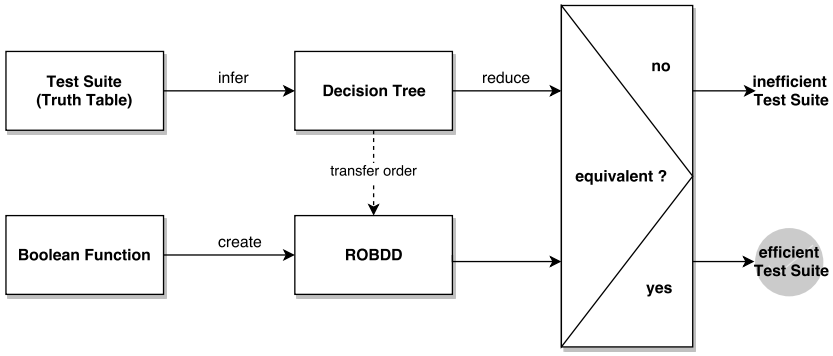

Figure 6.2.: Process of our test suite classification approach.

if *DT* is ordered. If there is such a total variable order Ψ, we reduce *DT* to an *ROBDD* as described in Subsection 6.2.3. Furthermore, we derive an ROBDD for the program or system under test as depicted in Subsection 6.2.4, using the same order Ψ.

In our last step, we finally compare the two resulting ROBDDs according to Definition 8, using the recursive function `isEqual` given in Algorithm 3.

Algorithm 3 Function to decide whether two ROBDDs are equivalent.

```
1: function isEqual(Node n1, Node n2)
2:     if isLeaf(n1) and isLeaf(n2) and label(n1) == label(n2) then
3:         return True
4:     else if isLeaf(n1) or isLeaf(n2) then
5:         return False
6:     else
7:         return label(n1) == label(n2) and
            isEqual(true(n1), true(n2)) and
            isEqual(false(n1), false(n2))
8:     end if
9: end function
```

6.2.1. Learning a Decision Tree from a Test Suite

Aiming at deriving a representative ROBDD for T, we certainly would like to avoid any misclassification of a test as it could occur with C4.5 (see Section 6.1). To this end, we extended the algorithm as shown in Algorithm 4.

Algorithm 4 Function to learn a decision tree from a test suite T.

```
1:  function ADDNODE(T, V, N)
2:      v, g ← MAX_IG(T, V, E(T))
3:      if g > 0 then
4:          N.v ← v
5:          N.true ← ADDNODE(T_{v_t} ⊆ T, V' ← V \ {v}, N' ← new_node())
6:          N.false ← ADDNODE(T_{v_f} ⊆ T, V', N' ← new_node())
7:      else
8:          if E(T) > 0 then
9:              v ← V.head
10:             N.v ← v
11:             N.true ← ADDNODE(T_{v_t} ⊆ T, V' ← V \ {v}, N' ← new_node())
12:             N.false ← ADDNODE(T_{v_f} ⊆ T, V', N' ← new_node())
13:         else
14:             N.name ← OUT(T)
15:         end if
16:     end if
17:     return N
18: end function
```

Our recursive function ADDNODE takes as input a test suite T, a set of variables V, and a node N, and returns the root node of a binary decision tree. Initially, the set V contains n Boolean input variables of the tested Boolean function. Node N is a new and empty node in DT, where we assign N's properties inside ADDNODE. These properties are $N.v$, and two successor nodes $N.true$ and $N.false$. As stated in the preliminaries, there are two types of nodes. First, decision nodes which represent a test of some input variable $v \in V$ (s.t. $N.v = v$) which divides the local T' into two subsets T'_{v_t} and T'_{v_f} according to the value of variable v. Correspondingly, each such decision

node has exactly two successor nodes. Then there are leaf nodes, reporting the output's value of the test cases classified to this node (thus $N.v \in \mathbb{B}$). A leaf node does not have any successor nodes.

In line 2, the function `MAX_IG` returns a variable v and its information gain g s.t. g is the maximum gain of all $v \in V$ (obviously there could be more than one such v). If there is such a variable with $g > 0$, we select it and recursively update $N.v$ and the *true* and *false* successor nodes. The two recursive calls of `ADDNODE` use as input $T_{v_t} \subseteq T$ s.t. v is *True* (or the complement T_{v_f} for *False*), the variable set $V' \leftarrow V \setminus \{v\}$ s.t. v was removed from V, and a fresh new node N'.

If there is no variable with $g > 0$, our adaption takes control. That is, if there are still different output values for T indicated by entropy $E(T) > 0$, we select the first variable from V ($V.head$) as decision variable and proceed like for a decision variable selected as of above (with corresponding recursive calls for the successor nodes). If all expected outcomes for the tests in T coincide ($E(T) = 0$), N is a leaf node to be assigned this expected outcome $\texttt{OUT}(T)$.

Note that we assume the test suites to be deterministic. Thus Algorithm 4 derived from C4.5 is guaranteed to terminate without misclassifications.

6.2.2. Isolating a Total Variable Order from *DT*

Two crucial questions for our approach are whether there is a total variable order fitting the learned decision tree *DT*, and in turn how to extract such an order so as to use it for generating an ROBDD for the SUT. For tackling these questions, we start by constructing an *ordering graph* as of Def. 10. But let us first define a path Π in *DT* and its variable sequence $\Phi = var(\Pi)$, as well as *DT*'s alphabet.

Definition 9 *A* path Π *of length* $|\Pi| = n$ *in a decision tree DT is a sequence of nodes* $\pi_0 ... \pi_{n-1}$ *such that there is an edge from* π_i *to* π_{i+1} *(*π_i *is parent of* π_{i+1}*) for* $0 \leq i < n-2$*. The path of some node s in DT is the node sequence from DT's root node r to s.* $\Phi = var(\Pi)$ *is the sequence of variables* ϕ_i *considered at the individual nodes* π_i *in* Π *s.t.* $\phi_i = var(\pi_i)$*. For the leaves, that per definition have*

no variable *label but are labeled either False or True, we have* $\epsilon = var(\pi_i)$ *s.t. in this case we have* $|\Phi| = |\Pi| - 1$. *The* alphabet $\Sigma = alphabet(DT)$ *is the union of the variables considered at the individual nodes in DT.*

Definition 10 *An* ordering graph *O for a decision tree DT is a directed graph represented by the tuple* $(Q, q_0, T \subseteq Q \times Q, \Sigma, l : Q \to \Sigma, \lambda : \Sigma \to Q)$ *such that*

- $\Sigma = alphabet(DT)$ *is a finite alphabet inherited from DT*
- Q *is a finite set of nodes, where* $|Q| = |\Sigma|$ *and for each* $\sigma \in \Sigma$ *there is some* $q_\sigma \in Q$ *s.t.* $l(q_\sigma) = \sigma$
- q_0 *is the root node, where* $l(q_0) = var(r)$ *such that r is DT's root node*
- T *is the transition relation, where* $(q_\sigma, q_\delta) \in T$ *iff DT features two nodes s and d s.t. (1) s is a parent of d, and (2)* $l(q_\sigma) = var(s)$ *and* $l(q_\delta) = var(d)$.
- l *is a labeling function that assigns each* $q \in Q$ *some* $\sigma \in \Sigma$.
- λ *is a function that returns for some* $\sigma \in \Sigma$ *a* $q_\sigma \in Q$ *such that* $l(q_\sigma) = \sigma$.

Corollary 1 *Due to* $|Q| = |\Sigma|$ *and the definition of Q, we have that for every* $\sigma \in \Sigma$ *there is a* unique $q \in Q$ *such that* $l(q) = \sigma$.

When deriving an ordering graph for a given decision tree, the construction of Q, q_0, l and λ is straightforward, and for T we can use Algorithm 5. This algorithm's idea is to traverse the whole tree from the leaves towards the root, and whenever we end up at the root or some state that we have visited before, we proceed with the next leaf. For each node visited in this process, we add its *incoming* edge to T, so that we end up with the edge collection required by Definition 10. It is easy to see that the algorithm terminates and that its run-time is linear in the amount of nodes in DT.

Via the following Theorem 1, we can then decide whether there is a total variable order that aggregates the individual partial orders as defined via $var(\Pi)$ by the available paths Π in DT. For our investigations it will suffice to focus on the leaves' paths since all other paths are contained in them.

Theorem 1 *For a decision tree DT with alphabet* Σ *there exists a total variable order that does not contradict any path from the root to some leaf if and only if there is no cycle in the corresponding ordering graph O as of Definition 10.*

Algorithm 5 Function to create T of an ordering graph for a binary tree DT.

```
1:  function createT(DT)
2:      unmark(DT.nodes)
3:      T ← ∅
4:      L ←list of DT's leaves, e.g., sorted according to depth.
5:      while |L| > 0 do
6:          node ← L.pop()
7:          node ← node′ = parent of node // skip leaf (has no variable label)
8:          next ← False
9:          while (!next) do
10:             if marked(node) then // proc. w. next leaf
11:                 next ← True
12:             else if node has parent node′ then
13:                 T ← T ∪ {(node.variable, node′.variable)}
14:                 mark(node)
15:                 node ← node′
16:             else
17:                 mark(node)
18:             end if
19:         end while
20:     end while
21:     return T
22: end function
```

Proof 1 *(sketch) A basic observation about ordering graphs is that for any edge in DT from s to d s.t. var* $(s) = \sigma \in \Sigma$ *and var* $(d) = \delta \in \Sigma$*, there is also a directed edge from* q_σ *to* q_δ *in O s.t.* $l(q_\sigma) = \sigma$ *and* $l(q_\delta) = \delta$*. It is important to note that, according to Def. 10, O does not contain any other edges.*

Now, let us assume that the partial variable orders as defined by the individual paths in DT contradict each other. That is, there are some variables σ *and* δ *in* Σ *s.t. in some path* Π^1 *we have* $i < j$ *for* $\sigma = var\,({\pi^1}_i)$ *and* $\delta = var\,({\pi^1}_j)$*, but there is also some path* Π^2 *s.t.* $j < i$ *for* $\sigma = var\,({\pi^2}_i)$ *and* $\delta = var\,({\pi^2}_j)$*. Since for any such edge in DT , there is a corresponding directed edge in O, this means that* $q_\delta = \lambda(\delta)$ *is reachable from* $q_\sigma = \lambda(\sigma)$ *via the node sequence* $\lambda(var\,({\pi^1}_i)) \ldots \lambda(var\,({\pi^1}_j))$*, while* q_σ *is reachable from* q_δ *via the sequence implied by* Π^2 *(i.e.,* $\lambda(var\,({\pi^2}_j)) \ldots \lambda(var\,({\pi^2}_i))$*). Consequently, O has a cycle.*

Now let us assume that for any two variables σ *and* δ *in* Σ *and the paths* Π^k *that consider* both *variables, we have for* $\sigma = var\,({\pi^k}_i)$ *and* $\delta = var\,({\pi^k}_j)$ *that either* $i < j$ *for all k or* $j < i$ *for all k. In other words, there are no contradictions between the individual partial orders of DT 's paths. Without losing generality let us assume* $i < j$*. Consequently,* q_δ *is reachable from* q_σ *via any sequence implied by some path* Π^k *in DT that features* σ *and* δ*. However,* q_σ *is unreachable from* q_δ*. That is, due to our observation on O's edges, this would require the presence of a path in DT where we would consider* q_δ *before* q_σ*, but which contradicts our assumption. With the definition of Q, we thus cannot have a cycle in O, so that O is indeed a directed acyclic graph (DAG).*

For identifying cycles in a directed graph, we can use, e.g, the STRONGCONNECT algorithm depicted in [73].

If the ordering graph is indeed a DAG, we can retrieve *some* total order Ψ (as a sequence of $\psi_i \in \Sigma$) via the topological sorting algorithm variant `CREATEORDER` given as Algorithm 6. The underlying idea is as follows: It is easy to see that we have that for every $\sigma \in \Sigma$, the source nodes q_α of q_σ's incoming edges define the complete set of variables $\alpha \in \Sigma$ that are considered *right* before σ for some path in DT. Consequently these variables also have to appear before σ in a total order, and we have to consider this property in a recursive way (reasoning again from α). For easy access, we thus store T's edges in a fashion that every $q \in Q$ has a list of parents $q.inlist$

and a list of children $q.outlist$. Now, if we traverse O starting with q_0, and follow the outgoing edges of some q_σ only whenever all of q_σ's incoming edges have been followed, we can establish a total order by appending $\sigma \in \Sigma$ to the order (once) whenever the outgoing edges of q_σ "become available". If O is indeed a DAG, we can do so in a breadth-first manner (for a cycle this obviously would not work).

Algorithm 6 Function to derive a total order Ψ of an acyclic ordering graph O.

```
 1: function createOrder(O)
 2:     assert isDAG(O)
 3:     Ω ← ∅
 4:     Ψ ← [l(q_0)]
 5:     for all q_i ∈ q_0.outlist do
 6:         remove q_0 from q_i.inlist
 7:         Ω ← Ω ∪ {q_i}
 8:     end for
 9:     while |Ω| > 0 do
10:         pop q_σ from Ω such that |q_σ.inlist| = 0
11:         for all q_i ∈ q_σ.outlist do
12:             remove q_σ from q_i.inlist
13:             if q_i ∉ Ω then
14:                 Ω ← Ω ∪ {q_i}
15:             end if
16:         end for
17:         Ψ.append(l(q_σ))
18:     end while
19:     return Ψ
20: end function
```

In our algorithm, thus, whenever we add a variable δ to Ψ (line 17), we *remove* the obligation of δ having to appear prior to σ for all σ s.t. $\lambda(\sigma)$ is a child of $\lambda(\delta)$ in line 12 (or 6 for q_0's children). Obviously, whenever all obligations have been met for some σ s.t. $|\lambda(\sigma).inlist| = 0$, we can select σ (line 10) and append it to Ψ (line 17). Since there is no reason to search for such a σ in the whole Σ, we keep a worklist Ω containing those $\lambda(\sigma)$ for which some of σ's obligations already have been met (and have been

removed from $\lambda(\sigma).inlist$). This worklist is filled with nodes in two ways. That is, the first node without any obligation is obviously the root note, so that we have $l(q_0)$ as first item of Ψ (line 3) and initially fill Ω with q_0's children (lines 5-7) after treating them as described above (line 6). The second option is that, whenever we remove in line 11 an obligation from $\lambda(\sigma)$ when considering some $\delta \neq l(q_0)$ s.t. $\lambda(\delta)$ is $\lambda(\sigma)$'s parent, we search whether the node is already in Ω and add it if this is not the case (lines 13-15).

Now let us show that this algorithm is complete and sound. That is (1) that it can always derive a sequence Ψ (a total order) from some acyclic ordering graph O, and (b) that a derived sequence Ψ is indeed a total order for DT.

Theorem 2 *For some acyclic ordering graph O, Algorithm 6 terminates and derives a sequence Ψ of variables $\psi_i \in \Sigma$.*

Proof 2 *(sketch) Due to line 4 and the fact that we only append $\sigma \in \Sigma$ ($l : Q \to \Sigma$) to Ψ in line 17, this leaves us to show that the algorithm terminates correctly if O is a DAG. Given the algorithm's structure, the crucial lines in this respect are lines 9 and 10. That is, line 2 is no problem if O is a DAG, but we have to show that Ω becomes empty at some point such that we leave the while loop, and that we can indeed pop some q_σ in line 10 for avoiding a deadlock.*

Let us start with the latter s.t. Ω is non-empty. Now let us assume that Ω contains some q_σ s.t. $|q_\sigma.inlist| > 0$ with $q_\delta \in q_\sigma.inlist$. Due to T and lines 10 to 15, this means that either q_δ is in Ω, or one of its ancestors $\lambda(var(\pi_i^k))$ in path Π^k in DT s.t. $i < j$ for $\delta = l(\pi_j^k)$. That is, since we initialized Ω with all children of q_0, due to lines 10 to 15 the only way for none of them to be in Ω would be that all obligations of q_δ were already fulfilled and q_δ was already chosen in line 10 and added to Ψ. But then q_δ would not be in $q_\sigma.inlist$ (see also the argument below). Assuming $\lambda(var(\pi_i^k))$ to be in Ω then means that either we could choose this node if its inlist *is empty, or via its inlist we could again find some node in Ω as described above. Since O is acyclic and $|Q| = |\Sigma|$ is finite, the number of times we have to do this until finding some node in Ω s.t. its inlist is empty is limited. It directly follows that if $|\Omega| > 0$, then there is also always a node q_σ in Ω s.t. $|q_\sigma.inlist| = 0$ that we can choose in line 10.*

Now let us show that Ω becomes empty eventually, which directly follows from the fact that $|Q|$ is finite and that we have that some $q_\sigma \in Q$ is added to Ω only once. That is, if q_0 is a parent of q_σ, then we add it to Ω in line 7. If not, then, when the first of q_σ's parents is chosen in line 10, we add q_σ to Ω in line 14. When in Ω, q_σ is not added a second time due to line 13. After q_σ was chosen in line 10 (and consequently removed from Ω) it will never be added again. That is, all of σ's obligations regarding variables that have to appear prior to σ have been met s.t. $\lambda(\sigma).inlist$ became empty–there is no further incoming edge that has not been considered so far and could add q_σ to Ω via lines 11 to 14. Since $|Q|$ is finite and there is always a node to choose in line 10, thus Ω becomes empty at some point s.t. the algorithm terminates successfully if O is acyclic.

Corollary 2 *Each $\sigma \in \Sigma$ appears exactly once in Ψ as of Theorem 2.*

The validity of the corollary is easy to see via Corollary 1, the definitions of T and Q, and the fact that some $q \in Q$ is added and removed from Ω (s.t. q's label is appended to Ψ) exactly once (for the latter see the proof of Theorem 2). That is, every q aside q_0 (but whose label is initially appended to Ψ in line 4) has some incoming edge(s) and if O is a DAG it is finally added to Ω (lines 7 or 14) when the first of its parents is selected (lines 5 or 10) as well as finally selected itself in line 10 (s.t. $l(q)$ is appended to Ψ in line 17) since the algorithm terminates (for both see the proof of Theorem 2). From Corollary 1 it then directly follows that each $\sigma \in \Sigma$ appears exactly once in Ψ.

Theorem 3 *The sequence Ψ returned by the algorithm as of Fig. 6 for some acyclic ordering graph O is a* total order. *This means that for every path Π in O's DT we have for any $0 \leq i, j < |\Pi|$ s.t. $i \neq j$ that $k < m$ if $i < j$ or $k > m$ if $i > j$ for $\psi_k = \lambda(\pi_i)$ and $\psi_m = \lambda(\pi_j)$.*

Proof 3 *(sketch) This directly follows from Corollary 2 and the proof of Theorem 2. That is, since q_σ is removed from Ω and appended to Σ only if all the obligations about variables that have to be present in Ω prior to σ (as encoded "recursively" in $\lambda(\sigma).inlist$), it is ensured that there is no path in DT that has some variable α being considered before σ s.t. α is not in Ψ when adding σ. Due to Corollary 2, we furthermore have that every $\sigma \in \Sigma$ is present in Ψ and appears exactly once. Thus Ψ is a total order as desired.*

6.2.3. Reducing the Learned Decision Tree DT to an ROBDD

If we successfully retrieved a total variable order from DT, we use an algorithm depicted by Bryant in [74] to reduce our *ordered* decision tree to an ROBDD. The idea behind his algorithm is to implement the following three rules [72]:

1. **Remove duplicate leaf nodes:** Eliminate all but one DT's leaf nodes with the same label and redirect all edges from the eliminated nodes to the corresponding remaining equivalent one.
2. **Remove duplicate nodes representing variables:** If two nodes $n_1, n_2 \in N$ have the same label, the outgoing edges $e_0(n_1)$ and $e_0(n_2)$ point to equivalent nodes, and also the outgoing edges $e_1(n_1)$ and $e_1(n_2)$ point to equivalent nodes, then eliminate n_1 and redirect all its incoming edges to n_2.
3. **Remove redundant tests:** If a node n's outgoing edges $e_0(n)$ and $e_1(n)$ lead to the same node n', eliminate n and redirect its incoming edges to n'.

Figure 6.3 illustrates the application of these three reduction rules on an example decision tree with three variables. From left to right, we first applied rule 1, such that we only have two leaf nodes, one for each $b \in \mathbb{B}$. Then we merge redundant nodes (rule 2) leaving us with only two nodes labeled with x_3 instead of four. In the last step, we remove two nodes with redundant / meaningless tests for x_2 and x_3 arriving at the ROBDD on the right. If none of the three reduction rules is applicable anymore, then the result is an ROBDD.

6.2.4. Creating an ROBDD for the SUT's specification

For creating an ROBDD from a Boolean function f, we use the algorithm presented in [75]. The underlying scheme is based on the *if-then-else normal form (INF)*, where a Boolean function is built entirely via the if-then-else operator, e.g., the if-then-else operator $x \rightarrow y_0, y_1$ is defined by $x \rightarrow y_0, y_1 = (x \wedge y_0) \vee (\neg x \wedge y_1)$. From f's INF we create an ROBDD by

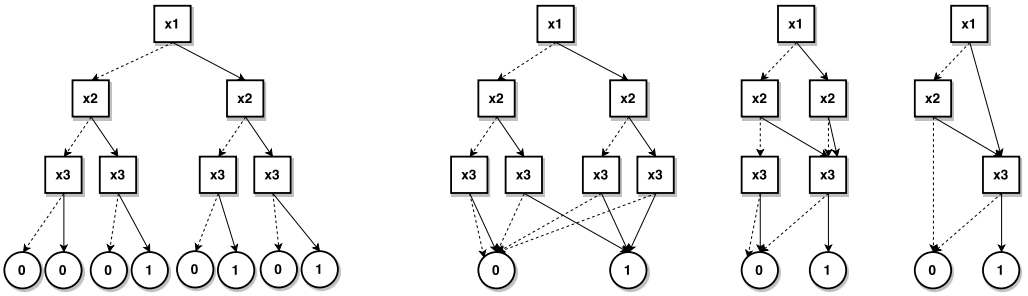

Figure 6.3.: Reduction of ordered decision tree to ROBDD, from left to right [72].

applying Shannon expansion [76], where the total variable order Ψ obtained from DT is used to replace the variables by constants in $\mathbb{B}$. Please note that any variable v present in T but not appearing in DT or Ψ (v is of no consequence) should be appended to Ψ for this construction (but should be absent from the ROBDD in the end).

When replacing the variables by Boolean constants, the Shannon cofactors emerge. Each cofactor can be viewed as an outgoing edge of a node in the ROBDD where the replaced variable represents the node. While creating the ROBDD, equivalent nodes are replaced such that after finalizing the ROBDD none of the reduction rules introduced in Section 6.2.3 can be applied.

Since the size of the ROBDD depends heavily on the variable order used, and finding a variable order that minimizes an ROBDD's size is a co-NP-complete problem [74], we extract a variable order from DT via entropy-based learning, rather than trying to come up with an ideal order for the SUT's ROBDD to be used then also for the test suite T's ROBDD.

6.3. Experimental Results

For evaluating our approach, we investigated its performance for 20 examples taken from [77] representing formal specifications (as shown in Figure 6.4) for TCAS II, an aircraft collision avoidance system. For these 20 Boolean specifications with 5 to 14 Boolean input variables, we generated

corresponding Boolean functions and exhaustive test suites featuring all possible input combinations.

The results reported in Table 6.3 were obtained by generating 100 different random test suites T s.t. we could classify T for each example. Since we cannot yet classify T if its ordering graph has some cycle, we sometimes had to create more than the 100, where the corresponding number of discarded ones is given in Table 6.2 (there were next to none classifiable duplicates that we had to discard). Each T was derived by randomly selecting test cases t from an example's exhaustive test suite s.t. $t \in T$ with a probability of 0.5.

For each of an example's 100 test suites, we calculated the mutation score, i.e., the proportion of mutants that T was able to *kill*. If some T showed different behavior for a mutant f' compared to f, then T was able to kill the mutant. For generating the mutants f' we replaced f's binary operators with alternatives, where the number of resulting mutants ($\#f'$) is given in Table 6.2.

Considering our classification into effective and ineffective test suites (sets T_+ and T_-), and comparing it to the maximum mutation score (1.0 for all examples), we would get only a few "false" positives ($MS(T \in T_+)$) like for example 8, where one of the two $T \in T_+$ killed only 14 out of 16 mutants (we found those two remaining mutants to be equivalent after closer inspection). For the total 37 Ts classified to be effective for some example, this means that only for one T the corresponding $MS(T)$ was less than the maximum achievable mutation score. Since T_+ was underrepresented in the random test suites (as we expected), we generated for each of the six examples 1/6/9/10/15/20 another 100 test suites s.t. we could classify 50 as effective

Table 6.2.: Number of detected cycles while creating 100 random test suites and the number of mutants $\#f'$ for each example.

example	1	2	3	4	5	6	7	8	9	10
cycles	155	197	444289	0	18	72	898	6	0	144
$\#f'$	22	34	45	4	19	27	20	16	9	14
example	11	12	13	14	15	16	17	18	19	20
cycles	75391	340002	192	43	47	641685	8588	2216	54	0
$\#f'$	18	16	12	11	16	36	10	10	8	7

and 50 as ineffective and report their performance in Table 6.3. Note that for examples 2, 3, 12, 14, and 16 we could not derive 50 effective test suites, likely a downside of us currently requiring O to be acyclic. Out of those 300 *effective* Ts only one for example 15 did not have $MS(T) = 1.00\,(= max.)$ but killed 16 out of 17 generated mutants only ($MS(T) = 0.94$). If we consider the ratio between T_+ and T_- from Table 8.7 we can also say that our approach is quite conservative in handing out its quality label. Keeping in mind that T does not provide the entire truth table, thus the *learned* classification is certainly attractive from those two points of view, at least considering our first experiments. Since we saw in our experiments also that some Ts with an ideal mutation score of 1.0 were classified as ineffective ($T \in T_-$), there is the immediate question whether the computed mutation scores were holistic enough, and what would be an ideal benchmark for comparing our classification (since it could also have been more precise than the mutants).

In terms of encoding efficiency, we see in Table 6.3 that the average size (amount of decision nodes) in the ROBBD R derived for some T was below $2 * \#v$, and most of the times ranged between $\#v$ and $\#v + \frac{\#v}{2}$. Even the maximum size for any R was below $2.5 * \#v$. Thus it seems that our information gain based learning of DT gives a compact ROBDD R with a size far below the worst case $2^{\#v} - 1$ [71].

The run-time for our classification can vary quite a bit with the example. For examples 10 and 20, it took us about 117 seconds and about 0.5 seconds to classify all 100 test suites as of Table 6.3, which we find to be quite attractive.

6.4. Related Research

In [68], Weyuker introduces a method to assess test data adequacy through program inference. Weyuker defines the relation that if a program is adequately tested, then it is correct, but a correct program does not imply that it has been adequately tested. For assessing adequacy, Weyuker uses inference adequacy, where a test suite is adequate if and only if the test suite contains sufficient data to infer the computations defined in the program under test

and its specifications. Weyuker infers programs in a subset of Lisp, but we learn a decision tree from the test data. Inference adequacy also depends on the determination of equivalence, but equivalence of a specification, a program, and an inferred program is in general undecidable. Therefore Weyuker uses approximations to make the inference adequacy criterion usable. Since T gives an incomplete truth table, our learned decision tree DT also is some sort of approximation. In [78] Walkinshaw introduces a test suite adequacy assessment method based on inductive inference, which does not require exact inference, but uses the Probably Approximately Correct (PAC) [63] framework for approximations. To determine equivalence of the inferred model of the test suite and the program under test's specifications we transfer both into ROBDDs where equivalence is decidable.

A family of different strategies, including MAX-A and MAX-B, for automatically generating test cases for Boolean expressions in disjunctive normal form (DNF) is given in [77], where they investigate also the fault detection effectiveness of the different strategies. For our evaluation we used their examples, but in contrast to evaluating test case generation (TCG), our approach classifies any given test suite T. In [79], Chen et al. describe how to generate test suites that satisfy the MUMCUT strategy for testing Boolean expressions in DNF. The MUMCUT strategy guarantees to detect seven fault

1. $(\neg(a \wedge b)) \wedge (d \wedge (\neg e) \wedge (\neg f) \vee (\neg d) \wedge e \wedge (\neg f) \vee (\neg d) \wedge (\neg e) \wedge (\neg f)) \wedge (a \wedge c \wedge (d \vee e) \wedge h \vee a \wedge (d \vee e) \wedge (\neg h) \vee b \wedge (e \vee f))$
2. $(a \wedge ((c \vee d \vee e) \wedge g \vee a \wedge f \vee c \wedge (f \vee g \vee h \vee i)) \vee (a \vee b) \wedge (c \vee d \vee e) \wedge i) \wedge (\neg(a \wedge b)) \wedge (\neg(c \wedge d)) \wedge (\neg(c \wedge e)) \wedge (\neg(d \wedge e)) \wedge (\neg(f \wedge g)) \wedge (\neg(f \wedge h)) \wedge (\neg(f \wedge i)) \wedge (\neg(g \wedge h)) \wedge (\neg(h \wedge i))$
3. $(a \wedge ((\neg d) \vee (\neg e) \vee d \wedge e \wedge (\neg((\neg f) \wedge g \wedge h \wedge (\neg i) \vee (\neg g) \wedge h \wedge i)) \wedge (\neg((\neg f) \wedge g \wedge l \wedge k \vee (\neg g) \wedge (\neg i) \wedge k))) \vee (\neg((\neg f) \wedge g \wedge h \wedge (\neg i) \vee (\neg g) \wedge h \wedge i)) \wedge (\neg((\neg f) \wedge g \wedge l \wedge k \vee (\neg g) \wedge (\neg i) \wedge k)) \wedge (b \vee c \wedge (\neg m) \vee f)) \wedge (a \wedge (\neg b) \wedge (\neg c) \vee (\neg a) \wedge b \wedge (\neg c) \vee (\neg a) \wedge (\neg b) \wedge c)$
4. $a \wedge ((\neg b) \vee (\neg c)) \wedge (d \vee e)$
5. $a \wedge ((\neg b) \vee (\neg c) \vee b \wedge c \wedge (\neg((\neg f) \wedge g \wedge h \wedge (\neg i) \vee (\neg g) \wedge h \wedge i)) \wedge (\neg((\neg f) \wedge g \wedge l \wedge k \vee (\neg g) \wedge (\neg i) \wedge k))) \vee f$
6. $((\neg a) \wedge b \vee a \wedge (\neg b)) \wedge (\neg(c \wedge d)) \wedge (f \wedge (\neg g) \wedge (\neg h) \vee (\neg f) \wedge g \wedge (\neg h) \vee (\neg f) \wedge (\neg g) \wedge (\neg h)) \wedge (\neg(j \wedge k)) \wedge ((a \wedge c \vee b \wedge d) \wedge e \wedge (f \vee (i \wedge (g \wedge j \vee h \wedge k))))$
7. $((\neg a) \wedge b \vee a \wedge (\neg b)) \wedge (\neg(c \wedge d)) \wedge (\neg(g \wedge h)) \wedge (\neg(j \wedge k)) \wedge ((a \wedge c \vee b \wedge d) \wedge e \wedge((\neg i) \vee (\neg g) \wedge k \vee (\neg j) \wedge ((\neg h) \vee (\neg k))))$
8. $((\neg a) \wedge b \vee a \wedge (\neg b)) \wedge (\neg(c \wedge d)) \wedge (\neg(g \wedge h)) \wedge ((a \wedge c \vee b \wedge d) \wedge e \wedge (f \wedge g \vee (\neg f) \wedge h))$
9. $(\neg(c \wedge d)) \wedge ((\neg e) \wedge f \wedge (\neg g) \wedge (\neg a) \wedge (b \wedge c \vee (\neg b) \wedge d))$
10. $a \wedge (\neg b) \wedge (\neg c) \wedge d \wedge (\neg e) \wedge f \wedge (g \vee (\neg g) \wedge (h \vee i)) \wedge (\neg(j \wedge k \vee (\neg j) \wedge l \vee m))$
11. $a \wedge (\neg b) \wedge (\neg c) \wedge ((\neg(f \wedge (g \vee (\neg g) \wedge (h \vee i)))) \vee f \wedge (g \vee (\neg g) \wedge (h \vee i)) \wedge (\neg d) \wedge (\neg e)) \wedge (\neg(j \wedge k \vee (\neg j) \wedge l \wedge (\neg m)))$
12. $a \wedge (\neg b) \wedge (\neg c) \wedge f \wedge (g \vee (\neg g) \wedge (h \vee i)) \wedge ((\neg e) \wedge (\neg n) \vee d) \vee (\neg n) \wedge (j \wedge k \vee (\neg j) \wedge l \wedge (\neg m))$
13. $a \vee b \vee c \vee (\neg c) \wedge (\neg d) \wedge e \wedge f \wedge (\neg g) \wedge (\neg h) \vee i \wedge (j \vee k) \wedge (\neg l)$
14. $a \wedge (\neg b) \wedge (\neg c) \wedge f \wedge (g \vee (\neg g) \wedge (h \vee i)) \wedge ((\neg e) \wedge (\neg n) \vee d) \vee (\neg n) \wedge (j \wedge k \vee (\neg j) \wedge l \wedge (\neg m))$
15. $a \wedge ((c \vee d \vee e) \wedge g \vee a \wedge f \vee c \wedge (f \vee g \vee h \vee i)) \vee (a \vee b) \wedge (c \vee d \vee e) \wedge i$
16. $a \wedge ((\neg d) \vee (\neg e) \vee d \wedge e \wedge (\neg((\neg f) \wedge g \wedge h \wedge (\neg i) \vee (\neg g) \wedge h \wedge i)) \wedge (\neg((\neg f) \wedge g \wedge l \wedge k \vee (\neg g) \wedge (\neg i) \wedge k))) \vee (\neg((\neg f) \wedge g \wedge h \wedge (\neg i) \vee (\neg g) \wedge h \wedge i)) \wedge (\neg((\neg f) \wedge g \wedge l \wedge k \vee (\neg g) \wedge (\neg i) \wedge k)) \wedge (b \vee c \wedge (\neg m) \vee f)$
17. $(a \wedge c \vee b \wedge d) \wedge e \wedge (f \vee (i \wedge (g \wedge j \vee h \wedge k)))$
18. $(a \wedge c \vee b \wedge d) \wedge e \wedge ((\neg i) \vee (\neg g) \wedge (\neg k) \vee (\neg j) \wedge ((\neg h) \vee (\neg k)))$
19. $(a \wedge c \vee b \wedge d) \wedge e \wedge (f \wedge g \vee (\neg f) \wedge h)$
20. $(\neg e) \wedge f \wedge (\neg g) \wedge (\neg a) \wedge (b \wedge c \vee (\neg b) \wedge d)$

Figure 6.4.: The 20 TCAS II examples taken from [77].

types found in Boolean expressions. Also in that paper, the examples from [77] were used to evaluate their approach. In contrast to MAX-A, MAX-B, and MUMCUT, for our approach it is not necessary for the program under test's specifications to appear in a certain normal form.

A strategy to assess the effectiveness of a test suite for decisions (i.e. Boolean expressions) is the modified condition/decision coverage (MCDC) criterion [50] which requires that each condition within a decision is shown by execution to independently affect the outcome of the decision. Showing that each condition independently affects the decision's outcome requires either that the test case generation was directed to satisfy the MCDC criterion, or to execute the program with the inputs from the test suite and check which conditions affect the outcome. Our classification approach does not require the execution of the program under test. MCDC requires for n input variables a test suite at least of size $n + 1$. We saw the requirement reflected also in our experiments when considering T_+. Since the size of a BDD is very sensitive to its variable order, Friedman and Supowit showed in [80] an algorithm for finding an optimal one which is in $O(n^2 3^n)$. Grumberg et al. propose in [81] an approach in which the variable ordering algorithm for creating BDDs gains experience from training models and uses the learned knowledge for finding good orders. In our work, we use entropy and information gain measures for the concrete example and a specific local situation for establishing the variable order.

In contrast to the work about learning automata [82–85] which is based on active learning while executing the program under test, our approach is passive which means that executing the program under test to classify the test suite effectiveness is not necessary.

6.5. Summary

In this chapter, we proposed a new approach at classifying a test suite T's effectiveness in identifying a program's functional faults. To this end, we tailored a special learning algorithm from C4.5 in order to learn a representative decision tree DT from T. If possible, we showed how to isolate a total variable order Ψ from DT via a derived ordering graph, so

that we then use Ψ when deriving a corresponding reduced ordered binary decision diagram also for the SUT's specification. If we were able to retrieve an order Ψ, we reduce also DT to an ROBDD in order to check the two ROBDDs for equivalence.

Our argument is that if they are equivalent, we can assume that a program satisfying T implements the desired functionality as described by the specification (in the form of Boolean functions). In our initial experiments as reported in this chapter, we computed the mutation score for random test suites and compared it to our classification. Even if we assume only the maximum mutation score to be the benchmark for our effectiveness classification (without some error margin), there were only very few "false positives", i.e. 2 out of 337 effective Ts. That we classified also some test suites with a perfect mutation score to be ineffective raises the question whether our mutations, and the mutation score in general, is holistic enough as benchmark for our classification, or if our effectiveness label was more precise than the considered mutations. A corresponding investigation with more examples from multiple domains and further mutation operators [86–88] will be subject to future work.

Aside conducting more experiments and evaluating further benchmark options, future research will also target the question of whether there would be more attractive representations like multi-terminal binary decision diagrams [89] (compared to an ROBDD) for our cause. Currently we are working on finding a suitable approximation metric that relaxes exact equality. Also the implementation of a reordering algorithm is on our agenda, so that we can use our approach also if the initial ordering graph is not acyclic.

Table 6.3.: Performance for 100 random T classified as effective T_+ or ineffective T_-.

		$\lvert T \in T_i \rvert$			$MS(T \in T_i)$					$\lvert T \in T_i \rvert$			$MS(T \in T_i)$		
sample		$\lvert T_i \rvert$	*min.*	*avg.*	*max.*	*min.*	*avg.*	*max.*	sample	$\lvert T_i \rvert$	*min.*	*avg.*	*max.*	*min.*	*avg.*	*max.*
1	T_+	1	63	63	63	1.00	1.00	1.00	11	0	-	-	-	-	-	-
	T_-	99	49	62	78	0.64	0.96	1.00		100	3973	4099	4219	1.00	1.00	1.00
2	T_+	0	-	-	-	-	-	-	12	0	-	-	-	-	-	-
	T_-	100	213	250	289	0.74	0.87	1.00		100	749	805	875	0.94	0.99	1.00
3	T_+	0	-	-	-	-	-	-	13	0	-	-	-	-	-	-
	T_-	100	331	398	444	0.75	0.93	1.00		100	1987	2045	2132	1.00	1.00	1.00
4	T_+	5	18	22	24	1.00	1.00	1.00	14	0	-	-	-	-	-	-
	T_-	95	9	15	21	0.75	0.99	1.00		100	44	63	77	1.00	1.00	1.00
5	T_+	1	265	265	265	1.00	1.00	1.00	15	22	234	259	278	1.00	1.00	1.00
	T_-	99	226	252	288	0.95	0.98	1.00		78	231	254	282	0.88	0.99	1.00
6	T_+	1	1031	1031	1031	1.00	1.00	1.00	16	0	-	-	-	-	-	-
	T_-	99	966	1000	1075	0.70	0.94	1.00		100	164	192	235	0.69	0.94	1.00
7	T_+	0	-	-	-	-	-	-	17	0	-	-	-	-	-	-
	T_-	100	475	498	553	0.85	0.97	1.00		100	968	1019	1067	1.00	1.00	1.00
8	T_+	2	125	127	129	0.88	0.94	1.00	18	1	504	504	504	1.00	1.00	1.00
	T_-	98	102	125	158	0.69	0.95	1.00		99	473	510	547	1.00	1.00	1.00
9	T_+	1	71	71	71	1.00	1.00	1.00	19	0	-	-	-	-	-	-
	T_-	99	50	63	73	0.55	0.92	1.00		100	103	124	143	1.00	1.00	1.00
10	T_+	1	4211	4211	4211	1.00	1.00	1.00	20	2	61	69	77	1.00	1.00	1.00
	T_-	99	3978	4090	4219	0.86	0.99	1.00		98	47	63	78	0.85	0.98	1.00

Table 6.4.: The experimental results for 6 of the 20 example specifications.

ex.	$\#v$	$\varnothing \lvert R \rvert$	$max. \lvert R \rvert$	$MS(T \in T_+) = 1$	$MS(T \in T_-) = 1$	$\varnothing \lvert T \in T_+ \rvert$	$\varnothing \lvert T \in T_- \rvert$
1	7	11.06	15	50	22	68.56	63.42
6	11	17.04	26	50	39	1847.38	1848.18
9	7	9.36	12	50	30	67.56	65.18
10	13	14.34	17	50	48	5467.16	5457.62
15	9	11.05	15	49	40	259.06	252.78
20	7	7.8	14	50	32	31.18	25.84

7. Mutation Score, Coverage, Model Inference: Quality Assessment For t-way Combinatorial Test Suites

This chapter is based on the work "Mutation Score, Coverage, Model Inference: Quality Assessment for T-Way Combinatorial Test-Suites" [8]. In this chapter we investigate the test suite quality, measured by mutation score [90], code coverage [91] and model inference for test suites created from t-way combinatorial interaction test case generation. Therefore the most important question of course is: (**RQ1**) How does incrementing t affect the test suite quality?

As a consequence from the limitations of assessing a test suite's quality by mutation score or coverage, we introduce a new quality assessment approach in this chapter. This new quality assessment approach is based on model inference. We infer a model from a test suite and assess the test suite's quality by evaluating the inferred model. The underlying idea is to use a test suite which is known to be of high quality and compare it to another test suite of unknown quality, to obtain a quality valuation for the test suite of unknown quality. As a comparison method we infer a model from the test suite and evaluate the inferred model by using a set of test data, which contains only the test cases from the high quality test suite that are not in the test suite of unknown quality. As an instantiation of model inference we use decision tree learning [51]. Therefore, for evaluation we classify the test data according to their input values down the tree to the leaf nodes and check whether the label of the leaf node corresponds to the expected outcome of the test case. Since this new approach does not

require the execution of the program under test, it could drop the limitations of mutation score and coverage computation. Consequently, for this new approach, we ask the following question: (**RQ2**) Does a model inference based test suite quality assessment approach show similar differences for test suite quality of test suites generated with different t as mutation score or code coverage?

In our experimental evaluation we used 6 examples to answer both research questions. First we created the input models for the respective parameters of the examples. Then, from these input models, we generated the t-way combinatorial test suites (we used the original unmodified examples as oracle to obtain the expected outcome for each test case). The experimental results show that increasing t leads to an enhancement of the quality of a test suite. Furthermore, the results show, that under restrictive conditions the introduced model inference based quality assessment approach is applicable and provides results similar to mutation score and code coverage.

7.1. Model Inference

In this chapter we infer a model from a test suite T_t by learning a decision tree again. We used again the C4.5 algorithm as introduced in [51]. Other than in Chapter 5 and Chapter 6 where model inference is used to evaluate the effectiveness or quality of a test suite by comparing the test suite to the program under test, we compute the quality valuation by comparing test suites by each other. The test suite for which we assess its quality is compared to the test suite $T_{t_{max}}$. We assume $T_{t_{max}}$ to be a test suite of highest quality. We assess a test suite to be of highest quality if the following two conditions hold:

1. The inferred model contains all possible outcomes of the set of outcomes O as leaf nodes.
2. The inferred model classifies a set of test data TD correctly to these leaf nodes.

The set of test data TD is the test suite $T_{t_{max}}$ excluding test cases which exist in other test suites with lower strength for the same program under test.

The test suite $T_{t_{max}}$ is a test suite generated with highest strength t_{max}. t_{max} is either n if the number of parameters n is less than 6, or 6 otherwise, because 6 is the highest strength supported by the test case generator we used in this work. Therefore TD is defined as:

Definition 11 (Test Data) *The test data TD are a subset of $T_{t_{max}}$ and are calculated as:*

$$TD = T_{t_{max}} \setminus \bigcup_{t=1}^{t<t_{max}} T_t \tag{7.1}$$

For a test suite T_t that is not of highest quality its quality is approximated by using the three values from: 1. the root mean squared error while inferring the model, 2. the model's root mean squared error after classifying TD, and 3. the number of outcomes from O which do not appear as leaf nodes in the inferred model.

If the outcomes from TD and the inferred model do not coincide when classifying a test case from TD down the learned decision tree, we calculate the root mean squared error as defined in Definition 6, but instead of the outcomes of an SUT we use the outcomes of TD.
If the outcomes of the inferred model and the ith test case in TD are different, the difference of $(p_i - a_i)$ depends on the number of different categories the inferred model and TD categorize and on possible misclassifications (misclassified tests) of test cases from T_t while inferring the model. A consequence of misclassified test cases from TS while inferring the model is a root mean squared error $RMSE_T > 0$.

7.1.1. Model contains all $o \in O$ criterion

Checking whether an inferred model contains all $o \in O$ requires to collect all distinct leaf node labels of the inferred model into a set L. The set O is built from all distinct outcomes from all test cases in the test suites T_1, ..., $T_{t_{max}}$. Since the model is inferred from a test suite T_t and O contains all distinct outcomes from T_t, ensures that $|L| \leq |O|$. Therefore we use the value missing classes MC when assessing the quality of a test suite. MC is defined as:

Definition 12 (Missing Classes) *Missing classes MC is the proportion between the number of outcomes which do not exist as leaf node label in the inferred model of T_t and the number of all possible outcomes $|O|$.*

$$MC = \frac{|O| - |L|}{|O|} \tag{7.2}$$

7.1.2. Model inference based quality valuation

The model inference based quality valuation is calculated from the classification results and the proportion of missing classes *MC* of a model inferred from the test suite under evaluation. The classification results are the *RMSE* obtained by classifying the test data *TD* and $RMSE_T$ which indicates misclassifications of *T* from which the model was initially inferred. Therefore, we define the valuation *MI* as:

Definition 13 (Model Inference Based Quality Valuation) *The value of MI is calculated as*

$$MI = 1 - (RMSE - MC + RMSE_T) \tag{7.3}$$

7.2. Experimental Results

In this Section we describe the experimental results we obtained for mutation score, code coverage, and our model inference based assessment approach as well as the tools we applied to obtain these results.

7.2.1. Tools

We used different tools to generate the *t*-way combinatorial test suites, generate mutants, instrument source code and analyze code coverage, and to infer models from a test suite for model inference based quality valuation.

- Test case generator: To generate *t*-way combinatorial test suites for our examples we used again the tool ACTS 3.0[1] (Automated Combinatorial Testing for Software). The tool implements various test case generation algorithms. Because we also used constraints in some examples we had to choose either IPOG or IPOG-F [17] which support constraints usage. A list of test case generators can be found in [92]. We decided to use ACTS, because it is free and publicly available, it is a Java tool that we can integrate in our tool chain, it is very popular in combinatorial testing research literature, and it can generate test cases up to 6-way interactions. Currently, three input types are supported: enum, boolean, and integer.
- Mutants generator: The Major[2] mutation framework is divided into two parts. First, a mutation generator for Java programs, and second, an analysis back-end to execute JUnit test cases and assess the test suite quality. In this work we use the first part to generate mutants for our sample programs with the exception of TCAS where we used the existing mutants. Major is integrated into OpenJDK[3] and generates mutants during compilation. A detailed description can be found in [38].
- Coverage analyzer: CodeCover[4] is a freely available so called glass box testing Java tool which we can use as a library in our tool chain. Glass box testing meaning the execution and recording of source code artifacts which are statements, branches, loops, etc.. Here we use CodeCover to instrument and analyze the code coverage of the example test suites we generated.
- Model inference: Weka [60, 93] is a collection of machine learning algorithm implementations for data mining tasks. It can be used as a standalone application or as a library in projects running within the Java Virtual Machine. Here we used Weka as a library to infer a decision tree from a test suite within a Java program. The Java program which includes the C4.5 algorithm is called J48 in Weka. For the J48 algorithm we changed from the standard settings the pruning settings of the inferred tree such that pruning is not performed. Furthermore

[1] http://csrc.nist.gov/groups/SNS/acts
[2] www.mutation-testing.org
[3] http://openjdk.java.net/
[4] www.codecover.org

Table 7.1.: Lines of code and number of mutants per example.

name	SLOC	#mutants
BMI	19	28
Triangle	30	35
UTF8	56	147
TCAS	100	41
J48	3406	3107
Soot-PDG	1701	567

we set the minimum number of instances (here test cases) per leaf to one (1).

7.2.2. Example Programs and Input Models

To evaluate empirically the quality of t-way combinatorial test suites we use 6 Java programs for which the number of source lines of code (SLOC) and the number of mutants (#mutants) are given in Table 7.1.

BMI

The BMI example [56] accepts 2 numeric floating point input parameters which represent weight and height values for which a body mass index is calculated. The returned body mass index is one of 5 possible outcomes. Table 7.2 shows the input model of the BMI example where we use 4 different values for height and 8 different values for weight.

Table 7.2.: BMI input model.

Parameter	Values
height	$\{1.6, 1.8, 2.0, 2.2\}$
weight	$\{73, 74, 99, 100, 119, 120, 159, 160\}$

Triangle

This example [57] uses 3 numeric input parameters representing the lengths of the 3 sides of a triangle and returns which type of triangle can be built with these values. 4 possible types exist where 3 types represent valid triangles and one represents an invalid triangle. The input model is given in Table 7.3.

Table 7.3.: Triangle input model.

Parameter	Values
a	$\{-1, 0, 1, 3, 4, 5, 2147483647\}$
b	$\{-1, 0, 1, 3, 4, 5, 2147483647\}$
c	$\{-1, 0, 1, 3, 4, 5, 2147483647\}$

UTF8

Guava UTF8[5] (UTF8) is a function in Google's Guava library which checks if an input sequence of up to 4 bytes is a well formed UTF8 encoded input. The input model for the 4 input parameters, which represent the input sequence, is given in Table 7.4. The values in the table are 8-bit signed Integer values.

In this example we also use some constraints which are shown in Table 7.5. These constraints ensure that, if an input parameter in the byte sequence is

Table 7.4.: UTF8 input model.

Parameter	Values
b1	$\{0, -1, 127, -128, -62, -63, -33, -32, -31, -30,$ $-20, -19, -18, -17, -16, -15, -14, -13, -12, -11\}$
b2	$\{-128, -65, -64, -97, -96, -112, -113, ?\}$
b3	$\{-128, -65, -64, ?\}$
b4	$\{-128, -65, -64, ?\}$

[5] https://github.com/google/guava

Table 7.5.: Constraints of UTF8 example.

constraints
$(b2 == ?) => (b3 == ?)$
$(b3 == ?) => (b4 == ?)$

Table 7.6.: TCAS input model.

Parameter	**Values**
Cur_Vertical_Sep	$\{299, 300, 601\}$
High_Confidence	$\{0, 1\}$
Two_of_Three_Reports_Valid	$\{0, 1\}$
Own_Tracked_Alt	$\{1, 2\}$
Own_Tracked_Alt_Rate	$\{600, 601\}$
Other_Tracked_Alt	$\{1, 2\}$
Alt_Layer_Value	$\{0, 1, 2, 3\}$
Up_Separation	$\{0, 399, 400, 499, 500,$ $639, 640, 739, 740, 840\}$
Down_Separation	$\{0, 399, 400, 499, 500,$ $639, 640, 739, 740, 840\}$
Other_RAC	$\{0, 1, 2\}$
Other_Capability	$\{1, 2\}$
Climb_Inhibit	$\{0, 1\}$

empty then also the following bytes are empty. Empty values are represented by $'?'$.

TCAS

The TCAS[6] example implements an aircraft collision avoidance system for which mutants exist. The input model is shown in Table 7.6.

[6] http://sir.unl.edu/portal/bios/tcas.php

Table 7.7.: J48 input model.

Parameter	Values	Parameter	Values
-U	$\{F, T\}$	-S	$\{F, T\}$
-O	$\{F, T\}$	-L	$\{F, T\}$
-C	$\{0.0, 0.1, 0.9, 1.0\}$	-A	$\{F, T\}$
-M	$\{0, 1, 2, 5\}$	-J	$\{F, T\}$
-R	$\{F, T\}$	-Q	$\{0, 1, 100\}$
-N	$\{0, 3, 10\}$	-B	$\{F, T\}$
-dNMSAV	$\{F, T\}$		

Table 7.8.: Constraints of J48 example.

constraints
$\neg(U \wedge S)$
$\neg(U \wedge R)$
$\neg R \vee \neg C$
$\neg U \vee \neg C$
$R \vee \neg N$

J48

We introduced Weka in Section 7.2.1 as tool, but we also used the J48 classifier package from Weka as an example for our empirical evaluation. The input for the J48 classifier is a set of data to build a classifier from. In this example we used the configuration parameters of the classifier to build t-way combinatorial test suites from.

The input model for this example is given in Table 7.7 and the constraints which prevent invalid configurations are listed in Table 7.8.

Soot-PDG

Soot[7] is a framework for analyzing and transforming Java and Android applications. Here we use only the part of Soot which constructs an in-

[7] https://sable.github.io/soot/

traprocedural program dependency graph (PDG) [94]. The input is a Java source file containing a single method with nested control statements up to a nesting depth of 6 levels. For each level one of the control statements from table 7.9 is selected. We grouped the control statements into 3 groups. The artifacts labeled with a $'*'$ are only used in the innermost nesting, which is level 6, because they do not represent branching statements which allow further nesting. The selection is conducted by combining t-way combinatorial test case generation and random input selection.

Table 7.9.: Soot-PDG control statements.

Artifacts		
Group 1	Group 2	Group 3
IF-ELSE_IF-ELSE	ENHANCED_FOR	THROW*
SWITCH	ENHANCED_FOR_BREAK	RETURN*
SWITCH_BREAK	ENHANCED_FOR_CONTINUE	CALLABLE*
TRY_CATCH_FINALLY	BASIC_FOR	NOP
LINEAR_RECURSION	BASIC_FOR_BREAK	
NOP	BASIC_FOR_CONTINUE	
	WHILE	
	WHILE_BREAK	
	WHILE_CONTINUE	
	DO_WHILE	
	DO_WHILE_BREAK	
	DO_WHILE_CONTINUE	
	NOP	

Table 7.10 shows the input model for the Soot-PDG example. It lists 6 parameters which represent the levels of nesting. The values for the parameters are the groups of Table 7.9 from which a control statement is randomly selected while building a test case.

7.2.3. Mutation score results

Here we show the mutation scores of the t-way combinatorial test suites for each example.

The BMI example has only two parameters, therefore we generated only two test suites. The results in Figure 7.1 show that an increasing mutation

Table 7.10.: Soot-PDG input model.

Parameter	Values
L1	$\{1,2\}$
L2	$\{1,2\}$
L3	$\{1,2\}$
L4	$\{1,2\}$
L5	$\{1,2\}$
L6	$\{1,2,3\}$

score correlates with an increasing test suite size. The t-way combinatorial test suite with t_{max} achieves a maximum mutation score of $\mu = 1$.

Because the Triangle example has only 3 parameters we generated only 3 test suites with $t_{max} = 3$. The results for the Triangle example are shown in Figure 7.2. These results also, as in the BMI example, show a correlation of mutation score and test suite size. Again the t-way combinatorial test suite with t_{max} achieves a maximum mutation score of $\mu = 1$.

The UTF8 example has only 4 parameters. The results for the 4 t-way combinatorial test suites are shown in Figure 7.3. The mutation scores show a degressive curve for increasing t, but test suite size shows a progressive curve. Also here a mutation score of $\mu = 1$ was achieved.

The TCAS example has 12 parameters and we generated a test suite for each $t \in \{1, ..., 6\}$. Here we used given mutants for which the mutation score results are shown in Figure 7.4. Again the mutation scores show a nearly degressive curve for increasing t, but test suite size shows a progressive curve. Also here a mutation score of $\mu = 1$ was achieved.

We generated a test suite for each $t \in \{1, ..., 6\}$ for the J48 example which has 13 different configuration parameters. Figure 7.5 shows the mutation score and the test suite size for the generated test suites. We executed the test suite with 3 different datasets (*.arff files) from which we learned the decision trees. These datasets contained different attribute types, which are numeric, nominal, or date (for decision tree learning, attributes of type String are not supported by Weka). For each dataset we obtained a set of killed mutants, therefore we calculated the mutation score from the union of these sets of killed mutants. The test suite size shows a progressive curve,

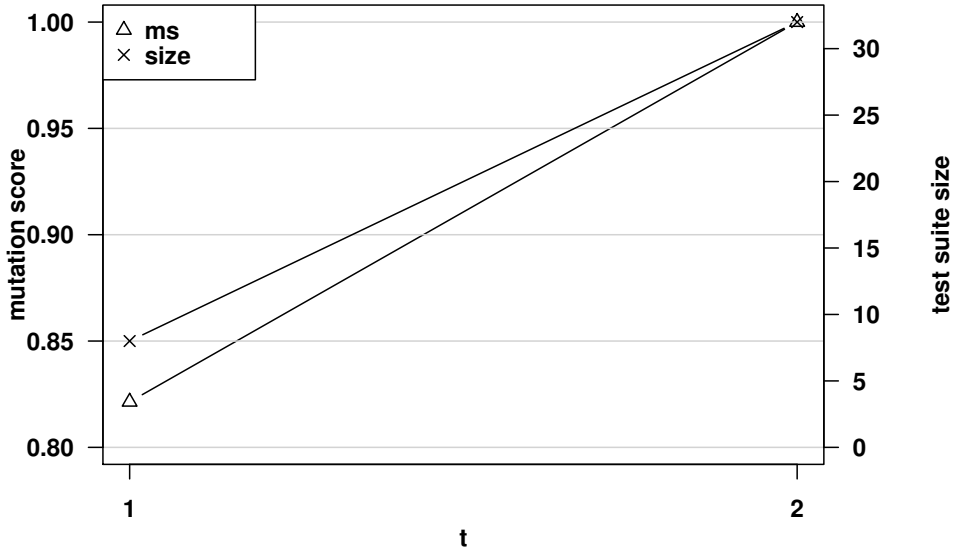

Figure 7.1.: Mutation score and test suite size per t-way combinatorial test suite for the BMI example.

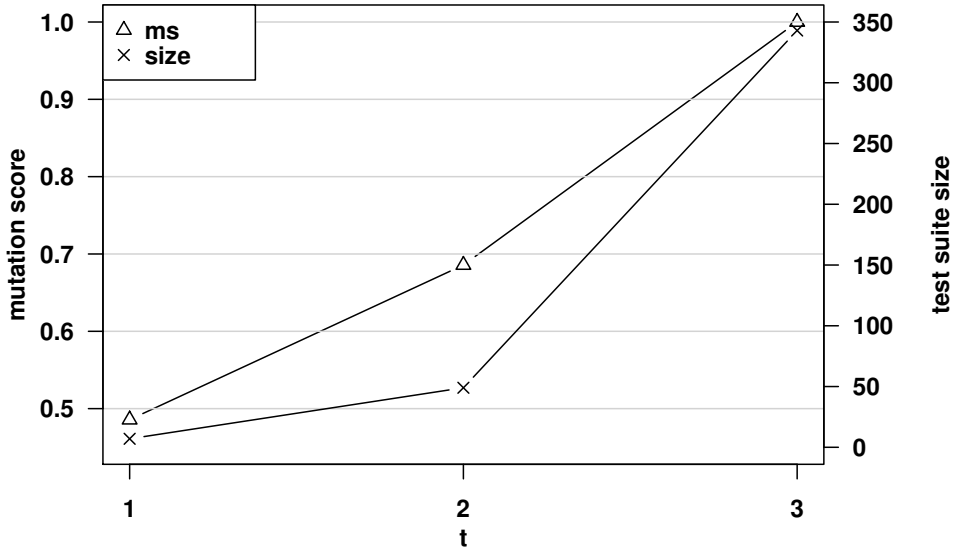

Figure 7.2.: Mutation score and test suite size per t-way combinatorial test suite for the Triangle example.

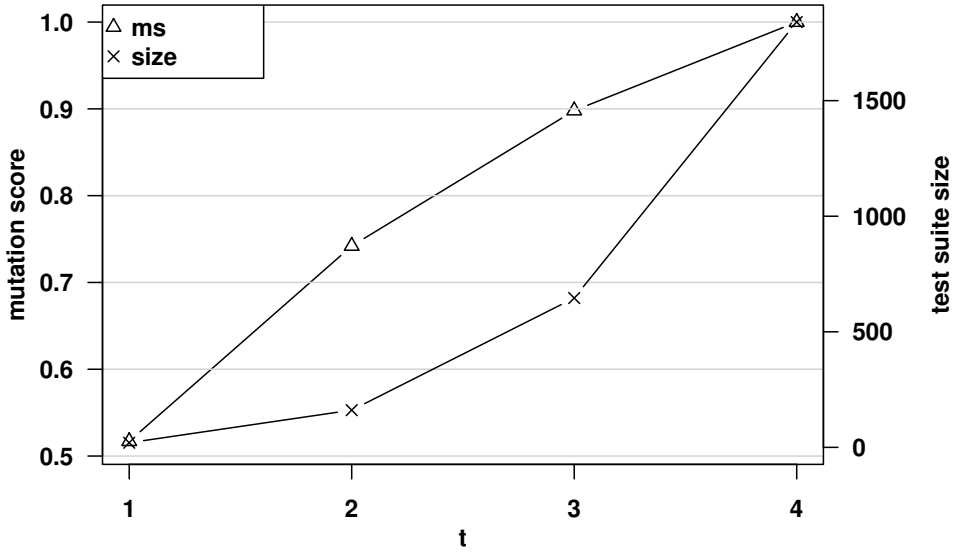

Figure 7.3.: Mutation score and test suite size per t-way combinatorial test suite for the UTF8 example.

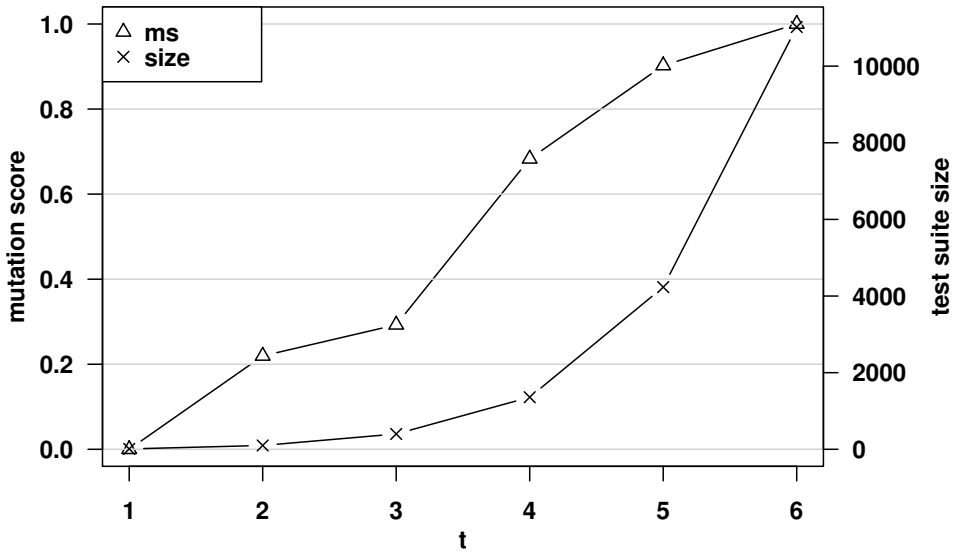

Figure 7.4.: Mutation score and test suite size per t-way combinatorial test suite for the TCAS example.

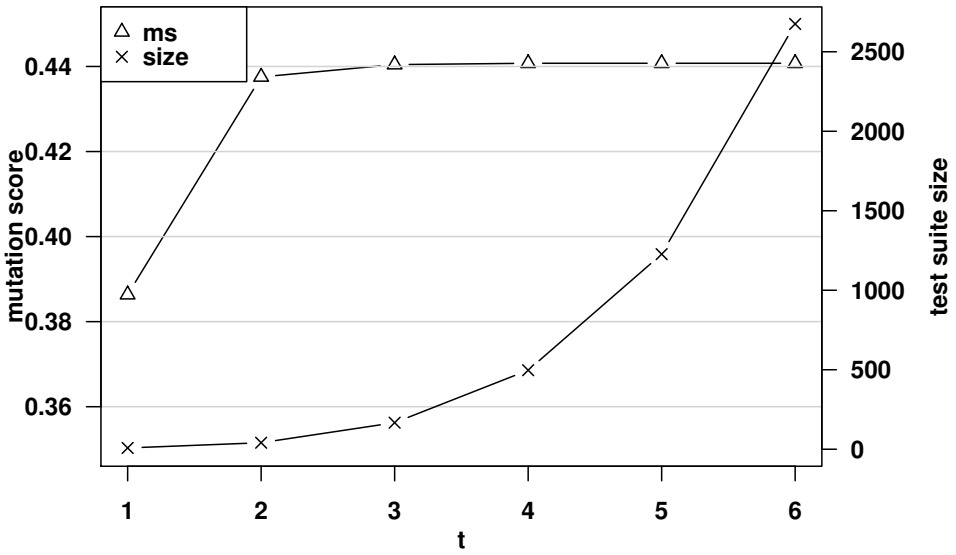

Figure 7.5.: Mutation score and test suite size per t-way combinatorial test suite for the J48 example.

but the mutation score almost does not change for t-way combinatorial test suites with $t \geq 2$.

In the input model for the Soot-PDG example we defined 6 parameters. Because we combined random input selection and t-way combinatorial test suite generation in this example we generated 10 test suites for each $t \in \{1, ..., 6\}$. Table 7.11 shows the min., max., and average mutation score results of the 10 test suites for each t. The mutation score shown in Figure 7.6 is the average mutation score from Table 7.11. Again the test suite size shows a progressive curve and the mutation score for $t \geq 2$ increases almost linearly.

7.2.4. Code Coverage Results

We divided the coverage results into two Tables 7.12 and 7.13, because we generated and analyzed for the Soot-PDG example multiple test suites for each strength and for the remaining examples we generated only a single test suite per strength. The code coverage results for the Soot-PDG example

Table 7.11.: Mutation score of Soot-PDG example.

t	min.	max.	avg.
1	0.31	0.40	0.36
2	0.37	0.42	0.40
3	0.37	0.44	0.41
4	0.37	0.45	0.42
5	0.41	0.45	0.44
6	0.45	0.45	0.45

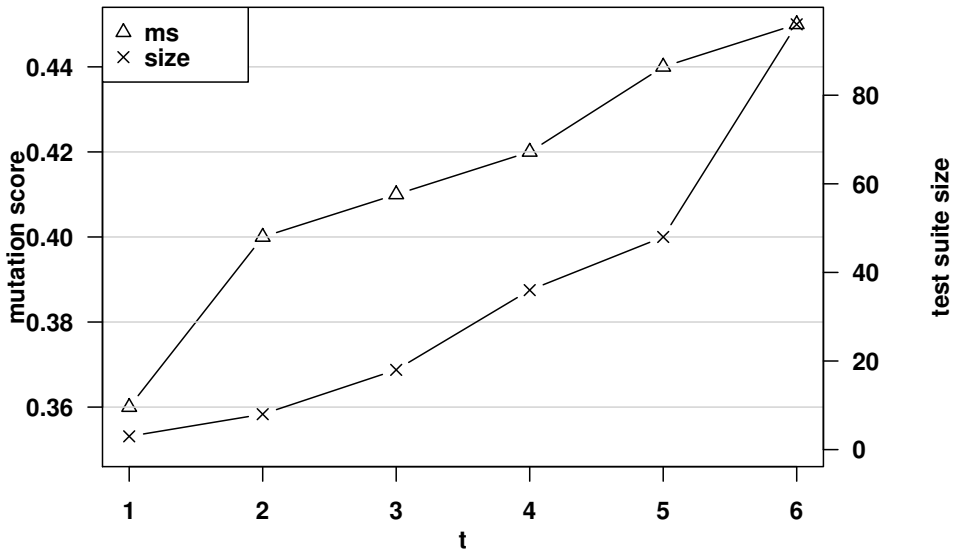

Figure 7.6.: Mutation score and test suite size per t-way combinatorial test suite for the Soot-PDG example.

are shown in Table 7.12. We used the same 10 test suites for each t as used for computing the mutation score and calculated the min., max., and average values for statement, branch, and MC/DC coverage. The differences of coverage for the Soot-PDG example from T_1 to T_6 are only in the range of 15% to 23%. The higher the strength, the closer are the values for min., max., and avg. coverage. The coverage results for the other examples are shown in Table 7.13. For the three examples BMI, Triangle, and UTF8 we achieved 100% statement, branch, and MC/DC coverage for $T_{t_{max}}$. Achieving 100% coverage for the TCAS example is impossible due to unreachable code. The coverage results for the J48 example are almost equivalent for all test suites from T_1 to T_6.

Table 7.12.: Coverage Results of Soot-PDG test suites.

	coverage								
	statement			branch			MC/DC		
t	min.	max.	avg.	min.	max.	avg.	min.	max.	avg.
1	57.41	67.12	61.95	29.30	46.48	37.66	35.86	51.31	43.06
2	61.32	71.16	66.36	39.06	49.22	44.69	44.76	53.66	49.55
3	64.46	72.91	68.48	39.45	52.34	47.77	45.03	55.76	52.20
4	61.73	73.18	69.77	39.84	53.13	48.44	45.29	56.28	52.38
5	66.98	73.32	72.36	44.53	53.52	51.84	48.69	56.54	55.24
6	72.91	73.32	73.11	52.34	53.52	52.93	55.76	56.54	56.15

7.2.5. Model Inference Results

Here we show the results of *MI* for the t-way combinatorial test suites, which are calculated as introduced in Section 7.1. Since we generated 10 test suites for each t of the Soot-PDG example, we also had to create 10 test data sets *TD*. We used these test data sets *TD* to assess the test suites and obtained the intermediate results as shown in Table 7.14. These results show the min., max., and average values of the size $|L|$, the number of incorrectly classified test cases from *TD*, and the *RMSE*. The results show that the number of distinct leaves increases with higher strength, but the number of incorrectly classified test cases from *TD* only slightly decreases.

Table 7.15 shows the *MI* results for the 6 example programs. As the test suites $T_{t_{max}}$ have an *MI* of 1, test suites of high quality have an *MI* of 1 or

Table 7.13.: Coverage Results.

		t					
		1	2	3	4	5	6
BMI	statem.	85.71	100.00				
	branch	87.50	100.00				
	MC/DC	87.50	100.00				
Triangle	statem.	61.54	92.31	100.00			
	branch	75.00	91.67	100.00			
	MC/DC	56.25	75.00	100.00			
UTF8	statem.	85.71	100.00	100.00	100.00		
	branch	85.00	100.00	100.00	100.00		
	MC/DC	57.50	100.00	100.00	100.00		
TCAS	statem.	50.00	94.44	94.44	97.22	97.22	97.22
	branch	08.33	83.33	83.33	91.67	91.67	91.67
	MC/DC	15.00	70.00	70.00	85.00	85.00	85.00
J48	statem.	48.22	49.81	49.81	49.81	49.81	49.81
	branch	48.06	51.74	51.74	51.74	51.74	51.74
	MC/DC	48.24	50.88	50.88	50.88	50.88	50.88

close to 1. Only the examples UTF8 and TCAS have test suites of quality > 0.9 for a test suite strength $< t_{max}$. The resulting MI_1 for the J48 and Soot-PDG example indicate test suites of lowest quality compared to a test suite $T_{t_{max}}$, but the results for the Soot-PDG example in general are very low, which indicates that the quality valuation MI might not be applicable for this example.

7.3. Discussion

Because for the UTF8 example there exist several invalid input byte sequences, creating the input model for the UTF8 example required more effort than for the BMI and Triangle examples. In all three examples BMI, Triangle, and UTF8 the number of input parameters n is smaller than 6. If the number of input parameters is less or equal to 6, the generated test suite $T_{t_{max}}$ is the exhaustive test suite of the provided input model, if no

Table 7.14.: *MI* Results for Soot-PDG.

	|L|			incorr.			RMSE		
t	min.	max.	avg.	min.	max.	avg.	min.	max.	avg.
1	2	3	2.8	59	96	88.1	0.0915	0.1160	0.1010
2	4	8	6.1	58	94	70.5	0.0917	0.1121	0.1002
3	9	14	11.4	65	81	74.1	0.0963	0.1124	0.1051
4	13	27	20.6	60	78	71.0	0.0970	0.1079	0.1029
5	25	34	29.0	63	82	72.6	0.0960	0.1086	0.1035

constraints exist (there exist constraints for the UTF8 example). We conclude that for the 3 smaller examples the created input models are well chosen and the test suites $T_{t_{max}}$ are of highest quality, because mutation score and coverage of $T_{t_{max}}$ are the maximum values. Considering the *MI* results, we see that mutation score, coverage, and *MI* show similar characteristics with increasing t.

For the TCAS example we took the same input model as provided in [25]. Also for the TCAS example the results of mutation score, coverage, and *MI* show similar characteristics. Achieving a coverage of 100% is not possible for the TCAS example, because it contains unreachable code. As discussed in [25] the strength of the mutants (or faults) to evaluate the TCAS example is higher than 6, which means that the minimum number of parameters that must be involved to trigger the fault is higher than 6. Thanks to the test generation algorithm which adds those values for the parameters in a test case randomly, which are not relevant in the current combination, the test suites can achieve highest quality coincidentally.

For the J48 example we obtained coverage results of around 50% for statement, branch, and MC/DC coverage and a max. mutation score of 0.44 from all generated t-way combinatorial test suites. These results encouraged us to run 2 different analyses to figure out the reasons for these weak results. First we analyzed whether there are certain types of mutants which could not be killed (same as in Chapter 5), because we only used a small set of inputs and used for generating the test suites the configuration parameters. But as shown in Table 7.16 there is no type of mutants which shows significant differences of mutation score to the overall mutation score.

Second we investigated the mutants in detail where we figured out that

Table 7.15.: Model Inference Results without $T_{t_{max}}$ where MI is 1 (results indexed by t).

	BMI	Triangle	UTF8	TCAS	J48	Soot-PDG
$\|O\|$	5	4	2	3	161	124.6
$\|TD\|$	24	288	1115	10860	2633	96
MC_1	0.2	0.5	0	0.67	0.9689	0.9775
$RMSE_1$	0.5164	0.4488	0.4442	0.0508	0.0982	0.1010
$RMSE_{TS_1}$	0	0	0	0	0	0
MI_1	**0.2836**	**0.0512**	**0.5558**	**0.2792**	**0**	**0**
MC_2		0.25	0	0.33	0.8882	0.9508
$RMSE_2$		0.32	0.3738	0.1722	0.0986	0.1002
$RMSE_{TS_2}$		0	0.1651	0	0	0
MI_2		**0.43**	**0.7913**	**0.4978**	**0.0132**	**0**
MC_3			0	0.33	0.6211	0.9083
$RMSE_3$			0.2831	0.0825	0.1009	0.1051
$RMSE_{TS_3}$			0.1899	0	0	0
MI_3			**0.9088**	**0.5875**	**0.278**	**0**
MC_4				0	0.2174	0.8354
$RMSE_4$				0.0907	0.1018	0.1029
$RMSE_{TS_4}$				0	0	0
MI_4				**0.9093**	**0.6808**	**0.0617**
MC_5				0	0.0373	0.7672
$RMSE_5$				0.0405	0.1023	0.1035
$RMSE_{TS_5}$				0	0	0
MI_5				**0.9595**	**0.8604**	**0.1293**

there were:

1. 8 mutants in an abstract class which methods were never executed.
2. 437 mutants in 4 classes implementing a Naive Bayes classifier [95] which is never executed.
3. 52 mutants in a class that was only used from a different package implementing a rule-based classifier [95].
4. 2 mutants that could not be killed because their execution was prevented by our constraints.
5. 208 mutants that occur in methods which are provided as an API, but do not contribute to learn a decision tree.
6. ∼400 mutants that can only be killed with different input files.

Since we only used 3 different input files to learn a decision tree there were

Table 7.16.: Mutation score per mutant type per t for the J48 example.

type	**#mutants**	$\mu_{t=1}$	$\mu_{t=2}$	$\mu_{t=3}$	$\mu_{t=4}$	$\mu_{t=5}$	$\mu_{t=6}$
AOR	708	0.5579	0.6314	0.6384	0.6384	0.6384	0.6384
LVR	1065	0.3831	0.4282	0.4310	0.4319	0.4319	0.4319
ROR	577	0.2617	0.2912	0.2912	0.2912	0.2912	0.2912
STD	366	0.2432	0.2814	0.2814	0.2814	0.2814	0.2814
COR	382	0.4031	0.4738	0.4764	0.4764	0.4764	0.4764
ORU	9	0.3333	0.4444	0.4444	0.4444	0.4444	0.4444

about 400 mutants which definitely could have been killed with different input files. In this chapter we investigated the effect of increasing t when generating combinatorial test suites over configuration parameters. Therefore, after investigating the unkilled mutants, we can raise the mutation score results, due to unreachable code thanks to a small set of inputs, to:
$\mu_{t=1}$=0.6003, $\mu_{t=2}$=0.6798, $\mu_{t=3}$=0.6843,
$\mu_{t=4}$=0.6848, $\mu_{t=5}$=0.6848, $\mu_{t=6}$=0.6848.
The investigation of the unkilled mutants also explains the low coverage results. Analyzing the *MI* results shows that the test suites T_1, T_2, and T_3 have a very high *MC* which means that most of the decision trees learned with the configuration parameters in T_6 could not be created with these smaller test suites. To compare the decision trees we used the textual output of Weka and checked for textual equivalence.

To generate the Soot-PDG test suites we combined t-way combinatorial test generation and random input selection. Since we only obtained an average statement coverage of 73.11% we conclude that there were again unkilled mutants in unreachable code, as explained for the J48 example. The average mutation score from T_2 is 0.40 and only slightly raises to 0.45 for T_6. For the Soot-PDG example mutation score and coverage correlate linearly, but *MI* is not applicable for this example, because every input leads to a different program dependency graph. Here we also used the textual outputs of the program dependency graphs from Soot and compared them by checking for textual equivalence.

For the examples with $|O|$ smaller than the test suite sizes, the results of mutation score, coverage, and *MI* are similar. When $|O|$ is bigger than the test suites the quality assessment using *MI* does not provide any meaningful

results as shown in the J48 and the Soot-PDG examples. This origins in the fact that the inferred models do not contain leaf nodes to which certain test cases in *TD* could be classified.

7.4. Threats to validity

The mutation score results might differ slightly, because we did not investigate the behavior of the mutants and therefore can not exclude the occurrence of equivalent mutants. Considering the selection of the examples for the experimental analysis, the examples are either very small, measured in lines of code, or represent only parts of a bigger program, because examples with more lines of code and more mutants are not executable in feasible time. Furthermore we did not investigate test suites with strength $t > 6$.

7.5. Related Work

Ghandehari et al. in [96] used 7 C-programs from the Siemens suite[8] and measured effectiveness of a test suite in two dimensions, i.e., code coverage and fault detection. They compared effectiveness of t-way combinatorial test suites and random test suites. To generate t-way combinatorial test suites they used the test case generation tool PICT [22]. PICT uses a greedy, random algorithm for t-way test generation which allows the user to specify a seed manually. In addition the authors generated for each t-way test suite a random test suite of the same size, where they used the same input model for random test generation as for combinatorial test generation. In most cases t-way testing was as effective as, or more effective than random testing. Overall the differences between the two test generation methods are not as significant for these small examples as one would probably have expected. The 7 C-programs vary only between 141 and 512 lines of code, whereas we use example programs of up to 3406 lines of code. In [97] the authors

[8] http://sir.unl.edu

described the seeded faults in the 7 C-programs as follows:"The faults are mostly changes in single lines of code, but a few involve multiple changes. Many of the faults take the form of simple mutations or missing code". Therefore these faults are essentially mutants. In [25] the authors report the details of modeling the input parameters for combinatorial testing of examples from the Siemens suite and show the effectiveness of test suites generated by combinatorial testing. The effectiveness is measured in terms of the number of detected faulty versions. Their results show that combinatorial testing is more effective than random testing, but for 5 out of the 7 examples only 2-way combinations were possible, one 3-way, and one 6-way. In this work we used examples with up to 13 parameters and provide different results for some 6-way combinatorial test suites.
Another study comparing the effectiveness in terms of mutation score of automatically generated tests, constructed using random and t-way combinatorial techniques is provided in [23]. The empirical evaluation contains 10 functions with 12 to 62 lines of code for which combinatorial t-way test suites for $t \in \{2,..,5\}$ were generated. Their results show that 2-way combinatorial test suites are not as effective as random test suites of the same size, and that random test suites can be effective but are not reliable. The authors of [98] conducted four relatively large projects, in which they applied model based testing and pairwise combinatorial test generation to systems with millions of lines of code. In their work they chose sequences of steps based on operational profiles and used 2-way combinatorial test generation to choose the values tested in each step. From the generated test suites 2% to 23% of the test cases failed. These test cases revealed so far undetected faults. Schroeder et al. compared in [24] fault detection effectiveness of t-way combinatorial test suites and random test suites in terms of code coverage. They concluded that t-way combinatorial test suites were no more effective than random test suites of the same size. The authors of [99] analyzed code coverage achieved by t-way combinatorial testing. They conclude that for 2-way combinations block coverage was comparable with exhaustive testing, but for an acceptable path coverage higher values for t are required.
In [100] the authors show that software failures in a variety of domains were caused by combinations of relatively few conditions. From an analysis of 15 years of recall data [21] they conclude that t-way combinatiorial test generation for a max. of $t = 6$ is enough to detect all faults. In [12] Nie

and Leung provide a survey of combinatorial testing. They investigated the history of combinatorial testing where they analyzed 93 papers and assigned them to certain combinatorial testing research categories, which are: modeling for combinatorial testing, test suite generation, constraints, failure diagnosis, prioritization, metric, evaluation, testing procedure, and the state of research.

7.6. Summary

We conclude that the quality of t-way combinatorial test suites increases with higher strength. The quality of a test suite heavily depends on the input model. Therefore the answer for **RQ1** is that in this chapter for the used Java examples incrementing t affects the test suite quality such that the quality raises. The three applied quality assessment criteria, which are mutation score, coverage, and model inference based assessment, show similar quality differences from test suite to test suite for different strength t. We intend to use our newly introduced model inference based quality assessment criterion as a supplementary criterion for mutation score and coverage that delivers results in a short time and does not affect the behavior of the program under test, due to intrusive instrumentations. The answer for **RQ2** is, that our newly introduced quality assessment criterion is applicable to compute quality differences of different test suites under restricted conditions. If the number of possible outcomes of a program under test is higher than the number of test cases in a test suite whose quality should be assessed the criterion does not provide meaningful results. In future work we extend our empirical evaluation considering more examples from an application domain, i.e., automotive control software. Here the open research question is, whether generating a test suite with higher strength and reducing it e.g. by test suite reduction as introduced in Chapter 8 provides higher quality test suites, than t-way combinatorial test suites generated with lower strength. Also we will investigate the applicability of our model inference approach for test suite prioritization, where an overview of current approaches is given in [101].

Part IV.

Model Inference Based Test Suite Reduction

8. Test Suite Reduction Does Not Necessarily Require Executing The Program Under Test

This chapter is based on the work "Test-Suite Reduction Does Not Necessarily Require Executing the Program under Test" [9].

Removing redundancies from test suites is an important task of software testing in order to keep test suites as small as possible, but not to harm the test suite's fault detection capabilities. A straightforward algorithm for test suite reduction would select elements of the test suite randomly and remove them if and only if the reduced test suite fulfills the same or similar coverage or mutation score. Such algorithms rely on the execution of the program and the repeated computation of coverage or mutation score.

When systems and programs evolve over time, their corresponding test suites become bigger and bigger causing an increase of computation time needed to execute the test suite. In addition, over time some elements of the test suite might lead to a redundancy within the test suite [102]. In order to optimize the test suite, redundant elements have to be eliminated. Such a redundancy elimination can be performed based on analyzing the program and identifying test cases leading to execution traces that are covered by execution traces of other tests (e.g., [103], [29]). Alternatively, we are able to remove tests that do not change the obtained coverage or mutation score value (e.g., [26–28, 104–107]). All these reduction methods have in common that they require the program under test to be repeatedly executed. Depending on the computational complexity and the underlying

test suite, such reduction can take a longer time. Hence, leading to the question whether there are faster approaches for redundancy elimination. Answering this question with yes, requires to introduce such a method, which does not rely on program execution or analysis for test suite reduction. One idea going in this direction would be to replace the program with an appropriate model, which would require additional effort if done manually (e.g., [108]). At this point we bring in another idea. Every test suite should at least partially capture the behavior of the program under test in a sufficient way. Hence, why not using the test suite itself to obtain the required model? For model extraction we make use of available machine learning techniques. In this way, we are able to automate the whole test suite reduction approach and furthermore do not require the program to be executed. The proposed test suite reduction approach now works as the following process: We obtain a model from the original test suite. Then we successively select test cases from the test suite, remove them, and learn again a model from the reduced test suite. In case the model of the original test suite and the new model are equivalent, the reduced test suite should have the same capabilities as the original one and we again repeat this process. Otherwise, the test case is added again to the current test suite. The process stops in case no test case can be removed without changing the model. The most important question of course is **RQ1:** Is the proposed test suite reduction approach able to reduce a test suite without compromising coverage or mutation score substantially? In order to answer this question we have to evaluate the approach. For this purpose, we introduce an instantiation of the approach that relies on decision trees as models and decision tree inference as their corresponding machine learning technique where we use the C4.5 algorithm [51] for inference. Of course there are many other machine learning techniques that can be used. We selected decision trees, because of the underlying application domain of embedded systems we had in mind. There the programs often implement state machines using state transfer functions, which are more or less represented using nested conditional statements.

We further on present the test suite reduction algorithm and provide results from the empirical evaluation of the algorithm also in comparison with traditional test suite reduction based on coverage and mutation score. The empirical evaluation is based on 7 example programs. These examples

and examples of similar size (size in lines of code) were already used to evaluate model learning, test suite reduction, and test case generation approaches (e.g., [56], [64], [109], [59], [9]). The obtained results of our model learning based reduction approach are very promising, showing substantial reductions and almost no decline of coverage and mutation score. E.g. in [110] the authors also reduced the TCAS test suite and obtained average reductions of 76.87% and 86.88%. But they also reported a huge decline of the mutation score of 55.1% and 63.95%. As shown in Section 8.3, we obtain for the TCAS test suite a reduction of up to 70.87% with a max. mutation score decline of 7.32%. Compared with traditional test suite reduction, we are able to compute the reduced test suites in a fraction of time which is shown empirically to answer **RQ2:** Is our model learning based test suite reduction approach more efficient regarding size and execution time than a coverage and mutation score based reduction approach?

In Example 1 we provide an introductory example of a simple test suite T containing 4 test cases, and show different decision trees inferred from T and subsets of T.

Example 1 (Introductory example) *Table 8.1 represents an example test suite for the Boolean expression* $(x > 0) \vee (y > 0)$. *A decision tree inferred from this test suite is shown in Figure 8.1. The decision tree inferred from the test suite in Table 8.1 excluding the test case with index #4 is the same decision tree as inferred from all existing test cases. The test suite used to infer the decision tree from Figure 8.2 was the test suite from Table 8.1 excluding the test cases with indexes #3 and #4.*

Table 8.1.: Test suite for a Boolean expression with two input variables x and y of type Integer, Boolean output, and # as index of the test cases.

#	x	y	out
1	0	0	F
2	1	0	T
3	0	1	T
4	1	1	T

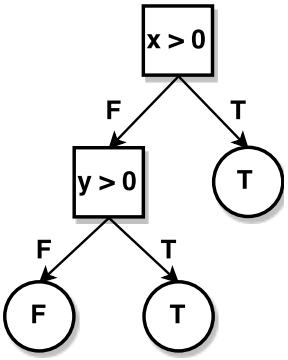

Figure 8.1.: Decision tree inferred from all test cases in Table 8.1 and from a subset containing only three of these test cases.

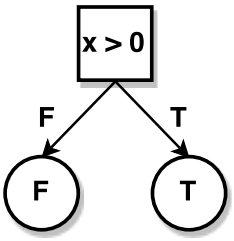

Figure 8.2.: Decision tree inferred from 2 of 4 test cases in Table 8.1 excluding test case #3 and #4.

8.1. Basic Definitions

In our approach we reduce test suites T, containing test cases, to subsets of these test cases. We consider a test case to be either a test vector tc of k input values and an expected output value, e.g., $tc = (in_1, .., in_k, out)$ or a sequence ts of n test vectors, e.g., $ts = \langle tc_1, .., tc_n \rangle$. A test case can either fail or pass. A test case passes if the expected outcomes from the vectors are equivalent to the outputs of the program under test after it was executed with the input values from the test vectors, otherwise it fails. The input types can be either discrete or numeric. Outputs are discrete. From such a test suite we infer a decision tree.
In this chapter we provide a new test suite reduction approach where we define reduction as follows:

Definition 14 (Reduction) *Given a test suite T and a reduced test suite $T' \subseteq T$, reduction is the difference in size of the two test suites, given in %, and calculated as:*

$$Reduction = (\frac{|T|-|T'|}{|T|}100).$$

8.1.1. Decision tree inference

Also in this chapter for test suite reduction we utilize the widely used C4.5 algorithm [51] to infer a decision tree (V, E) from a test suite T. A decision tree is a tree (V, E) having nodes V and edges E between nodes with the usual restrictions applying to trees. The tree has decision nodes, and leaf nodes. Each decision node represents a decision (i.e. a relational equation), e.g., $x > 0$ for numeric inputs or x *equals* "*open*" for discrete inputs. A leaf node represents a classification, which is one of the discrete output values. E.g., the decision tree shown in Figure 8.1 has two decision nodes $x > 0$ and $y > 0$, and three leaf nodes F, T, T. The root node of the tree is a decision node having only outgoing edges.

8.2. Test Suite Reduction Approach

Our reduction approach is based on changes in the test suite that do not cause changes in the learned model. After each reduction we learn a new model from the reduced test suite. Whenever a reduction causes a change in the learned model, in comparison to the model we had before the reduction, we detect the change. In order to detect these changes, we have to provide a notion of equivalence for the learned models, i.e., the decision trees. In this chapter, we define two methods to check for equivalence, where one model is learned from the reduced test suite T', and the other, learned from the original test suite T, serves as reference model. The first method determines syntactic equivalence of two models and the second method is based on a misclassification rate.

For simplicity, we assume a function ρ: $DT \rightarrow V$ with the universe of decision trees DT under consideration as input domain that returns the root node of a decision tree. E.g. the root node of the decision tree shown in Figure 8.1 is the decision node $x > 0$. We assume a function λ: $V \rightarrow D \cup C$ with the union of the set of decisions D and the set of classifications C as range that returns the content of a node. The content of a node is either a decision for decision nodes, or a classification for leaf nodes. E.g., the union of $D \cup C$ for the decision tree in Figure 8.1 is $\{x > 0, y > 0\} \cup \{F, T, T\}$. Because we

infer binary decision trees, the answer of a decision, e.g., whether $x > 0$ or not, is represented by a label of an outgoing edge from the decision node that can be accessed via the γ: $E \rightarrow \{True, False\}$ function. E.g. the outgoing edges of the root node from the decision tree in Figure 8.1 are labeled with T and F (short for *True* and *False*).

8.2.1. Syntactic Equivalence

Two decision trees are syntactically equivalent if and only if each node in a decision tree has a corresponding node in the other decision tree, representing the identical content and each edge in the decision tree has a corresponding edge in the other decision tree with the same label and connecting the same pair of nodes. This form of equivalence can be represented using a function EQUAL: $V \times V \rightarrow \{True, False\}$, which we define recursively as follows: Given two decision trees $(V_1, E_1) \in DT$ and $(V_2, E_2) \in DT$, and nodes $n_1, m_1 \in V_1$ and $n_2, m_2 \in V_2$, EQUAL returns *True*, if and only if:

1. $\lambda(n_1) = \lambda(n_2)$ (The content of the corresponding nodes has to be equivalent)
2. $|\{(n_1, m_1) | m_1 \text{ is a direct successor of } n_1\}| = |\{(n_2, m_2) | m_2 \text{ is a direct successor of } n_2\}|$ (The number of outgoing edges has to be equivalent)
3. $\forall(n_1, m_1)\exists(n_2, m_2)[m_1 \text{ is a direct successor of } n_1 \wedge m_2 \text{ is a direct successor of } n_2 \wedge \gamma(n_1, m_1) = \gamma(n_2, m_2) \wedge \text{EQUAL}(m_1, m_2)]$ (The corresponding outgoing edges and the sub-decision trees have to be equivalent)

Otherwise, EQUAL returns *False*. Using this function, we define syntactic equivalence of two decision trees as follows:

Definition 15 (Syntactic equivalence) *Given two decision trees* (V_1, E_1) *and* (V_2, E_2)*, these decision trees are* syntactically equivalent *if and only if the function* $\text{EQUAL}(\rho(V_1, E_1), \rho(V_2, E_2))$ *returns True.*

8.2.2. Equivalence Based on a Misclassification Rate

Two decision trees, a reference decision tree (V_1, E_1) inferred from a test suite T, and a decision tree (V_2, E_2) inferred from a test suite T' are equivalent

regarding their misclassification rate, if the following two conditions hold: First, the misclassification rate of (V_2, E_2), when evaluating T (therefore the reduced test cases serve as test data for the machine learning algorithm), is less than or equal to the misclassification rate of (V_1, E_1). Second, for all distinct classifications which exist in (V_1, E_1) an equivalent classification exists in (V_2, E_2). The misclassification rate is defined as:

Definition 16 (Misclassification rate) *The misclassification rate* MR *of a decision tree* (V, E) *is the ratio of the number of incorrectly classified test cases* $TF \subseteq T$ *to the number of all classified test cases* T*.*

$$MR(V, E) = \frac{|TF|}{|T|} \tag{8.1}$$

In the remainder of this work equivalence based on the misclassification rate is named semantic equivalence. Thus we define semantic equivalence as follows:

Definition 17 (Semantic equivalence) *A decision tree* (V_2, E_2) *is semantically equivalent to a reference decision tree* (V_1, E_1) *when classifying a test suite* T*, if the following equation holds:*

$$\begin{gathered} MR(V_2, E_2) \leq MR(V_1, E_1) \wedge \\ \forall v \exists w [v \in \mathit{Classifications}(V_1, E_1) \wedge \\ w \in \mathit{Classifications}(V_2, E_2) \wedge v = w] \end{gathered}$$

8.2.3. Test Suite Reduction

The algorithm Reduce for reducing a test suite without executing the program under test is shown in Algorithm 7. As a result the algorithm yields a reduced test suite T'. Reduce requires three inputs, i.e., a test suite T, an integer number *iterations* > 0, and an integer number *retries* > 0. T is a test suite containing all initially given test cases, *iterations* declares how often the function Reduce is restarted, and *retries* is the maximum number of attempts, where a test case is removed from T randomly, until a

decision tree is inferred, which is equivalent to the decision tree inferred from T. The decision trees are inferred in function `inferDecisionTree`(T) using the C4.5 algorithm as explained in Section 8.1. Equivalence of the models in REDUCE is checked either syntactically or semantically (set by configuration) as explained in Sections 8.2.1 and 8.2.2. During execution the function REDUCE finds a subset tT' of test cases from T for each iteration. After each iteration of REDUCE, the size of tT' is compared to the size of T'. If the size of tT' is smaller than the size of T', tT' is assigned to T'. This ensures that the algorithm returns the smallest reduced test suite which was found, but REDUCE does not guarantee to find a reduced test suite.

8.3. Experimental Results and Evaluation

In this Section we provide experimental results to answer the research questions posed for this chapter.

8.3.1. Example Programs

We evaluated our approach on seven different example programs which are:

1. **TCAS:** See Section 5.2
2. **BMI:** See Section 5.2
3. **Triangle:** See Section 5.2
4. **POP3:** See Section 5.2
5. **Car Alarm System (CAS):** See Section 5.2
6. **Guava UTF8 (UTF8):** See Section 5.2
7. **Cruise Control**[1] **(CC):** Simulates a car and its cruising controller.

The attributes source lines of code (*SLOC*), number of distinct discrete output values (*classifications*), size of the test suite T ($|T|$), and number of mutants (s) for the seven examples are given in Table 8.2. The size of a test suite is the number of test cases it contains.

[1] `http://sir.unl.edu/portal/bios/Cruise%20Control.php`

Algorithm 7 Function to reduce a test suite T.

```
1: function REDUCE(T, iterations, retries)
2:     T' = T
3:     DT = inferDecisionTree(T)
4:     for all i in range(0, iterations) do
5:         tT' = T'
6:         while true do
7:             found = False
8:             for all j in range(0, retries) do
9:                 t = getRandomTestCase(T')
10:                tT' = tT'\t
11:                DT2 = inferDecisionTree(tT')
12:                if DT equals DT2 then
13:                    found = True
14:                    break
15:                else
16:                    tT' = tT' ∪ t
17:                end if
18:            end for
19:            if !found then
20:                break
21:            end if
22:        end while
23:        if |tT'| < |T'| then
24:            T' = tT'
25:        end if
26:    end for
27:    return T'
28: end function
```

Table 8.2.: Attributes of the example programs.

Name	*SLOC*	*classifications*	$\|T\|$	s
TCAS	100	3	1,545	41
BMI	19	5	1,000	28
Triangle	30	4	1,000	35
POP3	122	10	1,000	167
CAS	110	5	1,000	167
UTF8	56	2	1,000	147
CC	261	4	1,000	363

8.3.2. Tools

In this work we utilized different existing software tools to implement and evaluate our test suite reduction approach. The different tools are ScalaCheck[2] for test case generation, Weka[3] for model learning, CodeCover[4] to measure the coverage of a test-suite, and the Major mutation framework[5] to generate mutants with the exception of TCAS and CC where we rely on the existing mutants.

8.3.3. Reductions With Proposed Equivalence Check Methods

The test suites of the seven example programs were reduced with a Java implementation of REDUCE. This reduction was executed using four different configurations for decision tree inference and the equivalence check. Two configurations utilized syntactical equivalence with first requiring the inference to classify a minimum of *one test case per leaf node (min 1)* and second a minimum of *two test cases per leaf node (min 2)*. The other two configurations were also first with a minimum of one test case per leaf node and second with two test cases, but with semantic equivalence. We used the values

[2] www.scalacheck.org
[3] www.cs.waikato.ac.nz/ml/weka/
[4] www.codecover.org
[5] www.mutation-testing.org

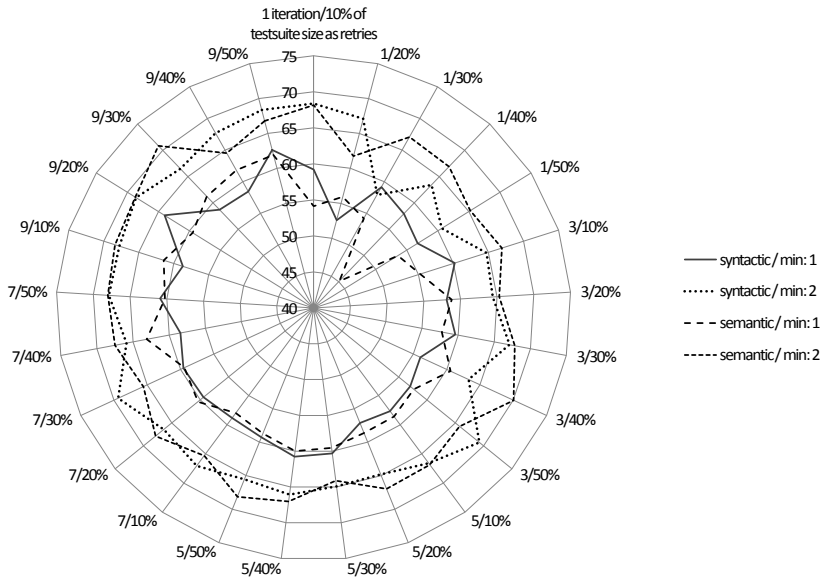

Figure 8.3.: *Reduction* of T for the TCAS example for four different configurations and 25 executions of REDUCE for each configuration.

$L = \{1, 3, 5, 7, 9\}$ for *iterations* and $M = \{10, 20, 30, 40, 50\}$ for *retries* where the values for *retries* are values in % relative to $|T|$, e.g., if $|T| = 1000$ and $r \in M = 10\%$ then $\textit{retries} = \frac{|T| * r}{100} = 100$. Each of the four configurations was executed once with each of the 25 value pairs in $L \times M$.

The first results were obtained by executing REDUCE for T of the TCAS example. The results for *Reduction* are shown in Figure 8.3. These results show that $\textit{Reduction} > 60\%$ is possible for each of the four different configurations.

Figure 8.4 shows the 25 executions for each of the four different configurations for T of the BMI example. For this example results of $\textit{Reduction} > 80\%$ were obtained. In Figure 8.5 the results for T of the Triangle example are shown. For the Triangle example results of $\textit{Reduction} > 70\%$ were achieved.

The results for the two state machine examples, namely POP3 and CAS, are very similar. Figure 8.6 shows the results for T of the POP3 example and Figure 8.7 shows the results for T of the CAS example. For both state

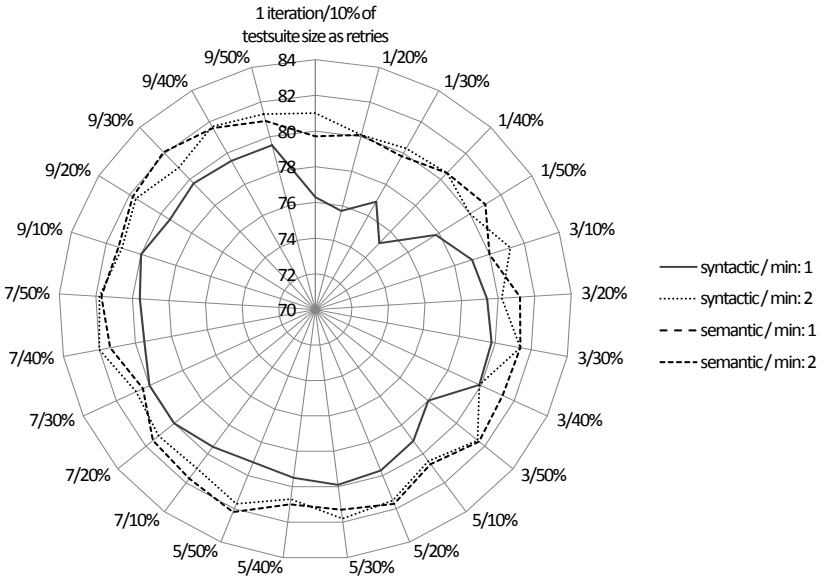

Figure 8.4.: *Reduction* of T for the BMI example for four different configurations and 25 executions of Reduce for each configuration.

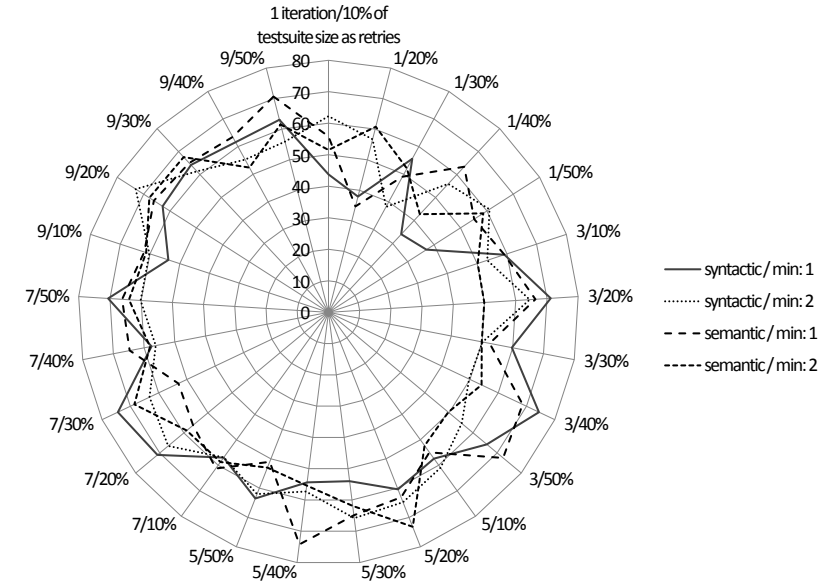

Figure 8.5.: *Reduction* of T for the Triangle example for four different configurations and 25 executions of Reduce for each configuration.

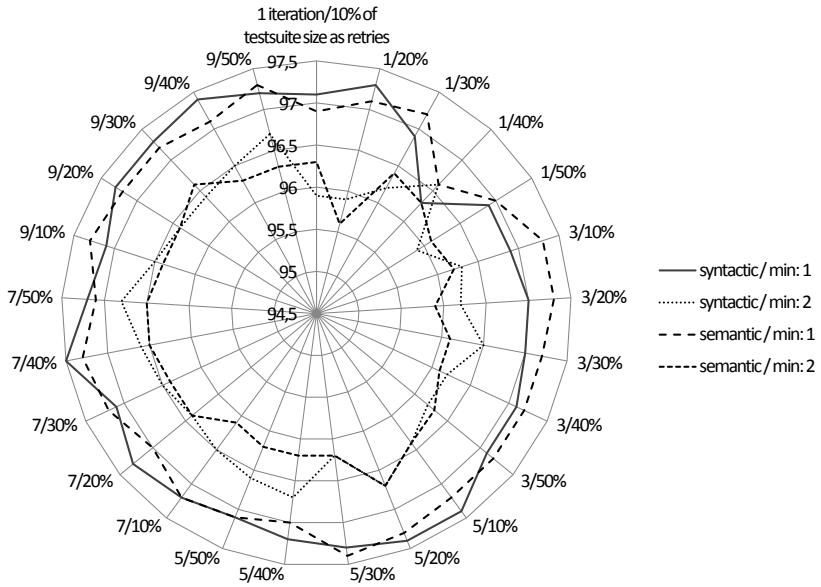

Figure 8.6.: *Reduction* of T for the POP3 example for four different configurations and 25 executions of Reduce for each configuration.

machine examples we obtained results of *Reduction* $> 96\%$ for each of the four configurations. The results for T of the UTF8 example are shown in Figure 8.8. Results of *Reduction* $> 90\%$ are possible for each of the four configurations.

The results for T of the CC example are shown in Figure 8.9. Results of *Reduction* $> 99\%$ are possible for each of the four configurations.

An overview of the results for *Reduction* from the seven examples is given in Tables 8.3 and 8.4. Table 8.3 shows the *min.*, *max.*, and *avg.* value of *Reduction* for each example, which were obtained by the 100 executions of Reduce (4 configurations * 25 value pairs for *iterations* and *retries*).

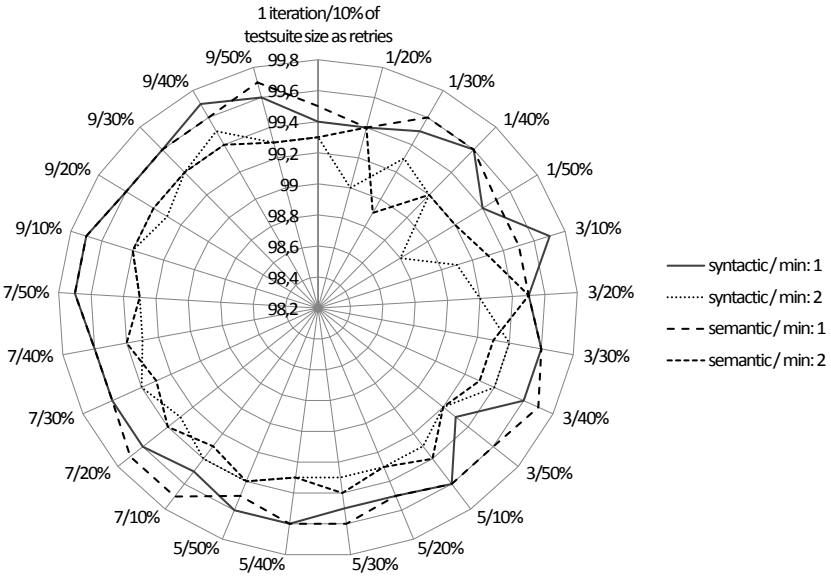

Figure 8.7.: *Reduction* of T for the CAS example for four different configurations and 25 executions of Reduce for each configuration.

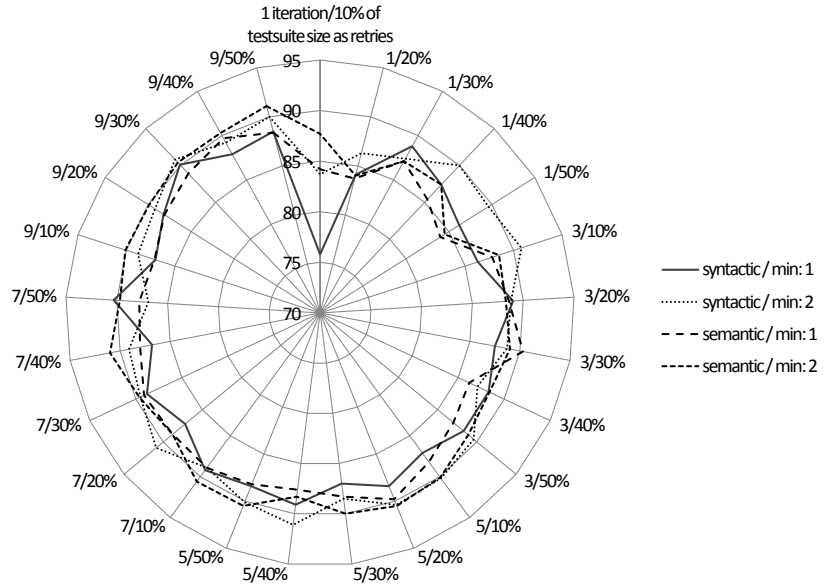

Figure 8.8.: *Reduction* of T for the UTF8 example for four different configurations and 25 executions of Reduce for each configuration.

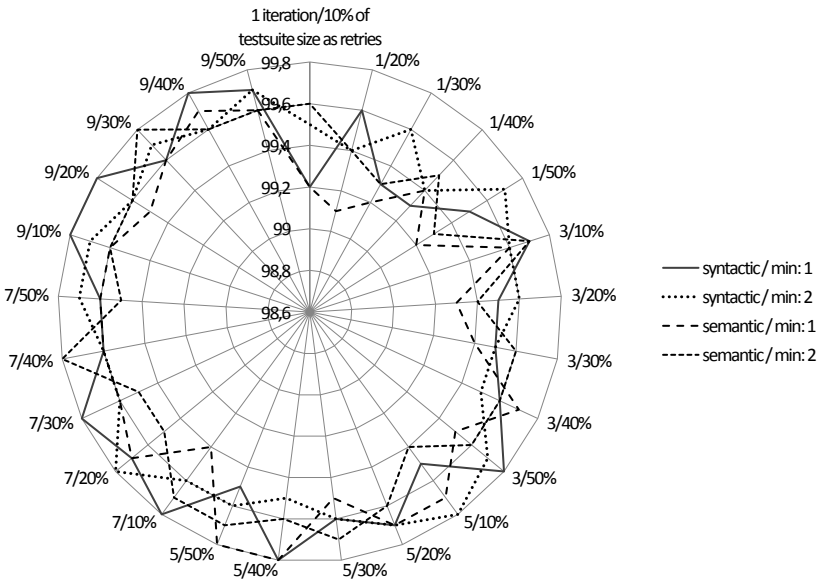

Figure 8.9.: *Reduction* of T for the CC example for four different configurations and 25 executions of REDUCE for each configuration.

Table 8.3.: *Reductions* with proposed equivalence check methods.

	Reduction (%)		
Name	*min.*	*max.*	*avg.*
TCAS	45.11	70.87	62.49
BMI	75.10	82.20	80.08
Triangle	34.00	74.40	58.79
POP3	95.60	97.50	96.74
CAS	98.80	99.70	99.44
UTF8	75.80	91.10	88.27
CC	99.10	99.80	99.58

Table 8.4.: *Reductions* with proposed equivalence check methods.

	Reduction (%)					
	syntactic			*semantic*		
Name	*min.*	*max.*	*avg.*	*min.*	*max.*	*avg.*
TCAS	52.56	69.39	62.43	45.11	70.87	62.54
BMI	75.10	82.00	79.95	77.20	82.20	80.20
Triangle	34.00	74.40	58.62	34.70	74.20	58.96
POP3	95.90	97.50	96.77	95.60	97.40	96.72
CAS	98.80	99.70	99.42	98.90	99.70	99.46
UTF8	75.80	91.10	88.23	83.70	91.10	88.32
CC	99.20	99.80	99.61	99.10	99.80	99.54
	min. 1 *per leaf node*			*min.* 2 *per leaf node*		
Name	*min.*	*max.*	*avg.*	*min.*	*max.*	*avg.*
TCAS	45.11	64.01	58.54	57.93	70.87	66.43
BMI	75.10	80.80	79.08	79.70	82.20	81.07
Triangle	34.00	74.40	59.56	38.20	73.20	58.02
POP3	96.30	97.50	97.15	95.60	96.80	96.33
CAS	99.30	99.70	99.58	98.80	99.50	99.30
UTF8	75.80	90.30	87.54	83.70	91.10	89.00
CC	99.10	99.80	99.57	99.30	99.80	99.59

8.3.4. Test Suite Reduction Results Using Syntactic Equivalence

In this section, to obtain a reduced test suite T' we executed REDUCE with different value pairs for *iterations* and *retries*, and syntactic equivalence. After some initial experiments we used the values $L = \{1, 2, 3, 4, 5\}$ for *iterations* and as in Section 8.3.3 with $M = \{10, 20, 30, 40, 50\}$ for *retries* which are values in % relative to $|T|$. We executed each pair in $L \times M$ (e.g., *iterations*=1, *retries*=10% of $|T|$) 25 times.

Figure 8.10 shows the results for *Reduction* of the TCAS example. These reductions cut the original test suite by around 60%. The box plots in Figure 8.10 are grouped by the five different values for *iterations*. These results show that for an increasing number of *iterations* the rectangles and whiskers in the plot become narrower and the median reduction value only slightly rises for an increasing number of *retries*. An example test vector $tc_1 \in T$ for the TCAS example is the vector
$tc_1 = (958, 1, 1, 2597, 574, 4253, 0, 399, 400, 0, 0, 1, unresolved)$
with the last value (*unresolved*) as expected outcome and the remaining values as inputs. In Figure 8.11 the results for *Reduction* of the Triangle example test suite are shown. The results are similar to the results of the TCAS example at around 60%. Also the different values for *iterations* and *retries* similarly affect the results. An example test vector $tc_2 \in T$ for the Triangle example from which the decision trees during execution of REDUCE are inferred is the vector $tc_2 = (64, 64, 64, equilateral)$ with the last value (*equilateral*) as expected outcome and the remaining values as inputs. An example test vector $tc_3 \in T$ for the BMI example from which the decision trees during execution of REDUCE are inferred is the vector $tc_3 = (1.717075, 183.332056, veryobese)$ with the last value (*veryobese*) as expected outcome.

The results for *Reduction* of the BMI example are shown in Figure 8.12. The results for *Reduction* are around 80%. Also for this example the increasing values for *iterations* cause the rectangles in the plot to become narrower and for increasing values of *retries Reduction* slightly rises.

For the UTF8 example the results of *Reduction* are shown in Figure 8.13. The results for *Reduction* of this example test suite are around 87%. Increasing

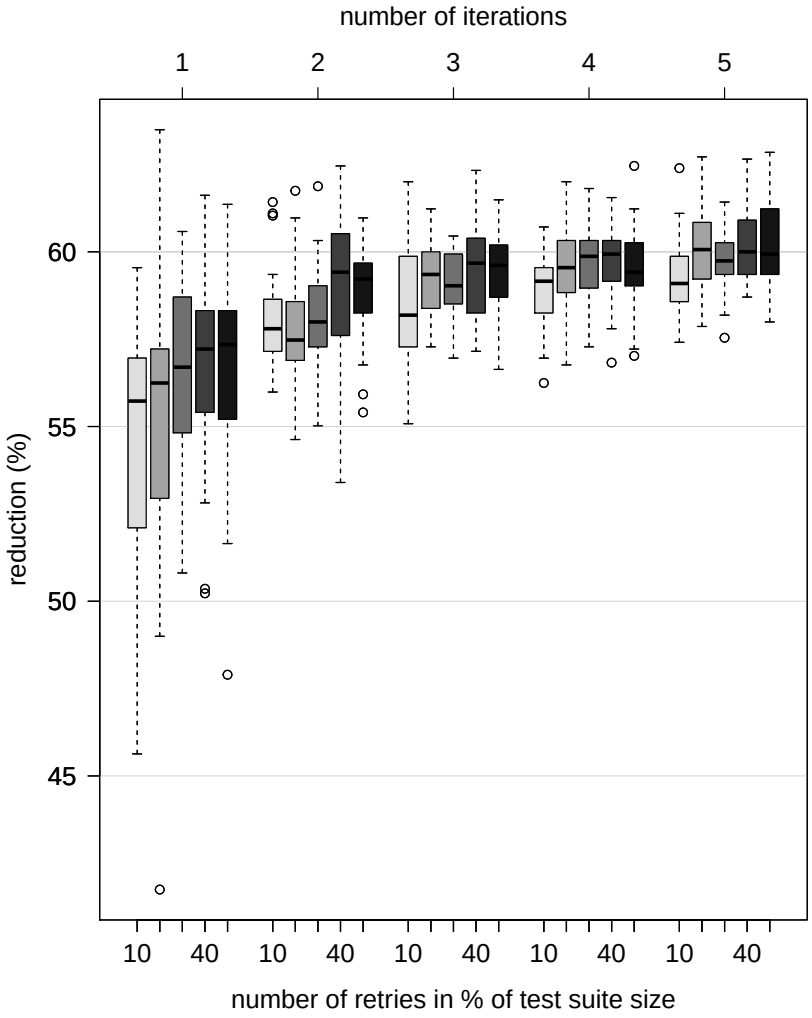

Figure 8.10.: *Reduction* results for the TCAS example.

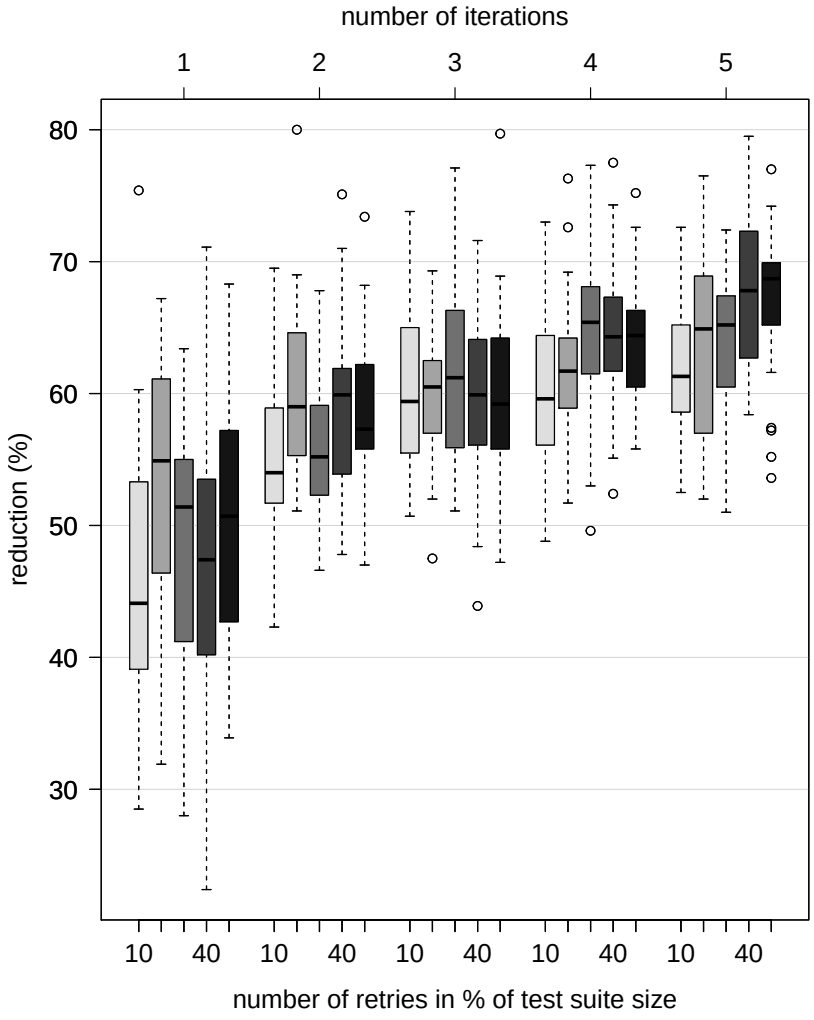

Figure 8.11.: *Reduction* results for the Triangle example.

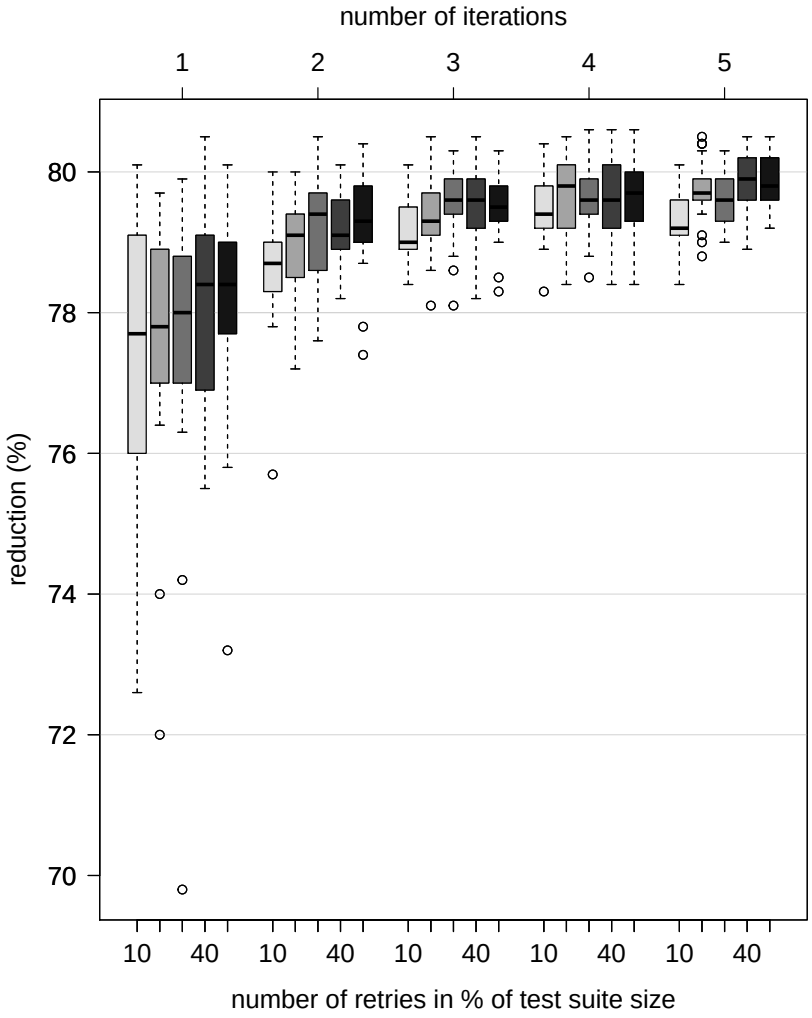

Figure 8.12.: *Reduction* results for the BMI example.

values for *iterations* affect the rectangles in the plot to become narrower and increasing values for *retries* affect slightly risings of *Reduction*. An example test vector $tc_4 \in T$ for the UTF8 example from which the decision trees during execution of REDUCE are inferred, is the vector $tc_4 = (0, ?, ?, ?, True)$ with the last value (*True*) as expected outcome and the remaining values as inputs. An input for the UTF8 example consists of up to four bytes. In tc_4 the input has a length of only one byte but, as the expected outcome indicates, is a valid UTF8 sequence. Decision tree inference can be used even when some input values have unknown values which are given as ? in tc_4. It is common to estimate missing input values for decision tree inference. For estimation either the value that is most common among the test vectors in T for the input variable is used, or probabilities are estimated based on the observed frequencies of the various values for the input variable from which a value is derived.

Figure 8.14 shows the results for *Reduction* of the POP3 example. The reductions cut the original test suite by around 97%. For this example, with increasing values for *iterations*, the rectangles become narrower, but increasing values for *retries* hardly affect the results. For the POP3 example a test case is a sequence $ts = \langle tc_1, .., tc_n \rangle$ of n test vectors. As input for decision tree inference we use the test vectors $tc_1, .., tc_n$ from all $ts \in T$. An example test vector $tc_5 \in ts$ for the POP3 example, from which the decision trees during execution of REDUCE are inferred, is the vector $tc_5 = (USER(0), off, 1, passr)$, where the first value is the input value, the second value is the expected outcome after executing the last test vector from the current test case ts, the third value is a label representing the current state of the program under test, and the fourth value is the expected outcome after executing tc_5. The decision trees are inferred from the test vectors of all remaining test cases $ts \in T'$, but REDUCE removes entire test cases.

In Figure 8.15 the results for *Reduction* of T from the CAS example are shown. The results for this example test suite are around 99.6%. Also for this example the increasing values for *iterations* cause the rectangles in the plot to become narrower but increasing values for *retries* hardly affect the results. For the CAS example a test case is a sequence $ts = \langle tc_1, .., tc_n \rangle$ of n test vectors. An example test vector $tc_6 \in ts$, used as input for decision tree inference, is $tc_6 = (opendoor, armed, PCAlarm, activated\_flash\_alarm)$. The

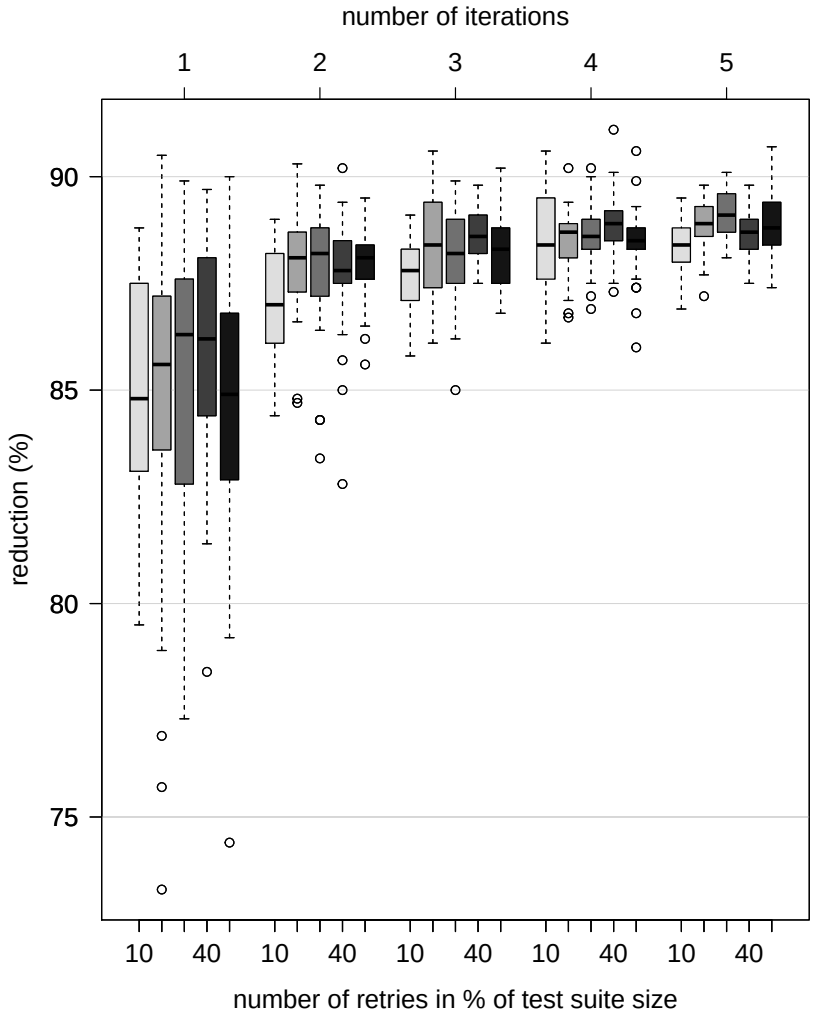

Figure 8.13.: *Reduction* results for the UTF8 example.

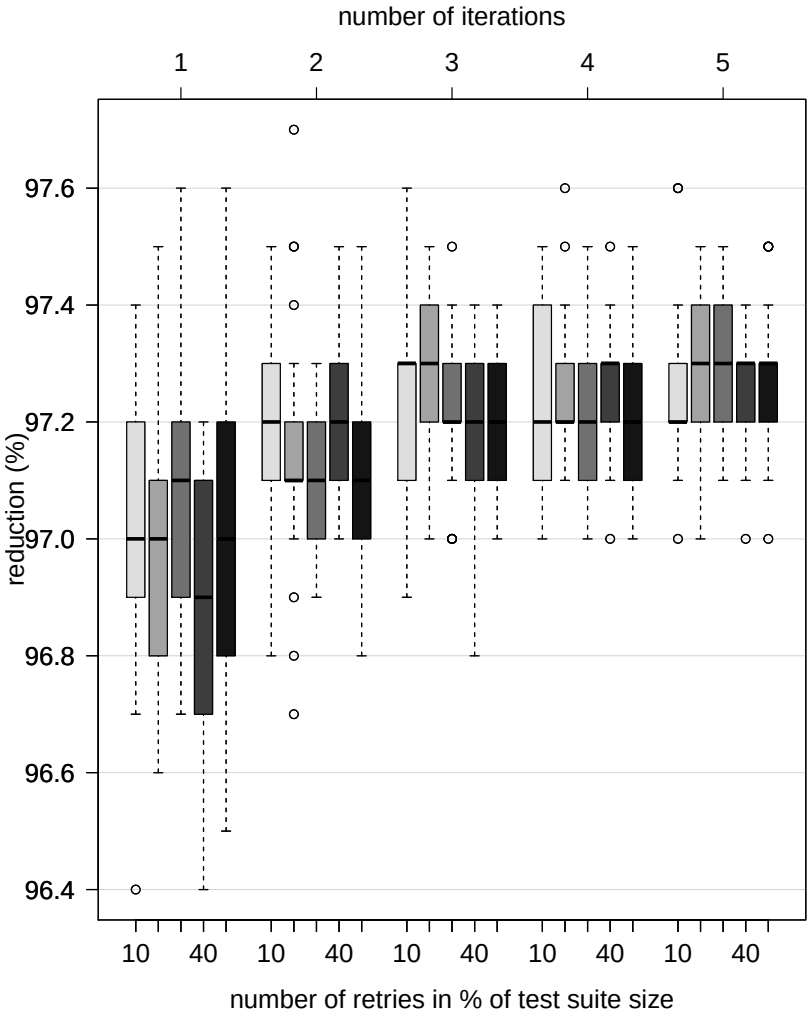

Figure 8.14.: *Reduction* results for the POP3 example.

properties of the values in the test vectors are the same as explained for the POP3 example.

Figure 8.16 shows the results for *Reduction* of the CC example. The reductions cut the original test suite by around 99.5%. For this example, with increasing values for *iterations*, the rectangles become narrower, but increasing values for *retries* hardly affect the results. For the CC example a test case is a sequence $ts = \langle tc_1, .., tc_n \rangle$ of n test vectors. As input for decision tree inference we use the test vectors $tc_1, .., tc_n$ from all $ts \in T$. An example test vector $tc_7 \in ts$ for the CC example, from which the decision trees during execution of REDUCE are inferred, is the vector $tc_7 = (accelerator, true, 0, 1)$, where the first value is the input value, the second value is *true*, if the speed of the car is the equivalent to the speed after executing the last test vector from the current test case *ts*. The third value represents the current state of the speed control, and the fourth value is the current state of the Cruise Control.

8.3.5. Evaluation

To evaluate our results we created an adapted version of REDUCE, namely CMREDUCE, where we used coverage and mutation score for the equivalence check. As shown in Algorithm 8 decision tree inference from REDUCE is replaced by a function to obtain coverage and mutation score in CMREDUCE. CMREDUCE uses the tool CodeCover and the mutants as listed in Table 8.2 and returns the two values MC/DC coverage[6] and mutation score for the provided test suite. Coverage and mutation score of a test suite are equivalent to coverage and mutation score of a different test suite if both values are equivalent.

Table 8.5 shows the differences of *Reduction* from using REDUCE to using CMREDUCE. The execution time to obtain the results for *Reduction* is given in Table 8.6. The values in Tables 8.5 and 8.6 are the *min.*, *max.*, and *avg.* values of the results obtained in Section 8.3.4 and the results of executing CMREDUCE with *iterations* $= 1$ and *retries* $= 10\%$ of $|T|$. The results in Table

[6]The applied tool to measure coverage, namely CodeCover, implements the Ludewig term coverage, which subsumes MC/DC for Boolean short circuit semantics.

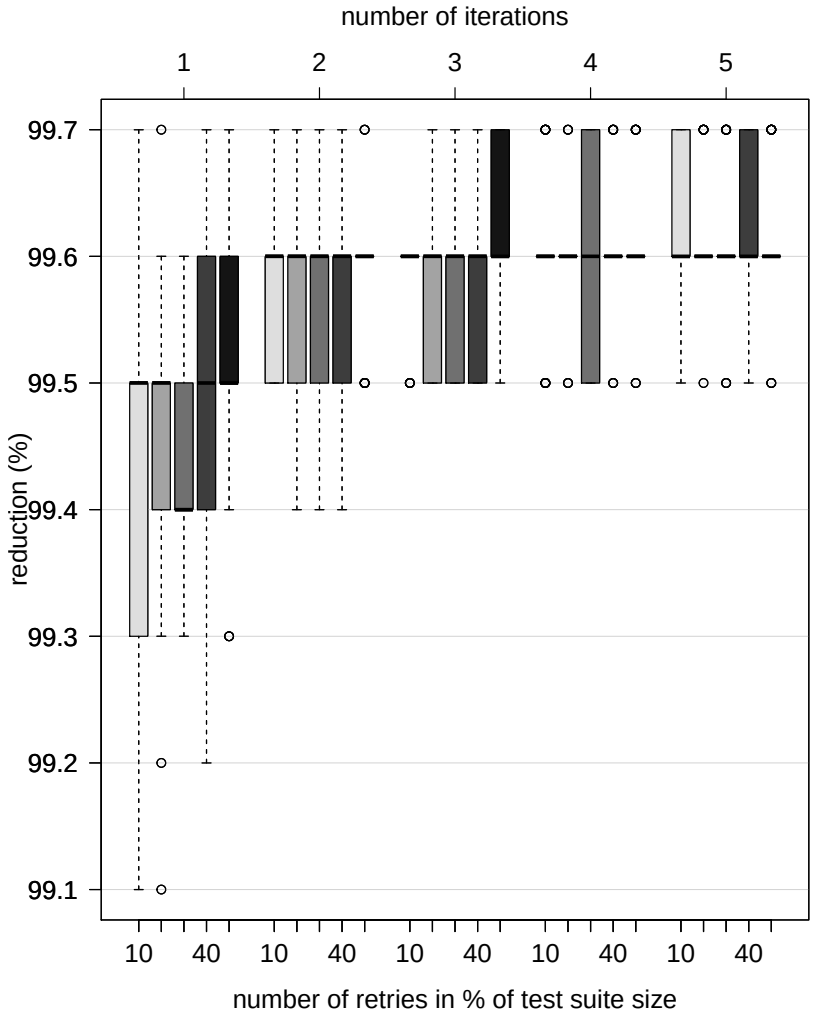

Figure 8.15.: *Reduction* results for the CAS example.

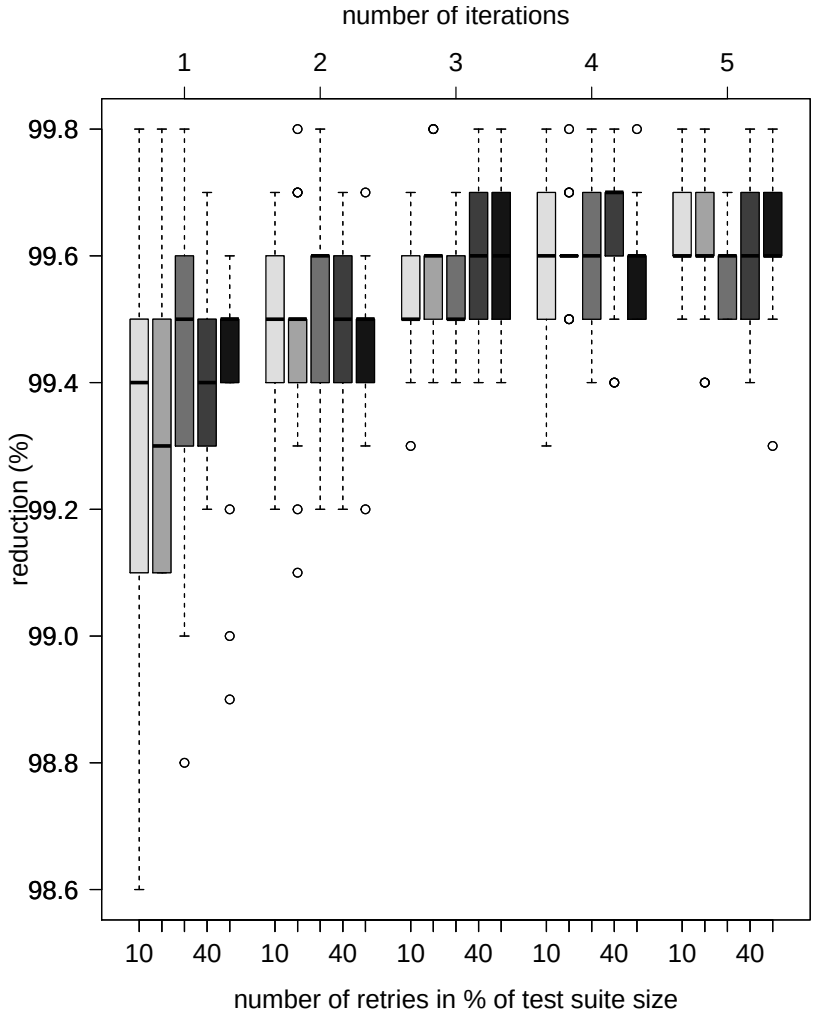

Figure 8.16.: *Reduction* results for the CC example.

Algorithm 8 Function to reduce a test suite T by coverage and mutation score.

```
1:  function CMReduce(T, iterations, retries)
2:      T' = T
3:      COV = getCovAndMS(T)
4:      for all i in range(0, iterations) do
5:          tT' = T
6:          while true do
7:              found = False
8:              for all j in range(0, retries) do
9:                  t = getRandomTestCase(tT')
10:                 tT' = tT'\t
11:                 COV2 = getCovAndMS(tT')
12:                 if COV equals COV2 then
13:                     found = True
14:                 else
15:                     tT' = tT' ∪ t
16:                 end if
17:             end for
18:             if !found then
19:                 break
20:             end if
21:         end while
22:         if |tT'| < |T'| then
23:             T' = tT'
24:         end if
25:     end for
26:     return T'
27: end function
```

Table 8.5.: Model learning based *Reduction* results (ml based) vs coverage+mutation score based *Reduction* (cm) results.

	Reduction (%)			
	ml based			cm
Name	*min.*	*max.*	*avg.*	
TCAS	41.75	63.50	58.54	98.9
BMI	69.80	80.60	79.10	99.5
Triangle	22.40	80.00	59.19	99.3
POP3	96.40	97.70	97.17	99.4
CAS	99.10	99.70	99.57	99.6
UTF8	73.30	91.10	87.66	94.4
CC	98.00	99.80	99.50	99.5

8.6 show that REDUCE is multiple times faster than CMREDUCE. Further the results in Table 8.5 show that the results of *Reduction* differ by up to 76% for test suites, where a test case is a single vector, but for test suites where a test case is a sequence of vectors the maximum difference is only 3%. Knowing these initial results we answer **RQ2** as follows: The model learning based test suite reduction approach is not more efficient regarding size than the coverage and mutation score based reduction approach, but for some examples the results are almost equivalent. Regarding execution time the model learning based reduction approach is multiple times faster. That is because for our approach it is not necessary to execute the program under test.

We further evaluated our results of *Reduction* against a random reduction. For this random reduction we took the *avg*. *Reduction* of our results from Section 8.3.4 and cut this size from the original test suite T of each example by randomly selected test cases. To get a representative result we executed this random reductions 25 times. These evaluation results are shown in Table 8.7. Table 8.7 shows the *min*., *max*., and *avg*. values for statement coverage, decision coverage, MC/DC coverage, and mutation score (μ) for the test suites T'_r and the test suites T'_{ml}. We obtained T'_r by random reduction and gained T'_{ml} in Section 8.3.4 (*ml based*). The results in Table 8.7 show that for the TCAS example the coverage results are not different for T'_r and T'_{ml}. A difference of the mutation score for the TCAS example

Table 8.6.: Reduction time to obtain *Reduction* results with model learning based (ml based) and coverage+mutation score based (cm) methods.

	reduction time (sec)			
	ml based			cm
Name	*min.*	*max.*	*avg.*	
TCAS	21.2	654.0	212.6	3692
BMI	5.2	81.9	28.5	1856
Triangle	9.7	206.8	69.9	1916
POP3	5.1	58.1	24.0	5708
CAS	7.0	81.1	35.0	188,925
UTF8	2.7	28.9	11.3	2809
CC	6.4	53.5	20.5	34,221

is shown where the mutation score for T'_{ml} is higher. For the Triangle example and the BMI example coverage and mutation score of T'_r and T'_{ml} are equivalent to the coverage and mutation score of T. Mutation score of the POP3 example is for both, T'_r and T'_{ml}, equivalent to the mutation score of T. There are almost no differences of statement coverage, decision coverage, and MC/DC coverage for T'_{ml} to these values of T. For T'_r the coverage results are considerably lower. The CAS example shows clearly that coverage and mutation score of T'_{ml} are higher than for T'_r. The UTF8 example provides higher values for statement coverage for T'_r than for T'_{ml}. Decision coverage, MC/DC coverage, and mutation score are higher for T'_{ml}. The CC example shows that coverage and mutation score of T'_{ml} are higher than for T'_r. Knowing these results we can answer **RQ1** partially. Our model learning based reduction approach does not compromise coverage and mutation score as severely as a random reduction. For examples where coverage and mutation score of the initial test suite are at maximum or close to maximum, coverage and mutation score for the test suites reduced by our approach, are close to coverage and mutation score of the initial test suite. Since the results are consistently better for our model learning based approach than a random reduction, our approach might be used to reduce a test suite.

Table 8.7.: Results of random reduced using same reduction size as obtained by model learning based reduction vs model learning based (ml based) reduction.

	statement - RTS_r (%)			statement - RTS_{ml} (%)			decision - RTS_r (%)			decision - RTS_{ml} (%)		
Name	*min.*	*max.*	*avg.*	*min.*	*max.*	*avg.*	*min.*	*max.*	*avg.*	*min.*	*max.*	*avg.*
TCAS	97.22	97.22	97.22	97.22	97.22	97.22	91.67	91.67	91.67	91.67	91.67	91.67
BMI	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
Triangle	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
POP_3	84.78	100.00	93.13	97.80	100.00	99.99	90.38	100.00	94.77	98.08	100.00	99.99
CAS	62.30	95.08	77.77	93.44	96.72	96.69	42.86	83.33	64.19	80.95	85.71	85.67
UTF8	92.85	100.00	99.14	85.7	100.00	96.71	90.00	100.00	96.20	90.00	100.00	97.70
CC	70.00	82.00	78.08	62.00	86.00	79.53	59.38	78.13	72.00	50.00	84.38	74.22

	MC/DC - RTS_r (%)			MC/DC - RTS_{ml} (%)			μ - RTS_r ([0..1])			μ - RTS_{ml} ([0..1])		
Name	*min.*	*max.*	*avg.*	*min.*	*max.*	*avg.*	*min.*	*max.*	*avg.*	*min.*	*max.*	*avg.*
TCAS	85	85	85	85	85	85	0.85	1.00	0.96	0.93	1.00	0.97
BMI	100.00	100.00	100.00	100.00	100.00	100.00	1.00	1.00	1.00	1.00	1.00	1.00
Triangle	100.00	100.00	100.00	100.00	100.00	100.00	1.00	1.00	1.00	1.00	1.00	1.00
POP_3	94.05	100.00	96.76	98.80	100.00	99.99	1.00	1.00	1.00	1.00	1.00	1.00
CAS	39.77	62.50	51.05	65.91	68.18	68.16	0.13	0.43	0.27	0.41	0.46	0.44
UTF8	72.50	97.50	86.10	85.00	100.00	93.64	0.52	0.90	0.75	0.80	0.99	0.90
CC	62.50	81.25	75.13	50.00	87.50	77.33	0.18	0.35	0.30	0.25	0.38	0.33

8.3.6. Threats To Validity

The selected examples in this chapter are used to investigate whether model learning based test suite reduction is possible in principle. Hence additional examples with more lines of code will be investigated in future work to underpin the approach. Since the structure (control flow, data flow, lines of code, etc.) of the program under test affects the approach, with more examples a possible classification can be created, where we can derive from the structure of the program under test whether the reduction approach is applicable.
In this chapter only two of the numerous existing methods to check whether two decision trees are equivalent were applied. Because the results are promising using these equivalence methods, additional methods are shown in Chapter 9. The test suites were generated randomly for various examples. Therefore of course, as for all test suite reduction approaches, the question, whether a test suite which is already of minimal size for a certain test purpose, is recognized as such from the reduction approach or not, arises.

8.4. Related Work

Harrold et al. [26] developed a test suite reduction technique based on a heuristic that selects a representative test suite from the original test suite by approximating the optimal reduced set. A representative test suite is a potential subset of test cases from the original test suite, which provides the same coverage. Their approach requires an association between a testing requirement and test cases, which satisfy the requirement. Since selecting a subset of test cases from the original test suite that satisfies all testing requirements with minimal cardinality is known as the *NP-complete problem of finding the minimum cardinality hitting set*, they introduce a heuristic to approximate the minimum cardinality. They also detect redundant test cases with this approach as we do in our work, but in addition they remove obsolete test cases from a test suite if a test requirement does not exist anymore.

Lin and Huang [110] introduce a test suite reduction technique called reduction with tie-breaking (RTB). In their paper they show the integration of their technique into two existing test suite reduction techniques, which are the technique introduced in [26] and the technique introduced in [111]. As an extension they use, additionally to a primary coverage criterion for the test requirements, a second coverage criterion to avoid elimination of a test case, which is more likely to detect faults when more than one test case has the same importance with respect to the primary coverage criterion in both cases. In their work they provide very interesting results, which can be directly compared to the results in our work as explained in detail in Chapter 8.

Taylor et al. [112] present a multi-objective search-based technique using behavior inference as fitness metric, to reduce a test suite. Behavior inference is a test adequacy metric first suggested in 1983 [68]. Taylor et al. infer finite state machines (FSM)s as models from execution traces of a test suite. They compare the FSM inferred from execution traces of a reduced test suite to the FSM inferred from execution traces of the original test suite, and thus compare the behavior coverage of test sets that produce them. Their results show that as long as the FSMs are equivalent, assessed by a Balanced Classification Rate, the reduced test suite retains the fault finding capability of the original test suite. The authors demonstrate that the reduced test suite retains all of the fault finding capability of the original test suite by using mutation testing, which also holds for our approach.

Briand et al. [109] describe a test suite refinement approach that relies on the black box testing technique Category Partition and machine learning. They use categories and choices to define the functional properties of a program under test where categories are associated with choices. E.g. a category representing an inequality relation has two choices of an inequality relation which are either greater than or less than. Based on these categories they transform test cases into abstract test cases. These abstractions are tuples of choices and an expected output value or an equivalence class of expected output values. Like in our work, they use the C4.5 algorithm to learn a decision tree in [109]. But in contrast to our work where we learn a decision tree from the raw values in a test suite, they learn decision trees from the abstractions obtained by category partitioning.

The decision trees learned by the abstractions are used to learn rules that relate categories or choices to equivalence classes of output values. They

analyze these rules to determine if some test cases are redundant or additional test cases are required. Potential problems which indicate redundancy or required improvements are misclassifications, too many test cases for a rule, unused categories, missing combinations of choices. These problems origin in different potential causes which are e.g. redundant test cases for the problem of too many test cases for a rule. They also provide case studies where students created the category partitioning manually. These categories and choices are used with a test suite to iteratively learn a decision tree, analyze the results, and improve the test suite. The studies showed that the resulting test suites were significantly more effective in terms of fault detection while only requiring a modest size increase in comparison to the originally provided test suite.
Because scientific work in test suite reduction has been done since decades, a tremendous amount of related publications exist. However, detailed overviews of test suite reduction literature can be found in [113] and [30].

8.5. Summary

In this chapter we introduced an algorithm for test suite reduction, which is based on model learning. The underlying idea is, any test case that is not relevant for learning a model from a test suite can be removed. In our implementation we utilized a decision tree inference algorithm for model learning and introduced two different notions of model equivalence. Furthermore, we compared the outcome of the approach with a traditional reduction approach based on coverage and mutation score of a test suite. For this purpose, we did an empirical evaluation that is based on seven example programs and their test suites to answer the two research questions posed in Chapter 8. The answer for RQ1 is that coverage and mutation score for the test suites reduced by our approach are close to coverage and mutation score of the original test suite. Coverage and mutation score for the randomly reduced test suites show much higher differences. For RQ2 the answer is that our model learning based reduction approach is multiple times faster, but not as efficient in size as a traditional approach depending on the content of a test case and the types of the inputs. On average the

obtained reduction was greater than 60% without substantial reduction of coverage and mutation score.

In future work, we will extend our empirical evaluation considering more examples from the application domain, i.e., automotive control software. Here the open research question is whether the program's structure, e.g., the existence of loops or recursive functions, or the use of certain expressions in the code, impacts the reduction results when using a specific machine learning algorithm.

9. A "Strength of Decision Tree Equivalence"-Taxonomy and Its Impact on Test Suite Reduction

This chapter is based on the work "A "strength of decision tree equivalence"-taxonomy and its impact on test suite reduction" [10].

An attractive feature of the approach introduced in Chapter 8 is that we do not need to execute the program under test for assessing the fault detection capabilities when removing a test case. The underlying idea was that every test suite T should at least partially capture the behavior of the program under test in a sufficient way. The strategy then is to use machine learning for model extraction, in order to derive representative characterizations from T and a reduced test suite T'. We introduced in Chapter 8 the following reduction process: Initially, we learn a characterizing decision tree from T, and when successively trying to remove test cases $t \in T$, we infer for each potential removal another decision tree from the updated test suite T'. If the decision tree for T' is *equivalent* to the initial one, we assume that the fault detection capabilities were not affected, and proceed with trying to remove further test cases. Otherwise, we go back one step and re-add t. The reduction terminates after a preconfigured number of unsuccessful, random tries to remove a further test case. With avoiding to execute T, we still could achieve reductions from 60 to 99% in our evaluation. Our approach for decision tree learning is limited to test cases t represented as some vector $t = \langle x_1, .., x_k, out \rangle$ of k input values and an expected output value out. The inputs are either numeric of an infinite domain, numeric of a finite domain,

or discrete strings or numbers. The output type has to be of a finite domain, whose values then build the labels of the decision trees' leaf nodes.

Since such a test suite reduction depends on an equivalence relation for decision trees, the following questions arise immediately: Which methods are there for determining equivalence? Are there more than structural and misclassification equivalence as discussed and used in Chapter 8 (coined syntactic and semantic equivalence there), and is there a relation between them? What is their impact on the efficiency and effectiveness of the reduction process?

Imagining variants, one has to take the characteristics of the derived trees into account. According to [114], optimizing a decision tree to a minimal number of nodes which would allow us to compare minimal or canonical ones, is in NP. Thus, the algorithm used to infer the decision trees in Chapter 8 is based on a statistical measure (the information gain of variables) and does not stringently build optimal decision trees. Consequently, trees inferred from different test suites might appear different in respect of their strict structure. Exploring flexibility in this respect, we consider five variants for checking some trees' equivalence. In particular, we consider in Section 9.2 structural ($\equiv$), spine ($=^s$), decision ($=^d$), table ($=^t$), and misclassification ($=^m$) equivalence aiming to cover and explore various decision tree aspects. We show and prove that these variants build a taxonomy as shown in Figure 9.1 in respect of their strength. We report in Section 9.3 on our corresponding experiments, considering computation time and the achieved reductions as well as the impact on fault detection capabilities. In Section 9.5 we conclude on our findings and line out future work.

9.1. Preliminaries

In this chapter, we infer a decision tree D from a test suite T via the well-known algorithm C4.5 [51]. Such a decision tree is a directed tree $D = (V, E)$ having nodes V and directed edges E connecting nodes. V can be split into decision nodes and leaf nodes, where a decision node has outgoing edges and represents a decision (i.e., a relational equation) like $x > 0$ (see Fig. 9.2) for some numeric input x, or x *equals* $\langle$*discrete value*$\rangle$ for discrete inputs.

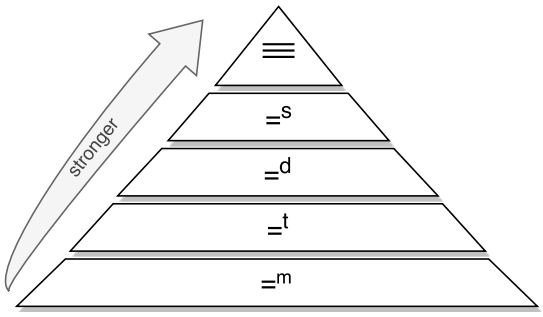

Figure 9.1.: Taxonomy of equivalence relation in respect of their strength.

A leaf node is a terminal one and offers a discrete classification. An edge (v, v') is a pair of nodes $(v, v' \in V)$, where v is parent of v'. For simplicity, we assume a function ρ: $DT \rightarrow V$ that returns the root node of a decision tree, with the universe of decision trees DT under consideration as input domain. Further we assume a function λ: $V \rightarrow J \cup C$ that returns the content of a node, with the union of the set of decisions J and the set of classifications C as range. The decision trees in this chapter are binary such that each decision node has exactly two outgoing edges. The answer of a decision, e.g., whether we have $x > 0$, is represented by an edge label that can be accessed via a function γ: $E \rightarrow \{T, F\}$. In our decision trees, paths are sequences containing nodes and connecting edges, starting from the root node, following down the tree, and ending at a leaf node. We define a path Π in a decision tree as follows:

Definition 18 (Path) *A path Π of length $|\Pi| = l$ in a decision tree D is a sequence of nodes $v_0 ... v_{l-1}$ such that there is an edge from v_i to v_{i+1} for $0 \leq i < l - 1$, starting with $v_0 = \rho(D)$ and ending at a leaf node v_{l-1}.*

With C4.5, decision trees are constructed top down, where decision nodes get selected using a statistical property called *information gain* that measures how well a decision separates the $t \in T$ according to their expected outcome[61]. A test case t is classified in a decision tree by following the decision nodes from the root node, down the tree to some leaf node, according to the values in t. Not necessarily all input variables appear in a decision tree, but numeric variables can occur also multiple times in different decision nodes, even in the same path. We define equivalence for decision trees as follows:

Definition 19 (Equivalence Relation) *Decision Tree Equivalence is a reflexive, symmetrical, and transitive binary relation R between two decision trees D_1 and D_2 from the universe of decision trees DT, such that:*
reflexivity: $\forall D \in DT : D\ R\ D$
symmetry: $\forall D_1, D_2 \in DT : D_1\ R\ D_2 \rightarrow D_2\ R\ D_1$
trans.: $\forall D_1, D_2, D_3 \in DT : D_1\ R\ D_2 \wedge D_2\ R\ D_3 \rightarrow D_1\ R\ D_3$

In our work, we consider the equivalence of decision trees when reducing test suites. When trying to remove test cases from a test suite T without effecting changes in the decision tree, the achieved reduction is an indicator of the reduction process' effectivity as shown in Definition 14.

9.1.1. Decision Tree Learning

When we infer a decision tree, we derive a hypothesis h regarding an approximation of a function f that we can use to predict f's outcome for future input values. Strategies for estimating the accuracy of such a hypothesis include k-folds cross validation [115], or assessment with additional input and output values [61]. In principle, for evaluating a hypothesis h, we can use the function $error(h, S)$ as given in Equation 9.1 in order to obtain a result in the range 0..1:

$$error(h, S) = \frac{1}{|S|} \sum_{t \in S} \delta(f(t), h(t)) \tag{9.1}$$

Equation 9.1 requires three parts: First, some set S that should be different to T (from which the hypothesis was learned) containing vectors t of input values and an expected output. Second, the target function $f : I^k \rightarrow O$, where I is the type of the k inputs and O represents the set of all possible outputs. Third, a function δ that detects deviating outcomes of f and h– returning 1 if $f(t) \neq h(t)$ for some $t \in S$ and 0 otherwise.

9.2. Equivalence Taxa

For our investigation, we considered five decision tree equivalence relations, ranging from structural equivalence to misclassification equivalence. Before showing at the end of this section that they form a taxonomy in respect of their strength, let us formally introduce them for the decision trees $D_1 = (V_1, E_1)$ and $D_2 = (V_2, E_2)$ first.

The first method determines structural equivalence of two decision trees, the second method determines spine equivalence, the third method determines decision equivalence, the fourth method is based on distinct classifications to determine table equivalence, and the fifth method is based on a misclassification rate and determines misclassification equivalence. The definitions of structural and misclassification equivalence were partially adopted from Chapter 8 where structural equivalence is named syntactic equivalence and misclassification equivalence is named semantic equivalence.

Structural Equivalence ($\equiv$): Two decision trees D_1 and D_2 are structurally equivalent, if and only if each node $v_1 \in V_1$ has a corresponding node $v_2 \in V_2$ and each edge $e_1 \in E_1$ has a corresponding edge $e_2 \in E_2$ connecting an equivalent pair of nodes. Structural equivalence can be represented using a function EQUAL: $V \times V \rightarrow \{True, False\}$, which we define recursively as follows: For two decision trees D_1, D_2, and nodes $v_1 \in V_1$, $v_2 \in V_2$, EQUAL returns *True*, if and only if:

1. $\lambda(v_1) = \lambda(v_2)$
2. $\forall(v_1, v_i) \in E_1,\ \exists(v_2, v_j) \in E_2,\ 0 \leq i, j < 2|$
   $\gamma(v_1, v_i) = \gamma(v_2, v_j) \wedge \text{EQUAL}(v_i, v_j)$ *(and vice versa)*

Using this function, we define structural equivalence of two decision trees as follows:

Definition 20 (Structural equivalence) *Two given decision trees D_1, D_2 are structurally equivalent if and only if the function* EQUAL$(\rho(D_1), \rho(D_2))$ *returns True.*

EQUAL terminates if it detects different node contents or different edge labels, or if all nodes have been visited.

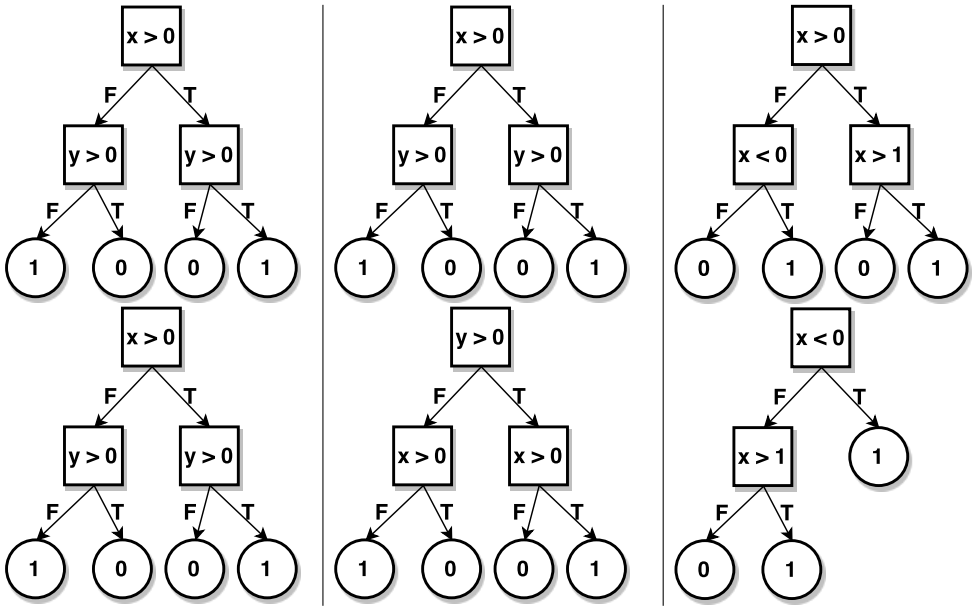

Figure 9.2.: Structurally (left), spine- (middle), and decision-equivalent (right) trees.

Example 2 (Structural equivalence) *Figure 9.2 shows two structurally equivalent decision trees where decision nodes, leaf nodes, and edges are equivalent and on the same position in both decision trees.*

Spine Equivalence ($=^s$): A decision tree consists of a set of spines SP. A spine $(\Pi, c) \in SP$ is described by a path Π to a leaf node v, such that $c = \lambda(v)$. Spine equivalence requires bag equivalence to hold, which is defined as:

Definition 21 (Bag equivalence) *Two paths Π_1 and Π_2 are equivalent as bags, if except for the ordering they contain nodes with precisely the same content and with equivalently labelled outgoing edges, such that for all $v_1 \in \Pi_1$ there exists an equivalent node $v_2 \in \Pi_2$ and vice versa, where $\lambda(v_1) = \lambda(v_2)$ and $\gamma(v_1, v_i) = \gamma(v_2, v_j)$.*

From the definitions of a path and bag equivalence, we define spine equivalence as:

Definition 22 (Spine equivalence) *Two decision trees D_1 and D_2 are spine equivalent if for the respective sets of spines SP_1 and SP_2, for every spine $(\Pi_1, c_1) \in SP_1$ there exists a spine $(\Pi_2, c_2) \in SP_2$ and vice versa, such that Π_1 and Π_2 are bag equivalent and $c_1 = c_2$.*

Example 3 (Spine equivalence) *Figure 9.2 shows two spine equivalent decision trees where the order of decision nodes in the paths differ, but the spines are equivalent. However, these decision trees are not structurally equivalent.*

Decision Equivalence ($=^d$): A constraint built from a spine's path is a conjunction of equivalence relations that contain a decision node's content and its outgoing edge's label for all decision nodes in the path. Satisfying a constraint classifies the inputs to that spine's c. In a decision tree, there may be multiple spines for some c. For decision equivalence, we thus build a summarizing constraint for each c as a disjunction of the corresponding conjunctions of the individual spines for c. E.g., from the top right decision tree in Figure 9.2, a constraint ψ of paths from spines with $c = 1$ is $(x > 0 = F \wedge x < 0 = T) \vee (x > 0 = T \wedge x > 1 = T)$. More formally, we define decision equivalence as:

Definition 23 (Decision equivalence) *Two decision trees D_1 and D_2 are decision equivalent, if for all leaf nodes $v_1 \in V_1$ an equivalent leaf node $v_2 \in V_2$ exists, and for each constraint ψ_1 of D_1 there exists a constraint ψ_2 in D_2 where the following equation holds:*

$$\psi_1 \textit{ equals } \psi_2 \tag{9.2}$$

Equation 9.2 is true, if no valuation exists for which ψ_1 is satisfiable and ψ_2 is unsatisfiable, and vice versa.

Example 4 (Decision equivalence) *Figure 9.2 shows two decision equivalent decision trees that do not contain the same decision nodes and are therefore not spine equivalent.*

Table Equivalence ($=^t$): A decision tree D classifies all test cases $t \in T$ according to the input values in t to a leaf node, as introduced in Section 9.1. A test case t is misclassified, if $\lambda(v)$ for the leaf node v to which t was classified and the value out of t differ. Otherwise t is classified correctly. This principle is also used in hypothesis evaluation as introduced in Section 9.1, where the function $h(t)$ returns the content of the leaf node to which t was classified, but unlike for hypothesis evaluation, here the outcome of the target function $f(t)$ is the value of out that is already included in t. We create a set M of pairs $(h(t), out)$ that contains for each $t \in T$ the content of the leaf node to which t was classified and the value out of t. Note that we have $|M| = |T|$. Two sets M_1 and M_2 are equivalent, if for each pair $(h(t)_1, out_1) \in M_1$ there is a pair $(h(t)_2, out_2) \in M_2$ such that $h(t)_1 = h(t)_2$ and $out_1 = out_2$, and vice versa. Consequently, we define table equivalence as:

Definition 24 (Table Equivalence) *Two decision trees D_1 and D_2 are table equivalent, when classifying a test suite T yields two equivalent sets M_1 for D_1 and M_2 for D_2.*

Example 5 (Table equivalence) *Figure 9.3 shows two decision trees that are table equivalent if T does not contain a test case $t = \langle 1, 1, 1 \rangle$, because then $h(t) \neq out$ only for the lower decision tree. These decision trees are not decision equivalent.*

Misclassification Equivalence ($=^m$): Two decision trees D_1 and D_2 are equivalent regarding their misclassification rate $error(D, T)$, if the following two conditions hold:

1. $error(D_2, T) = error(D_1, T)$.
2. For all distinct contents in the leaf nodes $C_1 \subset V_1$ an equivalent classification exists in the leaf nodes $C_2 \subset V_2$ and vice versa.

Definition 25 (Misclassification equivalence) *A decision tree D_2 is misclassification equivalent to a reference decision tree D_1, when classifying T, if the following equation holds:*

$$\begin{gathered} error(D_2, T) = error(D_1, T) \wedge \\ \forall v_1 \in C_1, \exists v_2 \in C_2 | v_1 = v_2 \textit{ (and vice versa)} \end{gathered} \tag{9.3}$$

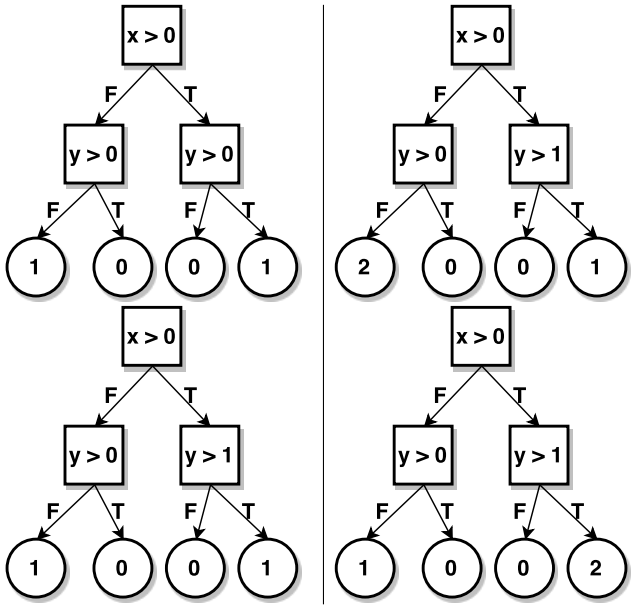

Figure 9.3.: Table (left) and misclassification-equivalent (right) trees.

Example 6 (Misclassification equivalence) *Figure 9.3 shows two decision trees where the same classifications exist in both decision trees as visualized by the leaf nodes. If T contains two test cases* $t_1 = \langle 0, 0, 2 \rangle$ *and* $t_2 = \langle 1, 2, 2 \rangle$*, the decision trees are misclassification equivalent, but not table equivalent.*

Theorem 4 *The five defined methods to determine equivalence of decision trees can be presented in a subset order, where for a decision tree D inferred from a test suite T, subsets* $DT_{\equiv} \subset DT$, $DT_{=^s} \subset DT$, $DT_{=^d} \subset DT$, $DT_{=^t} \subset DT$, *and* $DT_{=^m} \subset DT$ *from the universe of decision trees DT exist, which contain decision trees that were inferred from a test suite* $T' \subseteq T$*, and are equivalent to D. These subsets are ordered as*
$DT_{\equiv} \subseteq DT_{=^s} \subseteq DT_{=^d} \subseteq DT_{=^t} \subseteq DT_{=^m}$*, for subsets*
$DT_{\equiv} \subset DT$ *representing structural equivalent decision trees,*
$DT_{=^s} \subset DT$ *representing spine equivalent decision trees,*
$DT_{=^d} \subset DT$ *representing decision equivalent decision trees,*
$DT_{=^t} \subset DT$ *representing table equivalent decision trees, and*
$DT_{=^m} \subset DT$ *representing misclassification equivalent decision trees.*

Proof 4 *(sketch) Structural equivalence implies that all paths are equivalent. If all paths are equivalent, spine equivalence is ensured. If paths in two decision trees only have different orders of nodes, the decision trees are spine equivalent, but not structurally equivalent. Spine equivalence implies that all nodes contain the same content. Building constraints from the paths in spines ensures that the constraints are equivalent, because they contain the same contents of nodes and the same outgoing edges of the decision nodes. If a node is missing or redundant in a path of two decision equivalent decision trees, this contradicts spine equivalence. Decision equivalence implies that each possible input valuation leads to an equivalent classification or misclassification. Equivalent classifications for all possible input values ensure table equivalence, because table equivalence depends only on the classification of a test suite T, which contains only a subset of all possible input values. If a pair of decision trees is table equivalent, but test cases are missing for boundary values of the decisions, different decision nodes in a path lead to equivalent classifications for a test suite T, but not for each possible input valuation. This fact contradicts decision equivalence. Table equivalence implies that two decision trees provide equivalent classifications for a test suite T, independently of whether a test case was correctly classified or misclassified. If all test cases in T are equally classified or misclassified, misclassification equivalence is given. A misclassification equivalent pair of decision trees where classifying a test suite T yields the same misclassification rate, but different test cases from T are misclassified, violates table equivalence.* ■

As stated in Theorem 4, structural equivalence is the strongest method to determine equivalence of two decision trees, meaning that even if the four other equivalence check methods evaluate to true, structural equivalence can be false. Decision equivalence is the costliest method due to the NP-completeness of determining inequality of two constraints. Misclassification equivalence is the weakest method to determine equivalence of two decision trees, because it neither considers the structure of the decision tree nor the relation of inputs to outputs.

9.3. Experimental Evaluation

We used three different Java programs for our proof-of-concept experiments, generated combinatorial test suites using the tool ACTS, and evaluated the reduced test suites' fault detection effectiveness via their mutation score. For generating mutants, we used the Major mutation framework [38].

9.3.1. Results

The three examples are Triangle, TCAS, and UTF8, as introduced in Section 5.2. For test suite reduction, we used again a Java implementation of the *REDUCE* Algorithm 7 and instantiated the *equals* method in *REDUCE* at line 12 by all 5 equivalence methods introduced in Section 9.2. The input values for *iterations* and *retries* of *REDUCE* were set to 2 and $\frac{|T|}{10}$ respectively, since the results in Chatper 8 show that these values allow high reductions. To infer decision trees from a test suite, we used the Java library Weka [93] and its implementation J48 of the algorithm C4.5. In the configuration options of Weka, we disabled pruning and set the minimum number of leaf nodes to 1. The expected outcome for a test case was derived with the original program. We obtained test suites of size 343 (Triangle), 1840 (UTF8), and 11021 (TCAS), and generated 35 (Triangle) and 147 (UTF8) mutants. For the TCAS example, we used the 41 existing mutants[1]. In order to determine decision equivalence, we applied the SMT-solver Z3 [116] that provides a Java-API. For calculating the misclassification rate, we used the decision tree evaluation method integrated in the Weka library. Since the *REDUCE* algorithm selects potentially redundant test cases randomly, we executed the algorithm for each example 10 times per equivalence method and plot the execution time and the resulting reduction for each execution. All experiments ran on a MacBook Pro with an Intel Core i5 2.7GHz CPU, 16GB RAM, an SSD, and OS X 10.11.6. The resulting reductions and the runtime to obtain these reductions for the Triangle example are shown in Figure 9.4. The results in Figure 9.4 show that decision equivalence is multiple times slower than other equivalence methods. Structural equivalence is

[1] http://sir.unl.edu/portal/bios/tcas.php

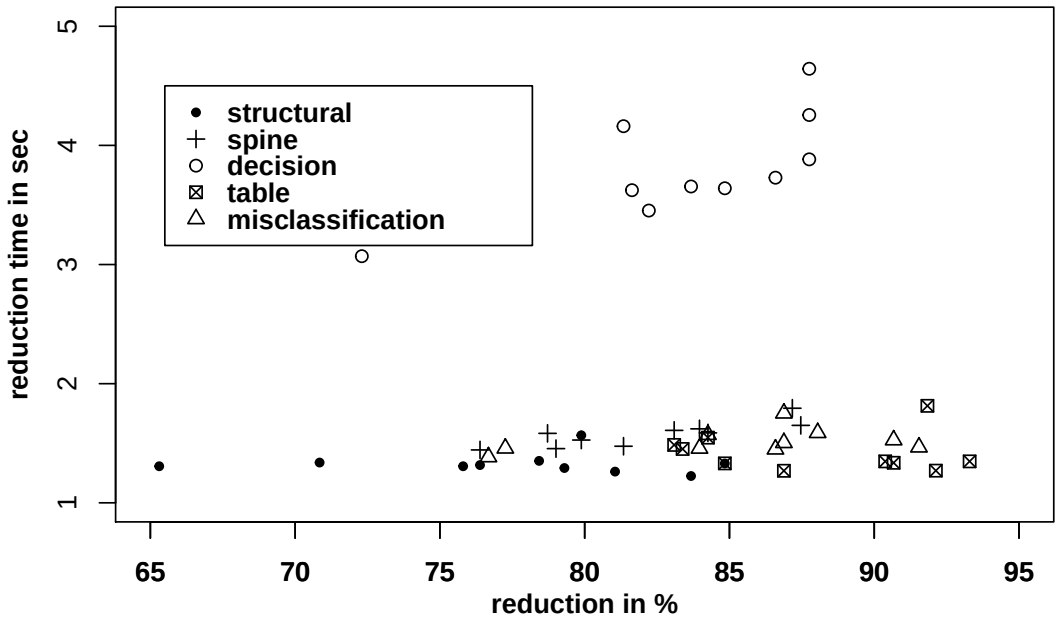

Figure 9.4.: Triangle results.

fastest, misclassification and table equivalence allow the highest reductions. Reductions of structural equivalence are lowest. The results in Figure 9.5 for the UTF8 example show that structural and spine equivalence are fastest, but the reductions are around 30% lower than for the other equivalence methods. Also for the UTF8 example decision equivalence was slowest. For the TCAS example the results in Figure 9.6 show that all reductions only vary in a range of around 10%. Also for TCAS, structural and spine equivalence are fastest and decision equivalence is slowest on average. The highest reductions were obtained by table and misclassification equivalence.

9.3.2. Discussion

Our results suggest that structural equivalence, whose complexity is linear in the number of nodes in a decision tree, is the fastest and decision equivalence is the slowest equivalence method. Deciding decision equivalence is an NP-complete problem and each pair of equivalent constraints in two decision

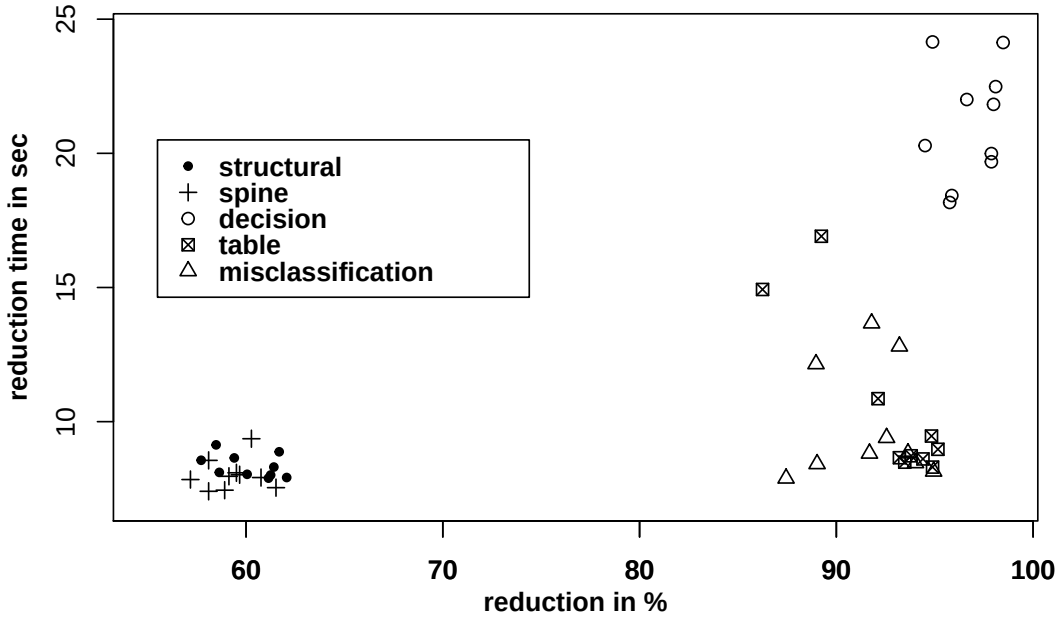

Figure 9.5.: UTF8 results.

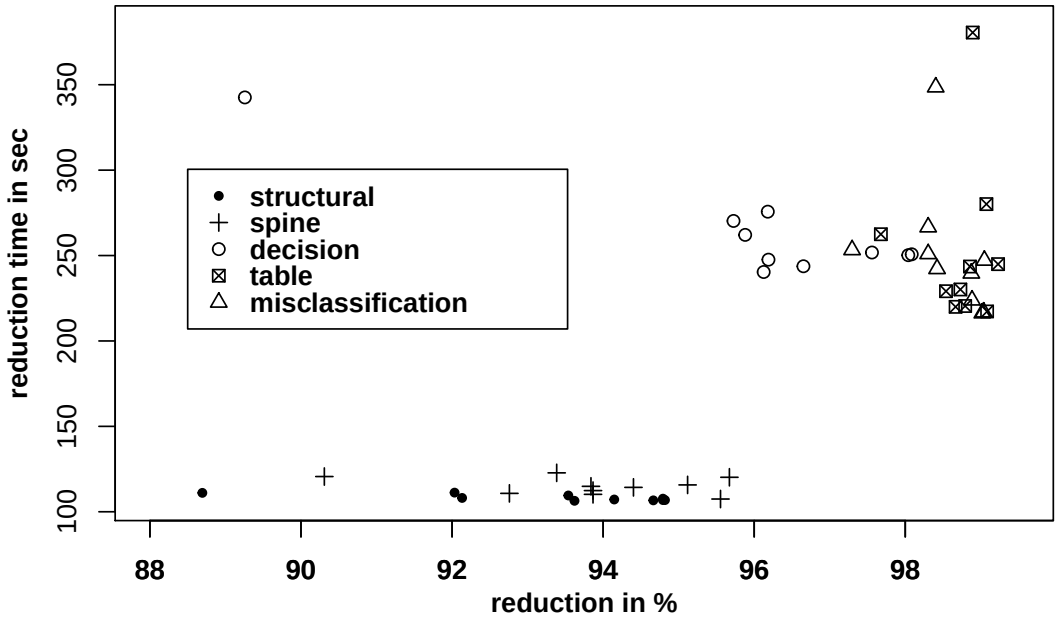

Figure 9.6.: TCAS results.

trees gives the worst case. When using misclassification equivalence, which allows the highest reductions, the time to reduce T was slightly higher than for structural equivalence. For evaluating a potential loss of the test suite's fault detection effectiveness, we derived the mutation score for all reduced test suites as reported in Figure 9.7. The mutation score of the initial test suites was 1 for each example. For each example in Figure 9.7, the equivalence methods are ordered according to their strength from left to right, starting on the left with the strongest one. The results show that for the strongest equivalence method there was almost no decline of mutation score, but for weaker methods the mutation score decreased. In particular for the UTF8 example, the median mutation score dropped to values in the range 0.6 to 0.7 for decision, table, and misclassification equivalence. These weak mutation scores origin in the fact that the initial test suite contained test cases with unknown values, which were approximated automatically while inferring a tree by the C4.5 algorithm. These approximations increased potential uncertainties of the tree to predict future outputs for additional input values. The dots in the plots for Triangle, UTF8, and TCAS represent outliers from the obtained results.

Using structural or spine equivalence provided similar reductions at similar costs. Although decision equivalence allowed high reductions, the computation time was highest from all equivalence methods. Table and misclassification equivalence provided the highest reduction results for our examples, consuming more time than structural and spine equivalence (but in most cases less time than decision equivalence). The mutation score results suggest the highest loss of fault detection effectiveness to occur when using table or misclassification equivalence. Therefore, if the execution time of the tests in the finally reduced test suite is low, structural equivalence should be chosen. If keeping the fault detection capabilities as high as possible for a reduced test suite, also structural equivalence should be chosen. In all other cases the results suggest that misclassification equivalence is an educated choice. Promising results of an empirical evaluation of structural and misclassification equivalence were shown in Chapter 8. With our results, we clarify that the runtime of the reduction approach depends on three parts. First, the runtime depends on the size of the test suite and the domain sizes of the inputs. The latter affects the run-time spent for the algorithm C4.5, since we have to learn a decision tree for each potentially removable

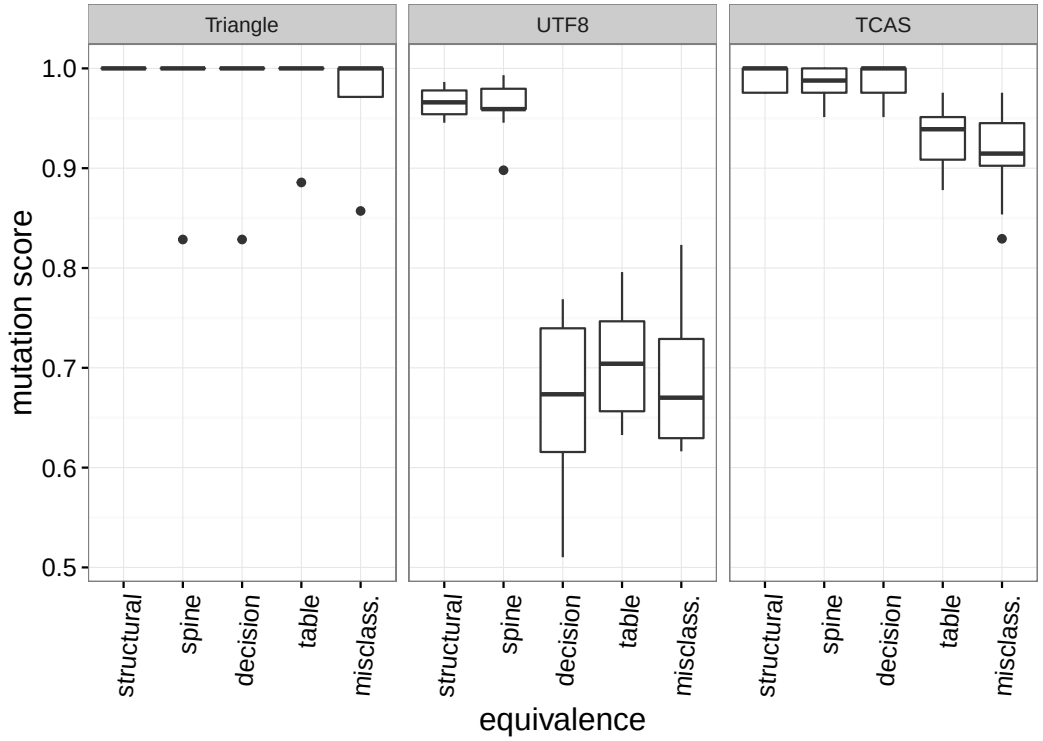

Figure 9.7.: Mutation score of reduced test suites.

test case. Second, as we surmised, the runtime depends on the complexity of the equivalence relation used. Last, but not least, we saw that the runtime increases also with the achieved reduction.

9.4. Related Work

Safavian and Landgrebe provide a survey of decision tree classifiers in [117]. They address the design, search strategies, issues like missing values and robustness, and potential problems of decision trees in their survey. In [118] Moret provides a common framework of definitions and notations for decision trees. In [119] Dattatreya and Kanal introduce the usage of decision trees in pattern recognition. In this context they define pattern recognition

as "the assignment of a physical object or an event to one of the prespecified categories". They consolidate the major methodologies for decision tree design, bring out those methodologies' commonalities, provide insight into multistage classification, explode the myth that decision trees are always simple to design and use, mention areas of applications of decision trees, and aid a decision tree designer to select an appropriate technique for the particular problem of interest.
Cockett introduces in [120] different notions of decision tree equivalence. These notions are structural, decision, and transposition equivalence that are similar to some of the notions we use in this work, which are structural, spine, and decision equivalence, but Cockett uses the notion of coalgebras to describe decision trees and the equivalence relations. In [121] Zantema presents a simple efficient algorithm to establish whether two decision trees are equivalent or not. This algorithm is an axiomatization for decision equivalence as we use it in this work. The complexity of this algorithm is bounded by the product of the number of nodes n and m of both decision trees ($O(n * m)$). The algorithm only processes decision trees representing discrete valued variables as decision nodes. In our work we also cover numeric inputs, which are handled by binary splits. The authors in [122] present an algorithm that reduces a decision tree by replacing the decision tree with a smaller equivalent decision tree. To find an irreducible tree using the reduction algorithm they also use decision and transposition equivalence. In [123] the authors address the question, whether for a given decision tree, a decision tree equivalent to the given one can be found, for which no decision equivalent decision tree of smaller size exists. Breslow and Aha provide an overview over methods how to simplify a decision tree in [124].

The underlying idea that a model inferred from a test suite can be used to indicate the fault detection effectiveness of the test suite was already introduced in Chapter 5. The promising results in Chapter 8, where reductions of 60-99% were possible, while still keeping coverage and mutation score almost the same, led to this work, where we used the same reduction algorithm. Briand et al. [109] describe a test suite refinement approach that relies on the black box testing technique Category-Partition [125] and machine learning. They use categories and choices to define the functional properties of a program under test, where categories are associated with choices. E.g. a

category representing an inequality relation has two choices of an inequality relation that are either greater than or less than. Based on these categories they transform test cases into abstract test cases. These abstractions are tuples of choices and an expected output value or an equivalence class of expected output values. Like in our work, they use the C4.5 algorithm [51] to learn a decision tree in [109]. But in contrast to our work, where we learn a decision tree from the raw values in a test suite, they learn decision trees from the abstractions obtained by category-partitioning.

Since test suite reduction has been of interest for decades, there is a tremendous amount of further related work. We refer the interested reader to [113] and [30] for detailed overviews.

9.5. Summary

In this chapter, we introduce a "strength of decision tree equivalence"-taxonomy of five different equivalence relations. Decision tree equivalence is a crucial part of a recently introduced test suite reduction approach that does not require to execute the program under test. We came up with five different methods to determine this equivalence and provide a theorem and a corresponding proof that these methods form a taxonomy in respect of their strength. As a proof of concept, our experiments show that the equivalence method indeed has a high impact on the effectiveness and efficiency when reducing a test suite. The results yield structural and spine equivalence as the methods with the lowest costs, but also with the smallest reduction. Decision equivalence is the costliest in respect of computation time, but achieves high reductions. When determining equivalence with table and misclassification equivalence, the reductions are very high, but suffer from the highest decrease in fault detection effectiveness.

Underpinning the reduction approach itself and the selection of the most appropriate equivalence relation will require an evaluation with additional, realistic scenarios. If some T does not contain redundancies, no reduction is possible. For detecting that T does not contain redundancies structural equivalence should be chosen, because it is the least time consuming relation to determine. Since the structure (control flow, data flow, lines of code,

etc.) of the program under test affects the reduction, with more examples possibly a classification can be created such that we could derive from the program structure in combination with background information on how T was generated which equivalence method would be best suited.

For our current experiments, we used first order mutants for evaluating the effectiveness in fault detection, but towards applicability of the reduction approach in practice, an examination with higher order mutants shall be part of future work. In future work we will extend also our empirical evaluation, considering more examples from application domains like automotive control software.

Part V.

Future Work

10. Directions for Quality Assessment of Test Suites Without Execution

Quality assessment of test suites has a long history. Various metrics were introduced and investigated empirically. Due to the increasing complexity of systems controlled by software also testing requires more and more effort. In this work we introduced a new method to assess the quality of a test suite, based on model inference, which is also applicable to eliminate redundancies in a test suite. This method does not necessarily require the execution of the test cases and does not require any instrumentation within the source or binary code. Therefore the effort to assess the quality of a test suite and eliminating redundancies can be reduced.
Beyond a general introduction of model inference based quality assessment we give an introduction of combinatorial, random, and property based test case generation, which were used to prepare the empirical results in this work. Also existing quality assessment methods are explained in detail, to support benchmarking of the newly introduced approach with existing approaches.
We started with investigating existing methods to automatically generate test cases and went into detail with combinatorial testing. Combinatorial testing allows to generate automatically a test suite that satisfies a quality metric called combinatorial coverage. Since combinatorial testing becomes more and more popular also in industrial applications, the results we found by generalizing manually written test cases and generating new combinatorial instantiations of these test cases are very promising. Additionally, property based and random test suite generation were applied the existing quality assessment methods, mutation score and code coverage, were im-

plemented. Based on these results we analyzed the correlation of the test suite quality assessment results with the results obtained from our newly introduced approach. For Boolean functions we introduced an effectiveness classification metric where an inferred model and the original function are transferred into the representation of reduced ordered binary decision diagrams, for which the equivalence problem is decidable.
Additionally we applied model inference based quality assessment to eliminate redundancies in test suites. For this approach we introduced equivalence relations, which can be represented in a taxonomy, that allow to assess the difference of the original test suite and the reduced test suite.

As the complexity of cyber physical systems continuously grows, this requires more and more the application of automatic test suite generation. Various methods and tools already exist for test suite generation, which enforce the necessity for an adaptable and feasible quality assessment method. The presented model inference based approaches give a basis for using machine learning or artificial intelligence in general, to be applied for quality assessment and redundancy elimination. Integration of other or combination of additional artificial intelligence approaches could build on the presented approaches.
As a consequence of the growing complexity also the respective test suites are massively growing. An at the moment ubiquitous example is the development of highly automated and autonomous driving systems. This area requires a huge amount of testing, due to the infinite number of events that possibly can occur on the road, and the consequences that can happen, due to erroneous software. To reduce the testing effort demands quality assessment and especially redundancy elimination. For this reason, a direction for future work of the presented approaches should be their application in an industrial application, e.g., highly automated and autonomous driving.

Bibliography

[1] Jim A McCall, Paul K Richards, and Gene F Walters. *Factors in software quality. volume i. concepts and definitions of software quality*. Tech. rep. GENERAL ELECTRIC CO SUNNYVALE CA, 1977 (cit. on pp. 3, 4).

[2] *Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Guide to SQuaRE*. Standard. Geneva, CH: International Organization for Standardization, Mar. 2014 (cit. on p. 4).

[3] Leon Osterweil. "Strategic Directions in Software Quality." In: *ACM Computing Surveys* 28.4 (Dec. 1996), pp. 738–750 (cit. on pp. 4, 5).

[4] H. Felbinger, F. Wotawa, and M. Nica. "Adapting Unit Tests by Generating Combinatorial Test Data." In: *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2018, pp. 352–355 (cit. on pp. 7, 15).

[5] Hermann Felbinger. "Test Suite Quality Assessment Using Model Inference Techniques." In: *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2015, pp. 1–2 (cit. on p. 7).

[6] Hermann Felbinger, Franz Wotawa, and Mihai Nica. "Empirical Study of Correlation Between Mutation Score and Model Inference Based Test Suite Adequacy Assessment." In: *11th International Workshop on Automation of Software Test*. ACM, 2016, pp. 43–49 (cit. on pp. 7, 43).

[7] Hermann Felbinger, Ingo Pill, and Franz Wotawa. "Classifying Test Suite Effectiveness via Model Inference and ROBBDs." In: *10th International Conference on Tests and Proofs*. 2016, pp. 76–93 (cit. on pp. 7, 61).

[8] H. Felbinger, F. Wotawa, and M. Nica. "Mutation Score, Coverage, Model Inference: Quality Assessment for T-Way Combinatorial Test-Suites." In: *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops*. IEEE. 2017, pp. 171–180 (cit. on pp. 7, 16, 85).

[9] H. Felbinger, F. Wotawa, and M. Nica. "Test-Suite Reduction Does Not Necessarily Require Executing the Program under Test." In: *2016 IEEE International Conference on Software Quality, Reliability and Security Companion*. IEEE, 2016, pp. 23–30 (cit. on pp. 8, 16, 111, 113).

[10] Hermann Felbinger, Ingo Pill, and Franz Wotawa. "A "strength of decision tree equivalence"-taxonomy and its impact on test suite reduction." In: *IFIP International Conference on Testing Software and Systems*. Springer. 2017, pp. 197–212 (cit. on pp. 8, 145).

[11] D. Richard Kuhn. "Fault Classes and Error Detection Capability of Specification-based Testing." In: *ACM Transactions on Software Engineering and Methodology* 8.4 (), pp. 411–424 (cit. on pp. 11, 16).

[12] Changhai Nie and Hareton Leung. "A Survey of Combinatorial Testing." In: *ACM Computing Surveys* 43.2 (), 11:1–11:29 (cit. on pp. 11, 106).

[13] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. "Interaction Testing of Highly-configurable Systems in the Presence of Constraints." In: *Proceedings of the 2007 International Symposium on Software Testing and Analysis*. London, United Kingdom: ACM, 2007, pp. 129–139 (cit. on pp. 11, 12).

[14] Yu Lei and Kuo-Chung Tai. "In-Parameter-Order: A Test Generation Strategy for Pairwise Testing." In: *The 3rd IEEE International Symposium on High-Assurance Systems Engineering*. IEEE, 1998, pp. 254–261 (cit. on p. 12).

[15] Yu Lei et al. "IPOG: A General Strategy for T-Way Software Testing." In: *Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*. IEEE, 2007, pp. 549–556 (cit. on p. 12).

[16] Yu Lei et al. “IPOG/IPOG-D: efficient test generation for multi-way combinatorial testing.” In: *Software Testing, Verification and Reliability* 18.3 (2008), pp. 125–148 (cit. on p. 12).

[17] Michael Forbes et al. “Refining the In-parameter-order strategy for constructing covering arrays.” In: *Journal of Research of the National Institute of Standards and Technology* 113.5 (2008), pp. 287–297 (cit. on pp. 12, 89).

[18] T. Xie et al. “Mutation Analysis of Parameterized Unit Tests.” In: *2009 IEEE International Conference on Software Testing, Verification, and Validation Workshops*. IEEE, 2009, pp. 177–181 (cit. on pp. 15, 33).

[19] Nikolai Tillmann and Wolfram Schulte. “Parameterized Unit Tests.” In: *Proceedings of the 10th European Software Engineering Conference*. ACM, 2005, pp. 253–262 (cit. on pp. 15, 16, 32).

[20] Justyna Petke et al. “Practical combinatorial interaction testing: Empirical findings on efficiency and early fault detection.” In: *IEEE Transactions on Software Engineering* 41.9 (2015), pp. 901–924 (cit. on p. 16).

[21] Dolores R Wallace and D Richard Kuhn. “Failure modes in medical device software: an analysis of 15 years of recall data.” In: *International Journal of Reliability, Quality and Safety Engineering* 8.04 (2001), pp. 351–371 (cit. on pp. 16, 106).

[22] Jacek Czerwonka. “Pairwise testing in the real world: Practical extensions to test-case scenarios.” In: *Proceedings of 24th Pacific Northwest Software Quality Conference*. 2006, pp. 419–430 (cit. on pp. 16, 105).

[23] Michael Ellims, Darrel Ince, and Marian Petre. “The Effectiveness of T-Way Test Data Generation.” In: *Proceedings of the 27th International Conference on Computer Safety, Reliability, and Security*. Springer-Verlag, 2008, pp. 16–29 (cit. on pp. 16, 106).

[24] P. J. Schroeder, P. Bolaki, and V. Gopu. “Comparing the fault detection effectiveness of n-way and random test suites.” In: *Proceedings of the International Symposium on Empirical Software Engineering*. IEEE, 2004, pp. 49–59 (cit. on pp. 16, 106).

[25] L. S. G. Ghandehari et al. "Applying Combinatorial Testing to the Siemens Suite." In: *2013 IEEE International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2013, pp. 362–371 (cit. on pp. 16, 102, 106).

[26] M. Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. "A Methodology for Controlling the Size of a Test Suite." In: *ACM Transactions on Software Engineering Methodology* 2.3 (1993), pp. 270–285 (cit. on pp. 16, 111, 141, 142).

[27] James A Jones and Mary Jean Harrold. "Test-suite reduction and prioritization for modified condition/decision coverage." In: *IEEE Transactions on Software Engineering* 29.3 (2003), pp. 195–209 (cit. on pp. 16, 111).

[28] W. E. Wong et al. "Test set size minimization and fault detection effectiveness: a case study in a space application." In: *Proceedings of the 21st International Conference on Computers, Software and Applications*. 1997, pp. 522–528 (cit. on pp. 16, 111).

[29] Gordon Fraser and Franz Wotawa. "Redundancy Based Test-suite Reduction." In: *Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering*. Springer-Verlag, 2007, pp. 291–305 (cit. on pp. 16, 111).

[30] S. Yoo and M. Harman. "Regression Testing Minimization, Selection and Prioritization: A Survey." In: *Software Testing, Verification And Reliability* 22.2 (2012), pp. 67–120 (cit. on pp. 16, 143, 161).

[31] Macario Polo Usaola, Pedro Reales Mateo, and Beatriz Pérez Lamancha. "Reduction of Test Suites Using Mutation." In: *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering*. Springer-Verlag, 2012, pp. 425–438 (cit. on p. 16).

[32] René Just, Darioush Jalali, and Michael D. Ernst. "Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs." In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ISSTA 2014. ACM, 2014, pp. 437–440 (cit. on pp. 16, 25).

[33] *JUnit*. `http://junit.org` (cit. on p. 17).

[34] Suresh Thummalapenta et al. “Retrofitting Unit Tests for Parameterized Unit Testing.” In: *Proceedings of the 14th International Conference on Fundamental Approaches to Software Engineering*. Springer-Verlag, 2011, pp. 294–309 (cit. on pp. 18, 19, 32).

[35] Peli de Halleux and Nikolai Tillmann. “Parameterized Test Patterns For Effective Testing with Pex.” In: (2008) (cit. on p. 18).

[36] *Defects4J*. https://github.com/rjust/defects4j. Version 1.1.0 (cit. on p. 25).

[37] *Cobertura - A code coverage utility for Java*. http://cobertura.github.io/cobertura/. Version 2.0.3 (cit. on p. 26).

[38] Rene Just, Franz Schweiggert, and Gregory M. Kapfhammer. “MAJOR: An Efficient and Extensible Tool for Mutation Analysis in a Java Compiler.” In: *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2011, pp. 612–615 (cit. on pp. 26, 89, 155).

[39] I. D. Mendoza et al. “CCM: A Tool for Measuring Combinatorial Coverage of System State Space.” In: *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2013, pp. 291–291 (cit. on p. 27).

[40] D Richard Kuhn et al. “Combinatorial coverage measurement concepts and applications.” In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*. IEEE. 2013, pp. 352–361 (cit. on pp. 27, 39).

[41] Gilles Bernot, Marie Claude Gaudel, and Bruno Marre. “Software Testing Based on Formal Specifications: A Theory and a Tool.” In: *Software Engineering Journal* 6.6 (1991), pp. 387–405 (cit. on p. 32).

[42] *Pex and Moles – Isolation and White Box Unit Testing for .NET*. https://www.microsoft.com/en-us/research/project/pex-and-moles-isolation-and-white-box-unit-testing-for-net/ (cit. on p. 32).

[43] Gordon Fraser and Andreas Zeller. “Generating Parameterized Unit Tests.” In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 2011, pp. 364–374 (cit. on p. 32).

[44] Gordon Fraser and Andreas Zeller. "Mutation-driven Generation of Unit Tests and Oracles." In: *Proceedings of the 19th International Symposium on Software Testing and Analysis*. ACM, 2010, pp. 147–158 (cit. on p. 32).

[45] David Saff, Marat Boshernitsan, and Michael D. Ernst. *Theories in Practice : Easy-to-Write Specifications that Catch Bugs*. Tech. rep. MIT Computer Science and Artificial Intelligence Laboratory, 2008 (cit. on p. 32).

[46] Hila Peleg, Dan Rasin, and Eran Yahav. "Generating Tests by Example." In: *Proceedings of the 19th International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer International Publishing, 2018, pp. 406–429 (cit. on p. 33).

[47] Yue Jia and Mark Harman. "An Analysis and Survey of the Development of Mutation Testing." In: *IEEE Transactions on Software Engineering* 37.5 (2011), pp. 649–678 (cit. on p. 38).

[48] René Just, Gregory M. Kapfhammer, and Franz Schweiggert. "Using Non-redundant Mutation Operators and Test Suite Prioritization to Achieve Efficient and Scalable Mutation Analysis." In: *Proceedings of the 2012 23rd IEEE International Symposium on Software Reliability Engineering*. IEEE Computer Society, 2012, pp. 11–20 (cit. on p. 38).

[49] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. 3rd. Wiley Publishing, 2011 (cit. on p. 39).

[50] Joseph Chilenski and Steven P Miller. "Applicability of modified condition/decision coverage to software testing." In: *Software Engineering Journal* 9.5 (1994), pp. 193–200 (cit. on pp. 39, 81).

[51] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., 1993 (cit. on pp. 41, 64, 85, 86, 112, 115, 146, 161).

[52] Karl Pearson. "Mathematical Contributions to the Theory of Evolution. III. Regression, Heredity, and Panmixia." In: *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 187 (1896), pp. 253–318 (cit. on p. 44).

[53] J. D. Evans. *Straightforward Statistics for the Behavioral Sciences*. Brooks/Cole Publishing, Pacific Grove, 1996 (cit. on p. 45).

[54] S. S. SHAPIRO and M. B. WILK. "An analysis of variance test for normality (complete samples)." In: *Biometrika* 52.3-4 (1965), pp. 591–611 (cit. on p. 45).

[55] *Software-artifact Infrastructure Repository*. http://sir.unl.edu/portal/bios/tcas.php. Accessed: 2016-01-13 (cit. on p. 46).

[56] Gordon Fraser and Neil Walkinshaw. "Assessing and generating test sets in terms of behavioural adequacy." In: *Software Testing, Verification and Reliability* 25.8 (2015), pp. 749–780 (cit. on pp. 46, 58, 90, 113).

[57] Daniel Hoffman et al. "Bad Pairs in Software Testing." In: *Testing – Practice and Research Techniques*. Springer Berlin Heidelberg, 2010, pp. 39–55 (cit. on pp. 46, 91).

[58] *Guava UTF8 source code*. https://github.com/google/guava/blob/master/guava/src/com/google/common/base/Utf8.java. Accessed: 2016-01-13 (cit. on p. 47).

[59] B.K. Aichernig, F. Lorber, and S. Tiran. "Integrating Model-Based Testing and Analysis Tools via Test Case Exchange." In: *Sixth International Symposium on Theoretical Aspects of Software Engineering (TASE)*. 2012, pp. 119–126 (cit. on pp. 49, 113).

[60] *Weka 3: Data Mining Software in Java*. http://www.cs.waikato.ac.nz/ml/weka/. Accessed: 2016-01-14 (cit. on pp. 50, 89).

[61] Tom M. Mitchell. *Machine learning*. Vol. 8. McGraw-Hill Boston, MA, 1997 (cit. on pp. 56, 147, 148).

[62] John B. Goodenough and Susan L. Gerhart. "Toward a theory of test data selection." In: *IEEE Transactions on Software Engineering* 1.2 (1975), pp. 156–173 (cit. on p. 58).

[63] L. G. Valiant. "A Theory of the Learnable." In: *Communications of the ACM* 27.11 (1984), pp. 1134–1142 (cit. on pp. 58, 80).

[64] Petros Papadopoulos and Neil Walkinshaw. "Black-box Test Generation from Inferred Models." In: *Proceedings of the Fourth International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*. IEEE, 2015, pp. 19–24 (cit. on pp. 58, 113).

[65] Laura Inozemtseva and Reid Holmes. "Coverage is Not Strongly Correlated with Test Suite Effectiveness." In: *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 435–445 (cit. on p. 58).

[66] M. G. Kendall. "A New Measure of Rank Correlation." In: *Biometrika* 30.1/2 (1938), pp. 81–93 (cit. on p. 58).

[67] René Just et al. "Are Mutants a Valid Substitute for Real Faults in Software Testing?" In: *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 654–665 (cit. on p. 58).

[68] Elaine J. Weyuker. "Assessing Test Data Adequacy Through Program Inference." In: *ACM Transactions on Programmming Languages and Systems* 5.4 (1983), pp. 641–655 (cit. on pp. 58, 79, 142).

[69] Hong Zhu, Patrick Hall, and John May. "Inductive inference and software testing." In: *Software Testing, Verification and Reliability* 2.2 (1992), pp. 69–81 (cit. on p. 58).

[70] M. M. Brandis and H. Mössenböck. "Single-pass generation of static assignment form for structured languages." In: *ACM TOPLAS* 16(6) (1994), pp. 1684–1698 (cit. on p. 63).

[71] S. B. Akers. "Binary Decision Diagrams." In: *IEEE Transactions on Computers* 27.6 (1978), pp. 509–516 (cit. on pp. 65, 66, 79).

[72] Randal E. Bryant. "Symbolic Boolean Manipulation with Ordered Binary-decision Diagrams." In: *ACM Computing Surveys* 24.3 (1992), pp. 293–318 (cit. on pp. 65, 66, 76, 77).

[73] Robert Tarjan. "Depth-first search and linear graph algorithms." In: *12th Annual Symposium on Switching and Automata Theory*. 1971, pp. 114–121 (cit. on p. 72).

[74] E. Randal Bryant. "Graph-Based Algorithms for Boolean Function Manipulation." In: *IEEE Transactions on Computers* 35.8 (1986), pp. 677–691 (cit. on pp. 76, 77).

[75] Henrik Reif Andersen. "An introduction to binary decision diagrams." In: *Lecture notes, available online, IT University of Copenhagen* (1997) (cit. on p. 76).

[76] Claude.E. Shannon. "The synthesis of two-terminal switching circuits." In: *The Bell System Technical Journal* 28.1 (1949), pp. 59–98 (cit. on p. 77).

[77] Elaine J. Weyuker, Tarak Goradia, and Ashutosh Singh. "Automatically generating test data from a Boolean specification." In: *IEEE Transactions on Software Engineering* 20.5 (1994), pp. 353–363 (cit. on pp. 77, 80, 81).

[78] Neil Walkinshaw. "The Practical Assessment of Test Sets with Inductive Inference Techniques." In: *Proceedings of the 5th Int. Academic and Industrial Conference on Testing - Practice and Research Techniques (TAIC PART)*. 2010, pp. 165–172 (cit. on p. 80).

[79] T.Y. Chen, M.F. Lau, and Y.T. Yu. "MUMCUT: a fault-based strategy for testing Boolean specifications." In: *Proceedings of the Sixth Asia Pacific Software Engineering Conference.* 1999, pp. 606–613 (cit. on p. 80).

[80] S. J. Friedman and K. J. Supowit. "Finding the Optimal Variable Ordering for Binary Decision Diagrams." In: *Proceedings of the 24th ACM/IEEE Design Automation Conference.* 1987, pp. 348–356 (cit. on p. 81).

[81] Orna Grumberg, Shlomi Livne, and Shaul Markovitch. "Learning to Order BDD Variables in Verification." In: *Journal of Artificial Intelligence Research* 18.1 (2003), pp. 83–116 (cit. on p. 81).

[82] Bernhard Steffen, Falk Howar, and Maik Merten. "Introduction to active automata learning from a practical perspective." In: *Formal Methods for Eternal Networked Software Systems*. Springer, 2011, pp. 256–296 (cit. on p. 81).

[83] Karl Meinke and Muddassar A Sindhu. "Incremental learning-based testing for reactive systems." In: *Tests and Proofs*. Springer, 2011, pp. 134–151 (cit. on p. 81).

[84] Muzammil Shahbaz and Roland Groz. "Inferring mealy machines." In: *FM 2009: Formal Methods*. Springer, 2009, pp. 207–222 (cit. on p. 81).

[85] Neil Walkinshaw, Ramsay Taylor, and John Derrick. "Inferring extended finite state machine models from software executions." In: *Empirical Software Engineering* (2015), pp. 1–43 (cit. on p. 81).

[86] Christopher Henard, Mike Papadakis, and Yves Le Traon. "MutaLog: A Tool for Mutating Logic Formulas." In: *Proceedings of the 7th IEEE International Conference on Software Testing, Verification, and Validation Workshops*. IEEE, 2014, pp. 399–404 (cit. on p. 82).

[87] Man F. Lau and Yuen T. Yu. "An Extended Fault Class Hierarchy for Specification-based Testing." In: *ACM Transactions on Software Engineering and Methodology* 14.3 (2005), pp. 247–276 (cit. on p. 82).

[88] T. K. Paul and M. F. Lau. "Redefinition of Fault Classes in Logic Expressions." In: *Proceedings of the 12th International Conference on Quality Software*. IEEE, 2012, pp. 144–153 (cit. on p. 82).

[89] Edmund M. Clarke, Masahiro Fujita, and Xudong Zhao. "Multi-Terminal Binary Decision Diagrams and Hybrid Decision Diagrams." In: *Representations of Discrete Functions*. Springer, 1996, pp. 93–108 (cit. on p. 82).

[90] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. "Hints on Test Data Selection: Help for the Practicing Programmer." In: *Computer* 11.4 (1978), pp. 34–41 (cit. on p. 85).

[91] Paul Ammann, Jeff Offutt, and Hong Huang. "Coverage Criteria for Logical Expressions." In: *Proceedings of the 14th International Symposium on Software Reliability Engineering*. IEEE, 2003 (cit. on p. 85).

[92] *Pairwise Testing - Available Tools*. `http://www.pairwise.org/tools.asp`. Accessed: 2016-09-21 (cit. on p. 89).

[93] Mark Hall et al. "The WEKA Data Mining Software: An Update." In: *SIGKDD Explorations Newsletter* 11.1 (2009), pp. 10–18 (cit. on pp. 89, 155).

[94] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. "The Program Dependence Graph and Its Use in Optimization." In: *ACM Transactions on Programming Languages and Systems* 9.3 (1987), pp. 319–349 (cit. on p. 94).

[95] Ian H Witten and Eibe Frank. *Data Mining: Practical machine learning tools and techniques*. 2005 (cit. on p. 103).

[96] Laleh Sh Ghandehari et al. "An empirical comparison of combinatorial and random testing." In: *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2014, pp. 68–77 (cit. on p. 105).

[97] Monica Hutchins et al. "Experiments of the Effectiveness of Dataflow- and Controlflow-based Test Adequacy Criteria." In: *Proceedings of the 16th International Conference on Software Engineering*. IEEE Computer Society Press, 1994, pp. 191–200 (cit. on p. 105).

[98] S. R. Dalal et al. "Model-based Testing in Practice." In: *Proceedings of the 21st International Conference on Software Engineering*. ACM, 1999, pp. 285–294 (cit. on p. 106).

[99] I. S. Dunietz et al. "Applying Design of Experiments to Software Testing: Experience Report." In: *Proceedings of the 19th International Conference on Software Engineering*. ACM, 1997, pp. 205–215 (cit. on p. 106).

[100] D. R. Kuhn, D. R. Wallace, and A. M. Gallo Jr. "Software Fault Interactions and Implications for Software Testing." In: *IEEE Transactions on Software Engineering* 30.6 (2004), pp. 418–421 (cit. on p. 106).

[101] Christopher Henard et al. "Comparing White-box and Black-box Test Prioritization." In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE '16. Austin, Texas: ACM, 2016, pp. 523–534 (cit. on p. 107).

[102] I. Pill et al. "Analyzing the reduction of test suite redundancy." In: *2015 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 2015, pp. 65–65 (cit. on p. 111).

[103] Gregg Rothermel and Mary Jean Harrold. "A Safe, Efficient Regression Test Selection Technique." In: *ACM Transactions on Software Engineering and Methodology* 6.2 (1997), pp. 173–210 (cit. on p. 111).

[104] Tsong Yueh Chen and Man Fai Lau. "Dividing Strategies for the Optimization of a Test Suite." In: *Information Processing Letters* 60.3 (1996), pp. 135–141 (cit. on p. 111).

[105] A. Jefferson Offutt, Jie Pan, and Jeffrey M. Voas. "Procedures for reducing the size of coverage-based test sets." In: *Proceedings of the 12th International Conference on Testing Computer Software*. ACM, 1995, pp. 111–123 (cit. on p. 111).

[106] G. Rothermel et al. "An empirical study of the effects of minimization on the fault detection capabilities of test suites." In: *Proceedings of the International Conference on Software Maintenance*. 1998, pp. 34–43 (cit. on p. 111).

[107] W. Eric Wong et al. "Effect of Test Set Minimization on Fault Detection Effectiveness." In: *Proceedings of the 17th International Conference on Software Engineering*. ACM, 1995, pp. 41–50 (cit. on p. 111).

[108] B. Korel, L. H. Tahat, and B. Vaysburg. "Model based regression test reduction using dependence analysis." In: *Proceedings of the International Conference on Software Maintenance*. 2002, pp. 214–223 (cit. on p. 112).

[109] Lionel C. Briand, Yvan Labiche, and Zaheer Bawar. "Using Machine Learning to Refine Black-Box Test Specifications and Test Suites." In: *8th International Conference on Quality Software*. 2008, pp. 135–144 (cit. on pp. 113, 142, 160, 161).

[110] Jun-Wei Lin and Chin-Yu Huang. "Analysis of Test Suite Reduction with Enhanced Tie-breaking Techniques." In: *Information and Software Technology* 51.4 (2009), pp. 679–690 (cit. on pp. 113, 142).

[111] T.Y. Chen and M.F. Lau. "A new heuristic for test suite reduction." In: *Information and Software Technology* 40.5-6 (1998), pp. 347–354 (cit. on p. 142).

[112] Ramsay Taylor et al. "Using Behaviour Inference to Optimise Regression Test Sets." In: *Testing Software and Systems*. 2012, pp. 184–199 (cit. on p. 142).

[113] Swarnendu Biswas et al. "Regression Test Selection Techniques: A Survey." In: *Informatica* 35.3 (2011), pp. 289–321 (cit. on pp. 143, 161).

[114] Laurent Hyafil and Ronald Rivest. "Constructing optimal binary decision trees is NP-complete." In: *Information Proc. Letters* 5.1 (1976), pp. 15–17 (cit. on p. 146).

[115] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., 2009 (cit. on p. 148).

[116] Leonardo De Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver." In: *Proceedings of 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer, 2008, pp. 337–340 (cit. on p. 155).

[117] S. R. Safavian and D. Landgrebe. "A survey of decision tree classifier methodology." In: *IEEE Transactions on Systems, Man, and Cybernetics* 21.3 (1991), pp. 660–674 (cit. on p. 159).

[118] Bernard ME Moret. "Decision trees and diagrams." In: *ACM Computing Surveys (CSUR)* 14.4 (1982), pp. 593–623 (cit. on p. 159).

[119] G.R. Dattatreya and L.N. Kanal. "Progress in Pattern Recognition 2." In: Elsevier Science Publishers B.V., 1985. Chap. Decision Trees in Pattern Recognition, pp. 189–240 (cit. on p. 159).

[120] J.R.B. Cockett. "Discrete decision theory: manipulations." In: *Theoretical Computer Science* 54.2 (1987), pp. 215–236 (cit. on p. 160).

[121] Hans Zantema. "Decision trees: Equivalence and propositional operations." In: *10th Netherlands/Belgium Conf. on AI (NAIC)*. 1998, pp. 157–166 (cit. on p. 160).

[122] J. R. B. Cockett and J. A. Herrera. "Decision Tree Reduction." In: *Journal of the ACM* 37.4 (1990), pp. 815–842 (cit. on p. 160).

[123] Hans Zantema and Hans L. Bodlaender. "Finding small equivalent decision trees is hard." In: *International Journal of Foundations of computer science* 11.2 (2000), pp. 343–354 (cit. on p. 160).

[124] Leonard A. Breslow and David W. Aha. "Simplifying Decision Trees: A Survey." In: *The Knowledge Engineering Review* 12.1 (1997), pp. 1–40 (cit. on p. 160).

[125] T. J. Ostrand and M. J. Balcer. "The Category-partition Method for Specifying and Generating Functional Tests." In: *Communications of the ACM* 31.6 (1988), pp. 676–686 (cit. on p. 160).