Master's Thesis

Formal Methods for the Analysis of Program Mutations

Heinz Riener

Institute for Applied Information Processing and Communications
Graz University of Technology, Austria

Advisor: Prof. Roderick Bloem

Graz, August 2011

Masterarbeit

Formale Methoden zur Analyse von Programm-Mutationen

Heinz Riener

Institut für Angewandte Informationsverarbeitung und Kommunikationstechnologie
Technische Universität Graz, Österreich

Begutachter: Prof. Roderick Bloem

Graz, im August 2011

Diese Arbeit ist in englischer Sprache verfasst.

Abstract

Mutation testing is a powerful testing methodology: the source code of a program is systematically seeded with syntactic changes (*faults*) and tested. Undetected faults indicate insufficient testing and can be used to improve the test bench. Some syntactic changes do not affect the program semantics — a program with this kind of faults is functionally equivalent to the original program. Deciding whether two arbitrary programs are functionally equivalent is undecidable, even if they are closely related and differ only in a single syntactic change. The impossibility to decide whether a seeded fault affects the program semantics circumvents counting how many faults remain undetected. This obstacle limits the practical applicability of mutation testing. A tester will always wonder whether the undetected faults cannot be found because they do not affect the program semantics or the test bench is insufficient in uncovering them.

In this thesis, we focus on the problem of deciding whether a seeded syntactic change affects the program semantics with the aid of formal methods.

We present a mutation testing approach that seeds faults into the intermediate representation of a compiler and, thus, is independent from the programming language of the compiler front-end. We show the construction of a meta-mutant for the intermediate representation that serves as an effective data-structure to represent a set of mutants. The meta-mutant collects a set of seeded faults in a single program with additional control logic which provides mechanisms to enable and disable the individual faults.

We develop two orthogonal approaches to decide functional equivalence for special cases of programs that differ in a single syntactic change. First, we propose a new code optimization approach using compiler construction techniques based on static analysis to decide functional equivalence of some seeded faults. Second, we present a bounded model checking approach which builds a model of the program and searches for counterexamples of finite length in the model. Each counterexample proves functional non-equivalence of a seeded fault. From a counterexample a test case can be extracted which results in different externally observable outputs for the original program and the program with the seeded syntactic change.

Finally, we present experimental results of our approaches for ANSI-C with a prototype implementation.

Keywords: Mutation Testing, Formal Methods, Functional Equivalence, Bounded Model Checking.

Kurzfassung

Mutationstesten ist eine mächtige Testmethodologie: In ein Programm werden systematisch, syntaktische Änderungen (*Fehler*) eingestreut und das Programm wird dann getestet. Unentdeckte Fehler deuten auf unzureichendes Testen hin und können zur Verbesserung der Testmenge verwendet werden. Einige syntaktische Änderungen wirken sich nicht auf die Programmsemantik aus — ein Programm mit dieser Art von Fehlern ist funktional äquivalent zum Originalprogramm. Die Entscheidung, ob zwei beliebige Programme funktional äquivalent sind, ist unentscheidbar, auch wenn die Programme in einem engen Zusammenhang stehen und sich nur in einer einzigen syntaktischen Änderung unterscheiden. Die Unmöglichkeit zu entscheiden, ob sich ein eingestreuter Fehler auf die Programmsemantik auswirkt, verhindert es, exakt zu bestimmen, wie viele Fehler nicht gefunden wurden. Dieses Problem limitiert die praktische Einsatzfähigkeit von Mutationstesten. Ein Tester kann sich nicht sicher sein, ob die unentdeckten Fehler nicht gefunden werden können, weil sie sich nicht auf die Programmsemantik auswirken, oder ob die Testmenge unzureichend testet, um die Fehler zu finden.

In dieser Arbeit befassen wir uns mit Hilfe von formalen Methoden mit dem Problem zu entscheiden, ob eine eingestreute syntaktische Änderung Einfluss auf die Programmsemantik hat.

Wir präsentieren einen Ansatz für Mutationstesten, der Fehler in die Zwischenrepräsentation eines Compilers einschleust, und deshalb unabhängig von der Programmiersprache des Compiler Front-Ends eingesetzt werden kann. Wir zeigen, wie ein Meta-Mutant auf der Ebene der Zwischenrepräsentation konstruiert wird. Der Meta-Mutant dient als effiziente Datenstruktur, die eine Menge von Fehlern in einem einzigen Programm sammelt, und mit Hilfe von zusätzlicher Kontrolllogik erlaubt einzelne Fehler ein- und auszuschalten.

Wir entwickeln zwei orthogonale Ansätze, um die funktionale Äquivalenz von Spezialfällen von Programmen zu entscheiden, die sich nur in einer einzigen syntaktischen Änderung unterscheiden. Erstens schlagen wir einen neuen Code-Optimierungsansatz vor, der Techniken aus dem Compilerbau basierend auf statischer Analyse verwendet, um funktionale Äquivalenz für einige eingestreute Fehler zu entscheiden. Zweitens präsentieren wir einen „beschränkten“ Modellprüfungsansatz (bounded model checking), der ein Modell eines Programmes baut und nach Gegenbeispielen endlicher Länge in dem Modell sucht. Jedes Gegenbeispiel beweist, dass ein eingestreuter Fehler die Semantik des Originalprogramms beeinflusst. Aus einem Gegenbeispiel kann ein Testfall extrahiert werden, der verschiedene extern beobachtbare Ausgaben für das Originalprogramm und das Programm mit dem eingestreuten Fehler bedingt.

Schließlich präsentieren wir experimentelle Ergebnnisse für unsere Ansätze für ANSI-C mit einer Prototyp-Implementierung.

Schlüsselworte: Mutationstesten, formale Methoden, funktionale Äquivalenz, gebundene Modellprüfung.

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

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 entnommene Stellen als solche kenntlich gemacht habe.

Graz, am …………………………… ………………………………………………..
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

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 marked all material which has been quoted either literally or by content from the used sources.

…………………………… ………………………………………………..
date (signature)

Acknowledgments

I am very grateful to my advisor Roderick Bloem, who continuously supported my work and has spent numerous hours for discussing and reviewing my premature ideas.

I would like to thank my colleague and friends at Graz, University of Technology for having a great time during studying.

Last but not least, I would like to thank my parents (and family) for their moral and financial support. They always believed in me and my decision to study Telematics.

Heinz Riener
Graz, Austria, August 2011

Danksagung

Ich möchte meinem Betreuer Roderick Bloem danken, der meine Arbeit stets unterstützt und zahlreiche Stunden geopfert hat, um unausgereifte Ideen zu diskutieren und zu kritisieren.

Auch sehr herzlich möchte ich meinen Kolleginnen und Kollegen an der Technischen Universität Graz für eine tolle Zeit während des Studiums danken.

Zu guter Letzt will ich meinen Eltern und meiner Familie für die moralische und finanzielle Unterstützung danken. Sie haben immer an mich und meine Entscheidung Telematik zu studieren geglaubt.

Heinz Riener
Graz, Österreich, August 2011

Contents

Chapter 1

Introduction

1.1 Motivation

Computer systems are an important and integral part of our everyday lives. Malfunctions of safety critical systems, e.g., in medical devices or automobiles, cause high costs in form of claims for damages, undermine a company's image, or endanger human lives. Some malfunctions, that had serious consequences, are the FDIV bug of the Intel® Pentium® P5 floating point unit which has infected some of the floating point calculations carried out by the processor [Pra95], and the launch failure of the Ariane 5 rocket which has led to its destruction only 37 seconds after start [Lio96]. In both cases a technical defect caused a financial damage of several hundred million US dollars. Another example is the malfunctioning of the Therac-25 medical accelerator which has demonstrably killed six people and injured several others [LT93]. The higher the importance of computer systems, the higher the price when they fail. Better techniques to derive confidence in the correct functioning of computer systems are desirable.

Testing and simulation techniques are the predominant methods used today to increase the confidence in the quality and reliability of complex digital systems. They examine the functional behavior of a system by stimulating the inputs of the system and observing the outputs and the behavior of the system during execution. If either an output or the observed behavior of the system does not conform to the expectations of the developers (or testers), a *failure* in the implementation of the system is found. The developers then deal with the tasks of *localizing* and *correcting* the corresponding *faults* in the implementation that caused the failure.

Testing and simulation examine "some" input stimuli from a huge (or even infinite) domain and, thus, they may miss failing executions of an implementation. This incompleteness property of testing and simulation has been expressed by Dijkstra's claim: "testing [and simulation] can show the presence of bugs but never their absence" [DDH72].

In this thesis, we focus on the testing of software systems. Testing software is a complicated and costly task. Already in the late 1970s and early 1980s software testing was recognized to account for more than half of the total development costs [Mye79, Boe81]. Nonetheless software systems are often inadequately tested. For instance, the annual costs of inadequate testing in the US is estimated to range from 22 to 59 billion US dollars. It is believed that over one half of these costs can be saved by the introduction of better testing infrastructure [Tas02].

One of the first places to detect faults in an implementation of a software system is *unit testing*. In unit testing a developer writes some selected test cases to check the functional behavior of an individual unit of the implementation, i.e., a single function or a module. The

```
1  TriType f(float a,float b,float c){
2    if (a == b && b == c)
3      return Equilateral;
4    if (a == b || b == c || b == a)
5      return Isosceles;
6    return Scalene;
7  }
```

a	b	c	Expected Output
2.0	2.0	2.0	Equilateral
3.0	4.0	5.0	Scalene
2.0	2.0	3.0	Isosceles

Figure 1.1: The function `f` (on the left side) decides for three given side-lengths a, b, and c of a triangle whether the corresponding triangle is equilateral, isosceles, or scalene. The table (on the right side) shows three test cases which are fully adequate with respect to the statement coverage, branch coverage, and path coverage criteria. The source code of the function f, however, has a typo. The third decision `b == a` in line 4 should read `c == a`.

expectations of the developer serve as an implicit, partial specification.

However, the decision when the software unit under test has been adequately tested, i.e., when to stop writing new test cases cannot be taken without a formal test criterion. The test criterion provides a quantitative measure of the quality of the test cases. Thus, the test criterion serves as a *stopping rule* for test case generation.

1.2 Background

1.2.1 Test Criteria

Only *exhaustive testing*, i.e., the execution of an implementation on all possible inputs can guarantee functional correctness of the implementation. The number of possible inputs of computer systems, however, are enormous or even infinite when input-dependent data-structures are used. Thus, generally, exhaustive testing is impossible [DDH72].

The notion of *test criteria* [GG75] has been introduced to support the selection of test cases by measuring how well a system is tested. A test criterion imposes a set of *test requirements* on the test bench that "good" test cases should examine, and measures in terms of *test coverage* how many of them are actually satisfied. The *coverage measure* (or *coverage score*) is the ratio of fulfilled test requirements to the total number of test requirements. It quantifies the fulfillment in form of a real value in the interval $[0, 1]$, which is commonly expressed as a percentage. The test bench is said to be *fully adequate* to the test criterion if the test bench achieves a coverage measure of 1.0 (100%) and *partially adequate* otherwise.

Academics provided a large variety of different test criteria each having their own advantages [AO08]. Classical test criteria are *statement coverage*, *branch coverage*, and *path coverage*. The test requirements of the classical test criteria necessities the execution of particular parts of a program during testing. The classical criteria, however, have been criticized because they do not force the individual test cases to examine the functional behavior of the program code executed during testing. For instance, consider the example code shown in Figure 1.1. The function `f` (on the left side) decides for the given three side-lengths a, b, and c of a triangle whether the corresponding triangle is equilateral, isosceles, or scalene. We assume a data type `TriType` has been declared and a variable of this data type can take one of the values `equilateral`, `isosceles`, and `scalene`. The source code of the function is taken from Renieris and Reiss [RR03]. The table (on the right side) shows three test cases (inputs and expected outputs) which are fully adequate with respect to the statement coverage, branch coverage, and path coverage criteria, i.e., all statement, branches, and paths are examined when all three test cases are executed in testing. However, the source code of the function `f` has a typo, i.e., a programmer erroneously has used the check `b == a` rather than `c == a` in the last decision of the condition in line 4. Thus, the function `f` returns `scalene` rather than `isosceles` when executed on inputs v_a, v_b, and v_c, with $v_a = v_c$ and $v_a \neq v_b$, e.g., $v_a = 2.0$, $v_b = 3.0$, and $v_c = 2.0$.

1.2.2 Mutation Testing

Mutation testing [DLS78, Ham77] is a powerful testing methodology based on the idea of making changes to a syntactic description of a computing task and deriving test cases from these changes. The changes mimic mistakes programmers or designers make during the description of the computing task. In the context of program-based mutation testing, the source code of a program, written in a high-level programming language, is seeded with artificial faults and then systematically executed on each test case of a test bench. Undetected faults indicate holes in the test bench and can be used to improve it.

We stick to the common mutation testing terminology: a program is seeded with *faults*, where a fault is a single syntactic change relative to the original program. For instance, an addition operator is turned into a subtraction operator. The unchanged program (without the seeded faults) is called the *original program*. A *mutant* is a duplicate of the original program containing exactly one fault. For the sake of simplicity, we use the terms fault and mutant interchangeable when the exact meaning is clear from the context. In the literature, faults and mutants corresponding to single syntactic changes are sometimes called *first order faults* and *first order mutants*. In this thesis, we focus on mutation testing considering only first order mutants.

Mutation testing provides a fault-based test criterion, called *mutation adequacy*, which attempts to quantify the fault-finding abilities of a test bench by measuring the number of faults *detected* by the test bench relative to the number of faults seeded into a program. The corresponding coverage measure is called *mutation score*. A test case *detects* (or *kills*) a fault if it results in a different externally observable behavior when executed on the mutant and the original program, respectively. Consequently, the corresponding mutant (and fault) is then said to be detected (or killed) by the test case. The meaning of "externally observable behavior" depends on the application of mutation testing. For instance, an externally observable behavior is the return value of a function or the value of a global program variable.

Mutation adequacy is a "stronger" test criterion compared to classical coverage criteria [AO08]. A test case only contributes to the mutation score if it is able to distinguish the mutant from the original program. Thus, the test case must examine the value of at least one variable affected by a seeded fault to detect a mutant.

A fault which changes the syntax of a program may not affect the semantics of the program, i.e., the corresponding mutant is functionally equivalent to the original program. We call these faults and the corresponding mutants *functionally equivalent* (or *equivalent* for short). Thus, a seeded fault which affects the program semantics is *non-equivalent*. Moreover, a mutant may also be functionally equivalent to another mutant. We call these mutants and the corresponding faults *redundant*. Functionally equivalent and redundant mutants are obstacles in mutation testing which we would like to avoid. A functionally equivalent mutant corresponds to an unsatisfiable test requirement because it is impossible to find a test case that detects the mutant. A redundant mutant corresponds to a redundant test requirement. A test case that detects one

```
while ( C ) {
  B;
}
```

```
if ( C ) {
  B;
  if ( C ) {
    B;
    assert( !C );
  }
}
```

Figure 1.2: The pseudo code of a loop (on the left) and a two-times unrolled version of the loop (on the right).

of a set of pairwise mutually redundant mutants will also detect the other mutants from the same set.

The effectiveness of mutation testing has been shown by analytic comparison to other criteria [MW94b, OV96, AO08] and is additionally supported by numerous empirical studies [FW93, MW94a, OPTZ96, LPO09].

1.2.3 Formal Methods

Besides traditional testing, several formal methods have been proposed to detect failures in implementations of systems. One of these methods is *model checking* [CGP99]. Given a finite state-transition system M representing the implementation under test and a logic formula φ in temporal logic (a specification), a model checking algorithm proves whether the state-transition system M is a model for the formula φ, i.e., $M \models \varphi$. The algorithm exhaustively explores the state space of the implementation and verifies whether one of the states violates the property to be checked.

In general, model checking suffers from the large number of potential execution states which is typically referred to as the *state explosion problem.* Explicitly representing each state is possible but it was not until the introduction of symbolic algorithms that model checking became practical. Symbolic algorithms compress larger state sets into a symbolic representation to make the approach applicable to larger systems. McMillan [McM94] suggested the usage of *Ordered Binary Decision Diagrams* (OBDD) [Bry86] to represent and manipulate the state-transition system. Biere et al. [BCCZ99] proposed SAT-based *Bounded Model Checking* (BMC) as a complementary approach to OBDD-based model checking.

In BMC, the state-transition system is first *unfolded* for a fixed number of steps k and then encoded in conjunction with the negated specification $\neg\varphi$ as an instance of the *satisfiability problem* (SAT), i.e., a logic formula in Conjunctive Normal Form (CNF) which is checked for satisfiability. A satisfying assignment corresponds to a counterexample of finite length (smaller or equal k) that refutes the specification. If the formula is unsatisfiable, no such counterexamples exists in the unfolded model. Then, either the number of unfolding steps k was chosen too small or the implementation conforms to its specification.

BMC is an effective technique to find counterexamples of finite length. However, proving the correctness of a property with BMC means to increase k until a *completeness threshold* is reached, i.e., an upper bound k' such that if no counterexample of length $k < k'$ is found, the property holds. Finding the completeness threshold, however, is complicated and the resulting value is often quite large (in case of software even infinite) and, thus, BMC can be considered an inherently incomplete approach.

BMC was originally proposed as a hardware verification technique but it was later extended to software verification [CKOS04, ISGG05]. In software, unfolding the program corresponds to a recursive unrolling of the loops in the program, i.e., the body of each loop is k-times replicated, where k is a finite loop bound, and each time guarded by a conditional statement that checks whether the loop condition is fulfilled. Figure 1.2 [BHvMW09] schematically illustrates loop unrolling. The loop on the left side is two-times unrolled and results in the unrolled version on the right side. We denote the loop condition and the loop body by C and B, respectively. After unrolling the loop two-times, we add an assertion that checks whether the loop is left after two iterations. This assertion ensures that in case of insufficient unrolling, the spurious branch is not reachable.

More recently, Armando et al. [AMP09] used an Satisfiability Modulo Theories (SMT) solver rather than a SAT solver to encode BMC which improves the scalability in case of software programs with complex arithmetic or array manipulations. The SMT solver implements an efficient decision procedures for large instances of constraint satisfaction problems according to some specific background theories. An SMT solver either operates on top of a SAT solver (*eager approach*) or uses a solver specialized for the background theory (*lazy approach*). SMT provide high-level semantics to directly encode and manipulate word-level operations (e.g., addition, if-then-else, etc.). The resulting formulae are more compact and allow for a better maintenance of their meaning. For instance, an integer multiplication results in a set of complex constraints when encoded into a SAT formula because only Boolean operators are allowed. Contrarily, SMT theories often directly support integer multiplication (e.g., theory of bit-vectors or theory of non-linear integer arithmetic).

1.3 Problem Addressed in the Thesis

The obstacles in mutation testing are twofold [OU00, JH10]: (1) the number of possible faults to be seeded is infinite and an effort to check a reasonable finite subset by explicitly enumerating each fault is costly. (2) Equivalent mutants do not affect the program semantics. They show the same externally observable behavior as the original program and, thus, no test case exists that can detect an equivalent mutant.

Equivalent mutants cause a systematic under-estimation of the mutation score, i.e., the total number of mutants is, in general, higher than the number of mutants which can be detected during testing.

Most of the research in mutation testing focuses on the first problem [OU00, JH10] which deals with the computational expense of mutation testing. Generally, the proposed techniques to overcome the computation expense target either the reduction of the number of created mutants or the reduction of the execution cost that is caused by running the test cases on each individual mutant.

A pragmatic solution to the first problem is the usage of more or faster computers. The second problem is more significant because it describes a theoretical limitation of mutation testing: without the detection of equivalent mutants a mutation score of 100% is not possible and a test engineer will always wonder whether the undetected mutants are equivalent to the original program or the test bench is insufficient in detecting them.

In this thesis, we focus on the detection of equivalent mutants. We call the problem to decide whether a mutant is functionally equivalent to its original program the *equivalent mutant problem*. The equivalent mutant problem is a decision problem which can be answered with either ‘Yes’ or ‘No’. Unfortunately, deciding the equivalent mutant problem, i.e., giving the correct answer is a subtle task: on the one hand, studies have shown that manually separating a given set of mutants into the categories “equivalent” and “non-equivalent” is time-consuming [Acr80] and complicated [SZ10]. The participants of the studies tend to put the mutants into the wrong category and they needed a significant amount of time to make their decisions. On the other hand, deciding the equivalent mutant problem algorithmically is impossible, i.e., the equivalent mutant problem is undecidable [BA82]. Thus, generally, no decision procedure exists to decide whether a mutant is functionally equivalent to its original program.

A (first order) mutant, however, is per definition derived from its original program by seeding a single syntactic change into the source code of the program. The mutant and the original program are closely related. They differ only in a single syntactic change and, thus, by neglecting completeness the equivalent mutant problem has the potential of being decidable in several cases.

1.4 Outline of the Solution

We address the equivalent mutant problem from two sides: firstly, we use methods from compiler construction based on static analysis that aim to detect some equivalent mutants. Secondly, we use a counterexample generation approach to decide which mutants are guaranteed to be non-equivalent.

The first attempt reduces the total number of seeded faults by the number of detected equivalent mutants, i.e., the denominator of the mutation score is decreased. The second attempt detects mutants to be non-equivalent, i.e., the numerator of the mutation score is increased. The difference between the numerator and denominator gives a quality indication of our approach. We refer to the two approaches to partially solve the equivalent mutant problem as the *code optimization approach* and the *counterexample approach*.

The code optimization approach uses an optimization pipeline to transform the original program and a mutant into optimized program versions. The optimization pipeline consists of a sequence of optimizing source-to-source transformations which are consecutively applied to the original program and the mutant. We conclude functional equivalence of the original program and the mutant if the obtained optimized versions are syntactically equivalent.

The counterexample approach uses BMC to generate a finite model of the original program and a mutant. Both programs are unfolded for a fixed number of steps and then encoded into a single logic formula. In the resulting formula, we constrain the variables corresponding to the program inputs of the original program and the mutant, respectively, to be equal and assert the variables corresponding to the outputs of the original program and the mutant to be different. A satisfying assignment (if one exists) serves as a counterexample that effectively disproves functional equivalence of the original program and the mutant.

We present two decision procedures, MetaOptimizer and SymBMC, corresponding to the code optimization and the counterexample approach. The decision procedures take their equivalence and non-equivalence decisions for the original program and a set of mutants (rather than a single mutant) encoded in a meta-mutant [UOH93]. The meta-mutant serves as an effective data-structure to reason about a set of mutants.

We built prototype implementations of the MetaOptimize and the SymBMC procedures. The implementations operate on the *Low Level Virtual Machine* (LLVM) intermediate representation [Lat02], i.e., a programming language with a RISC-like assembly instruction set. We use an ANSI-C language front-end compiler (`llvm-gcc`) from the LLVM infrastructure to transform

ANSI-C code to LLVM. However, our implementation is not tied on the programming language ANSI-C but can handle any language an LLVM front-end compiler exists (in particular also C++ programs). The prototype implementation of the MetaOptimizer procedure generates a logic formula over the theory of bit-vectors from the program and uses an SMT solver (in particular Boolector [BB09] and Z3 [MB08]) as back-end to find a satisfying assignment for the formula.

The contribution of the thesis is as follows:

- We present a simple fault model for the LLVM intermediate representation and show how a meta-mutant is constructed in LLVM.
- We give a new code optimization approach that attempts to detect some equivalent and redundant faults for the LLVM meta-mutant.
- We apply symbolic bounded model checking to the LLVM meta-mutant to construct one counterexample (test case) for each non-equivalent fault represented in the meta-mutant. The counterexample disproves functional equivalence of the original program and the corresponding mutant. This counterexample approach is based on the work in [RBF11].
- We implement both approaches into a tool and show initial results of the equivalence and non-equivalence detections provided by the code optimization approach and the counterexample approach.

1.5 Structure of the Document

The remainder of the thesis is organized as follows: in Chapter 2, we explain the techniques underlying our approaches and establish notation. We start with a formalization of software testing. Then, we introduce mutation testing and bounded model checking in detail. We discuss a subset of the LLVM intermediate representation which is used in the later chapters. We give a simple fault model for LLVM and show the construction of the meta-mutant specific for the LLVM language.

In Chapter 3, we describe an approach to detect equivalent and redundant mutants using static analysis methods leveraging source-to-source transformations from an optimizing compiler. We give an initial motivation for this approach using the *GNU Compiler Collection* (GCC) and the Unix tool *diff*. We present the Optimizer and MetaOptimizer procedure. The Optimizer procedure attempts to show functional equivalence of a program and one of its mutants. The MetaOptimizer procedure generalizes the approach and aims to detect equivalent and redundant faults encoded as a meta-mutant. Both procedures are built on optimizing source-to-source transformations provided by the LLVM compiler infrastructure.

In Chapter 4, we describe an approach that searches for a counterexamples that shows non-equivalence of the original program and its mutants. We present the procedure SymBMC which uses bounded model checking to generate a finite model by unfolding the meta-mutant and then encodes model into an SMT formula. We start the description with a simplified version of SymBMC, called SimplifiedBMC, which searches for a counterexample of a program and one of its mutants. We then discuss unfolding and encoding of the program as a bit-vector formula.

Chapter 5 presents experimental results for the procedures MetaOptimizer and SymBMC. First, we list the numbers of detected equivalent and non-equivalent faults for each procedures separately. Then, we give experimental results when both procedures are applied one after another.

Chapter 6 concludes the thesis. We summarize the code optimization and the counterexample approach and outline future work.

Chapter 2

Preliminaries

2.1 Introduction

In the following section, we explain the techniques underlying our equivalence and non-equivalence detection approaches and establish notation: we start with the formalization of testing (Section 2.2.1) and discuss two testing perspectives (Section 2.2.2). We then present mutation adequacy (Section 2.2.3), i.e., the fault-based test criterion underlying mutation testing which attempts to quantify the fault finding abilities of a test bench. We discuss mutation testing in detail (Section 2.3). We introduce mutation analysis (Section 2.3.1), i.e., a fault-based method to measure the quality of a given test bench. We discuss why (Section 2.3.2) and how (Section 2.3.3) mutation testing works and, lastly, introduce the concept of a mutation operator (Section 2.3.4) and the idea of a meta-mutant (Section 2.3.5).

Next, we present SAT-based Bounded Model Checking (BMC) (Section 2.4). BMC is an effective formal technique to find counterexample of finite length which we will use to generate test cases in later chapters.

Lastly, we introduce the LLVM intermediate representation (Section 2.5): We give an overview of LLVM and its characteristics (Section 2.5.1). Then, we discuss the structure of an LLVM program (Section 2.5.2) and the syntax and semantics of individual LLVM instructions (Section 2.5.3). We present LLVM metadata (Section 2.5.4) which is used to attach additional information on LLVM programs, e.g., source locations from the front-end language to simplify debugging or in our case information about the seeded faults. We then show the fault model used for LLVM programs (Section 2.5.5). Finally, we deal with the construction of the meta-mutant from an LLVM program (Section 2.5.6) and summarize the chapter (Section 2.6).

2.2 Software Testing

2.2.1 A Formalization of Testing

We focus on *deterministic*, *transformational programs*. These programs have two characteristics: firstly, for given input values, transformational programs aim to calculate a final result (output values) at the end of a terminating computation. We assume, however, that transformational programs may fail to terminate due to a fault in the program. Secondly, the programs under consideration are deterministic, i.e., repeated executions of a program on the same input values yield the same output values. We denote the universe of all deterministic, transformational programs by $\mathcal{P}$.

The semantics of a transformational program $P \in \mathcal{P}$ are described by its *semantic function* $\hat{P} : \mathbb{I} \to \mathbb{O}$, where $\mathbb{I}$ and $\mathbb{O}$ are the *input domain* and the *output domain* of P. The elements of $\mathbb{I}$ and $\mathbb{O}$ are recursively enumerable. We write $\hat{P}(i) = o$ to denote that the execution of P on the input values $i \in \mathbb{I}$ yields the output values $o \in \mathbb{O}$.

A *formal specification* of a program $P \in \mathcal{P}$ defines a (binary) input-output relation $\text{Spec} \subseteq \mathbb{I} \times \mathbb{O}$, where $\mathbb{I}$ and $\mathbb{O}$ are the input domain and the output domain of the program P. We do not distinguish between a formal specification and the input-output relation defined by the formal specification. The *domain* $dom(R)$ of a set (or a binary relation) $R \subseteq X \times Y$ is the set of all $x \in X$ for which there is a $y \in Y$ such that $(x, y) \in R$.

Suppose $P \in \mathcal{P}$ is a program and Spec is a formal specification over the same input domain $\mathbb{I}$ and output domain $\mathbb{O}$. The program P and the formal specification Spec *agree* on a subset $\mathbb{I}' \subseteq \mathbb{I}$ of the input domain *if and only if* (iff) for all $i \in \mathbb{I}'$ there are output values $o \in \mathbb{O}$ in the output domain such that $\hat{P}(i) = o$ and $(i, o) \in \text{Spec}$. Otherwise, we say that P and Spec *disagree* on $\mathbb{I}'$. The program P is *correct* with respect to Spec (or P *conforms* to Spec) iff P and Spec agree on $dom(\text{Spec})$. For all $i \notin dom(\text{Spec})$ the formal specification makes no assumptions on the correct output values of the program. A formal specification is *complete* if $dom(\text{Spec}) = \mathbb{I}$ and *partial* otherwise.

A *test case* of a program $P \in \mathcal{P}$ is a pair $t = (i, o)$, where i and o denote inputs and outputs from the input domain and the output domain of P. A test case $t = (i, o)$ is said to *pass* on P if the execution of i on P yields the outputs o. Otherwise, the test case is said to *fail* on P. A *test bench* is a set of test cases. The definition of pass and fail are naturally extended on test benches: a test bench T passes on P if all test cases $t \in T$ pass on P and otherwise the test bench fails on P. A *test bench* T defines a (possibly partial) formal specification Spec_T, where Spec_T is the set of all pairs $(i, o) \in T$.

2.2.2 Test Reliability and Test Adequacy

Research in software testing [GG75, How76, BA82, DO91a] has established two testing perspectives: *test reliability* and *test adequacy*. Test reliability [How76] relates testing to program correctness, i.e., the passing of a reliable test bench implies program correctness. Test adequacy [GG75] attempts to make a test bench fail on incorrect programs, i.e., testing becomes a process of eliminating faulty programs. In this case testing has a tight relation to program equivalence.

Definition 1 (Test reliability [How76])**.** *Consider a program $P \in \mathcal{P}$ that implements a computing task described by the formal specification* Spec*. The test bench T is* reliable *for P and* Spec *iff the test bench T passes on P implies P is correct with respect to* Spec*.*

A reliable test bench fails whenever a program does not conform to its specification. The exhaustive test bench T for which $dom(\text{Spec}_T) = \mathbb{I}$ is reliable but such a test bench is hardly useful in practice. Howden [How76] argued that for all programs $P \in \mathcal{P}$ and formal specifications Spec a finite reliable test bench exists. If P conforms to Spec any test bench, e.g., the empty test bench $T = \{\}$, is reliable. If P does not conform to Spec then a test bench consisting of a single test case $t = (i, o)$ for which $\hat{P}(i) = o$ and $t \notin \text{Spec}$ is a reliable test bench for P and Spec. Although reliable test benches always exist, generally, it is neither possible to decide that a test bench is reliable nor to generate one [How76].

Notice that for all programs $P \in \mathcal{P}$ and each finite test bench T a formal specification Spec exists such that P and Spec agree on $dom(T)$ but disagree on $dom(\text{Spec}) \setminus dom(T)$. Thus, test reliability depends on both a program and a formal specification. As a consequence, a procedure to generate test data needs knowledge about the program P and its formal specification Spec.

Definition 2 (Test adequacy [GG75])**.** *Consider a program $P \in \mathcal{P}$ that implements a computing task (described by the formal specification* Spec*). The test bench T is* adequate *for P iff the test bench fails on all faulty programs $P' \in \mathcal{P}$ which are not functionally equivalent to P.*

An adequate test bench distinguishes all faulty program P' from P, i.e., for each P' there exists at least one test case $t = (i, o) \in T$ such that $\hat{P}(i) \neq \hat{P}'(i)$. The definition of test adequacy makes no use of a formal specification. However, the formal specification is implicitly required to determine the "correct" output values of the test bench. Budd and Angluin [BA82] proved that a finite adequate test bench not always exists. In practice a restricted version of test adequacy is often of interest which is discussed in the next section.

2.2.3 Mutation Adequacy

Mutation adequacy is a test criterion which approximates test adequacy by restricting the domain of faulty programs under consideration to the set of all first order mutants of a program.

Definition 3 (Mutation adequacy [DLS78])**.** *Consider a program $P \in \mathcal{P}$ that implements a computing task (described by the formal specification* Spec*) and the set $\mathcal{N}$ of all first order mutants of P under a fixed fault model. The test bench T is* mutation adequate *for P with respect to the fault model iff the test bench fails on all programs $P' \in \mathcal{N}$ which are not functionally equivalent to P.*

For every program $P \in \mathcal{P}$ and finite set $\mathcal{N}$ of mutants of P a finite adequate test bench T exists. Suppose T is an empty test bench. A program $P' \in \mathcal{N}$ is either functionally equivalent to P or there are input values $i \in \mathbb{I}$ for which $\hat{P}(i) \neq \hat{P}'(i)$. We add these input values and their output values to T. The resulting test bench T is adequate by construction and finite because $\mathcal{N}$ is finite.

Given a program P and a finite set of mutants $\mathcal{N}$, generating a finite adequate test bench and deciding adequacy of a given finite test bench is decidable iff functional equivalence for all pairs of programs of $\mathcal{N}$ is decidable. This is stated by the following theorem [BA82].

Theorem 1. *For a given program $P \in \mathcal{P}$ and a set $\mathcal{N}$ of first order mutants of P under a fixed fault model, the following three statements are equivalent:*

(a) Determining adequacy of a given test bench T is decidable.

(b) Generating some adequate test bench for P is computable.

(c) The functional equivalence problem for all programs in $\mathcal{N}$ is decidable.

Proof. It suffices to show the following directions:

"$(a) \Rightarrow (b)$" Given a procedure A to determine whether a test bench is adequate for P and $\mathcal{N}$, then an adequate test bench is generated as follows. Suppose T_i is the test bench containing the first i test cases from the domain $\mathbb{I} \times \mathbb{O}$. Since an adequate test bench for P and $\mathcal{N}$ always exists, T_i is adequate for some value i. Thus, we successively create T_i for all i starting by 0 and use A to decide whether T_i is adequate. If A returns 'Yes' then we return T_i as an adequate test bench.

"$(b) \Rightarrow (c)$" Given a computable procedure G to generate an adequate test bench for P and $\mathcal{N}$, we want to decide equivalence of two program $P, Q \in \mathcal{N}$. We use G to obtain a finite adequate test bench for P and execute the test bench on P and Q. If P and Q return the same values for all inputs from $dom(T)$ then P and Q are equivalent and otherwise they are non-equivalent.

"$(c) \Rightarrow (a)$" Given a program P, the set of mutants $\mathcal{N}$, and a procedure S that determines whether two programs from $\mathcal{N}$ are equivalent. We can use S to decide whether T is adequate. We use S to decide whether P and Q are equivalent and verify that their is a test case that returns different outputs for at least one $i \in dom(T)$ in case they are non-equivalent.

$\square$

Theorem 2.2.3 states that there is a close relation between the problem of generating adequate test benches and proving functional equivalence of programs, i.e., the former is decidable iff the latter is decidable. Proving functional equivalence, however, is for most programming systems, e.g., Turing-complete programming languages, undecidable. Thus, generating adequate test benches is undecidable.

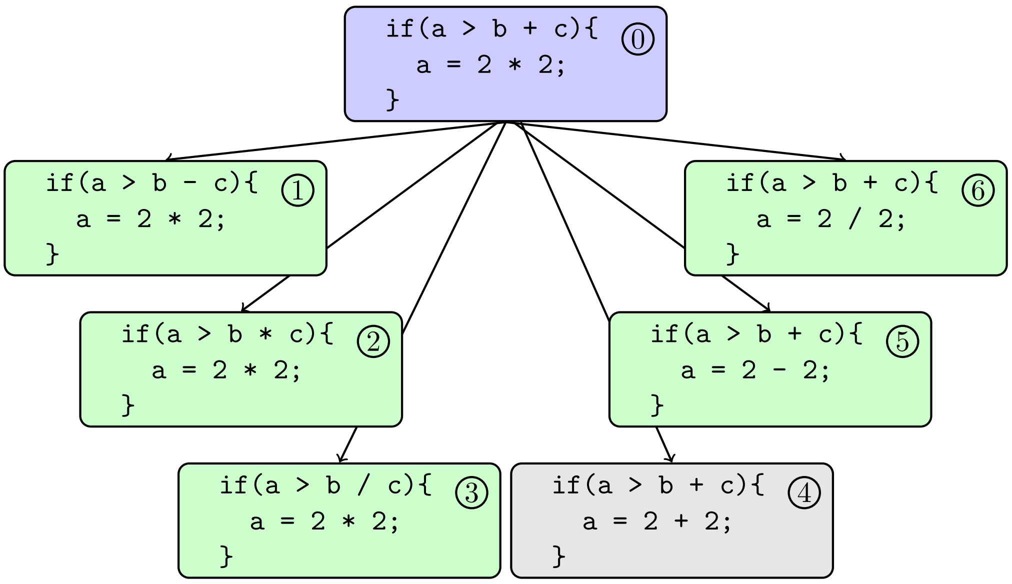
2.3 Mutation Testing

2.3.1 Mutation Analysis

Mutation adequacy can be used to analyze the fault-finding abilities of a test bench. The analysis steps are as follows: firstly, a program is seeded with artificial faults according to a fixed fault model. Each seeded fault is kept in an individual copy of the original program source (the mutant). The mutants are generated by applying *mutation operators* to the program. A mutation operator is a rule that describes how a particular syntactic pattern of the program is changed. When the mutation operator is applied to a source program, it parses the program source top-down. For each syntactic part of the program which matches the pattern of the mutation operator the source program is duplicated and the matched part of the source code is replaced by syntactically different source code resulting in new mutants of the program. Thus, a mutation operator defines a mapping from a program to a finite set of mutants of the program.

For instance, Figure 2.1 illustrates the effect of a mutation operator applied to a fragment of ANSI-C program source. Each box is labeled with a number from 0 to 6. The box on top labeled with 0 (in blue color) represents the original program code. The six boxes below labeled with 1 to 6 (in green and gray color) are mutants of the original program code. Each mutant is a copy of the original program source with a single syntactic change relative to the original program. In the example, the mutation operator has replaced arithmetic operators with other arithmetic operators. The box on the bottom right labeled with 4 (in gray color) denotes an equivalent mutant because the replacement of `a = 2 * 2` by `a = 2 + 2` does not affect the semantics of the program fragment, i.e., in both program fragments the resulting value assigned to the program variable `a` is 4. All other mutants labeled with 1 to 3, 5 or 6 (in green color) are non-equivalent to the original program.

Secondly, each test case from the test bench is executed on the original program and on the mutants. Initially, all mutants are per-definition *stubborn*. A mutant M of the program P is said to be detected (or killed) by the test case t if t effectively distinguishes the mutant from



if(a > b + c){ a = 2 * 2; } ⓪
if(a > b - c){ a = 2 * 2; } ①
if(a > b * c){ a = 2 * 2; } ②
if(a > b / c){ a = 2 * 2; } ③
if(a > b + c){ a = 2 + 2; } ④
if(a > b + c){ a = 2 - 2; } ⑤
if(a > b + c){ a = 2 / 2; } ⑥

Figure 2.1: The effect of a mutation operator applied to a fragment of ANSI-C program code.

the original program, i.e., either the test case t fails on M and passes on P or the test case t passes on M and fails on P.

Thirdly, the number of detected mutants is used to quantify the fault-finding abilities of the test bench by means of the mutation score.

Definition 4 (Mutation Score)**.** *Suppose $P \in \mathcal{P}$ is a program and $\mathcal{N}$ is a set of mutants of P, the* mutation score *of a test bench T is the ratio of mutants detected by T to the number of mutants in $\mathcal{N}$ which are not functionally equivalent to P.*

The higher the mutation score, the more artificial faults are detected by the test bench. Recall that we call mutants which are functionally equivalent to P equivalent mutants. Determining the functional equivalence of two arbitrary programs of $\mathcal{N}$ is for most sets $\mathcal{N}$ impossible.

In order to overcome this problem in practice, the definition of the mutation score is weakened, i.e., the number of mutants in $\mathcal{N}$ not functionally equivalent to P is over-approximated by the total number of mutants in $\mathcal{N}$.

Definition 5 (Estimated Mutation Score)**.** *Suppose $P \in \mathcal{P}$ is a program and $\mathcal{N}$ is a set of mutants of P, the* estimated mutation score *of a test bench T is the ratio of mutants detected by T to the number of mutants in $\mathcal{N}$.*

As a result, the estimated mutation score under-approximates the "real" mutation score. The estimated mutation score, however, can be calculated in practice because the undecidable functional equivalence detection is avoided. As an optional step, equivalent mutant detection targets the correction of the estimated mutation score by identifying some equivalent mutants allowing a variety of trade-offs in the calculation. A detected equivalent mutant adds no information to mutation testing and is discarded from future analysis. Each detected mutant improves the approximation of the estimated mutation score. When all equivalent mutants are detected, the estimated and the "real" mutation score become equal.

2.3.2 Underlying Hypothesis

Mutation adequacy is based on two fundamental assumptions [DLS78], the *competent programmer hypothesis* and the *coupling effect hypothesis*. The competent programmer hypothesis expresses that developers typically write almost correct program code but they may introduce some minor mistakes. The coupling effect hypothesis states that complex failures are the result of several interacting simple faults, i.e., the simple faults are coupled together.

These two assumptions legitimate the restriction to consider first order faults: firstly, a competent programmer makes simple mistakes similar to first order faults. A test bench which is trained to detect seeded first order faults will, thus, detect real faults too. Secondly, complex faults are similar to coupled simple faults. A test bench which detects a significant number of simple faults will detect several complex faults too. The effects of several simple faults may mask each other. Fault masking, however, is rare [Off89]. Intuitively, adding faults to a faulty program will more likely result in a program which fails when executed than make the program correct.

The hypothesis have been confirmed empirically [Off92, ABL05] and analytically [Wah03] with a simple mathematical model which represents a program as a composition of finite functions.

2.3.3 Test Process

Figure 2.2 shows a test process which is driven by the quality estimation of a test bench. The process uses mutation adequacy as the quality measure. Thus, a high-quality test bench is good in the detection of seeded faults according to a fixed fault model.

The process is started with a program P and a test bench T and proceeds in the following steps: firstly, the test bench is executed on the program. If the test bench fails either the program or the test bench is faulty. The program and the test bench then needs to be analyzed to localize and fix the inconsistency between them. The test process is then restarted with the corrected versions of the program and the test bench. For the sake of simplicity, in Figure 2.2 we assume that the test bench is correct.

Secondly, the mutation adequacy of the test bench is calculated using the estimated mutation score. If the resulting mutation score is lower than a predefined threshold, the test bench is insufficient in detecting artificial faults. Then, the test bench is analyzed and improved. The test process is restarted with the improved test bench. Each time the process is restarted either the original program or the test bench is improved until eventually the mutation score of the test bench exceeds the predefined threshold. The resulting test bench passes on P and is partial mutation adequate with respect to the predefined threshold.

Mutation systems typically automate the execution of the test bench and the calculation of the estimated mutation score. However, the mutation systems do not support testers in analyzing and fixing the program or analyzing and improving the test bench. Moreover, mutation adequacy is calculated on basis of estimated mutation score, i.e., the mutation system assumes that each seeded fault is non-equivalent.

In this thesis, we attempt to improve the estimation of mutation adequacy, i.e., we will present two approaches: one to detect some equivalent mutants and one to detect non-equivalent mutants. The two approaches can be used to tighten the approximation of the estimated mutation score.

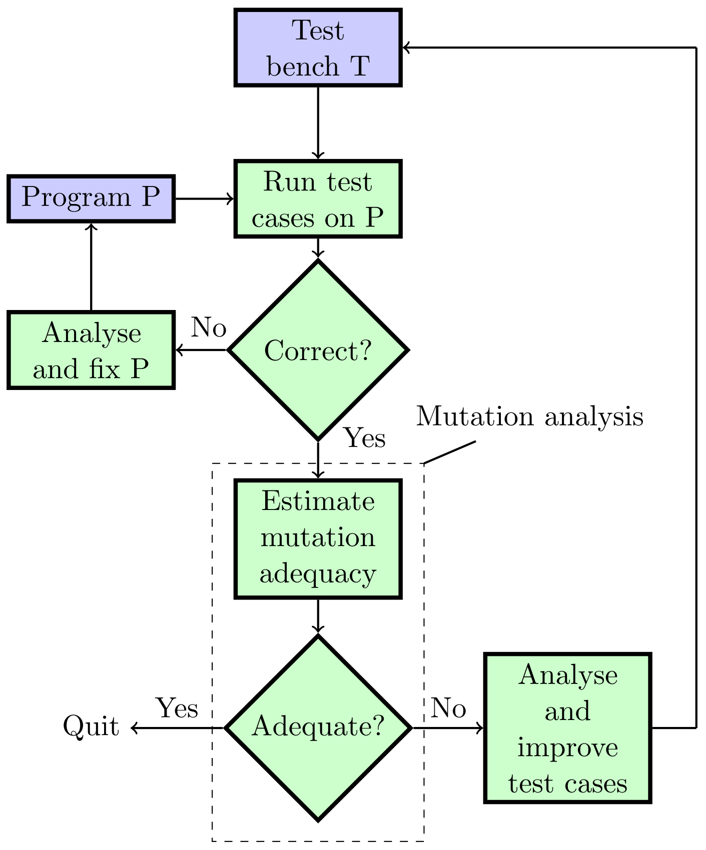


Figure 2.2: Test processes driven by the mutation score to generate a high-quality test bench. Blue boxes indicate process inputs. Green boxes refer to generic actions performed on process inputs.

2.3.4 Mutation Operators

A mutant is a duplication of the original program source containing one syntactic change. For instance, an addition operator is turned into a subtraction operator. A *fault model* defines a set of mutation operators, that are, rules describing possible syntactic changes when applied to the source code of a program. Fault models are defined with respect to a particular programming language.

The set of mutation operators has a large influence on the mutation score and the underlying quality indication of mutation analysis and mutation testing. Different mutation operators have been design for a variety of descriptive formalism for specifying and implementing a computing task including database schemata, specification languages, and programming languages. We focus on program-based mutation testing.

Research in program-based mutation testing mostly targets fault seeding at the unit-level [KO91]. A particular software unit is seeded with artificial faults and tested. Delamaro et al. [DMM96] and Ghosh and Mature [GM01] introduced *interface mutation* as a technique for scalable fault-based integration testing. In interface mutation, faults are seeded into interfaces which connect two or more units, i.e., the mutation operators introduce syntactic changes into method and function calls used to communicate between different unit. Mateo et al. [MUO10] applied mutation testing to an entire program as a system-level testing method. Their mutation operators, however, are a mixture of mutation operators proposed for unit testing [KO91] and integration testing [DMM96].

In the following, we focus on mutation operators proposed for unit testing. King and Offutt [KO91] originally designed 22 mutation operator for Fortran. They argued that these mutation operators can be used for any imperative programming language. The mutation operators were chosen such that test cases generated in mutation testing likely catch typical programming mistakes and subsume test cases generated by other test criteria (e.g., statement or branch coverage). For instance, a mutation operator generates two mutants for each expression in the program, where the expression is increased by 1 in the first mutant and decreased by 1 in the other mutant. This is an example of a mutation operator which mimics programming mistakes, namely off-by-one faults. Another mutation operator generates two mutants for each condition in the program, where the condition is replaced by true in the first mutant and replaced by false in the other mutant. This mutation operator generates mutants such that a test bench which detects these mutants subsumes branch coverage.

Offutt et al. [OLR+96] reduced the 22 mutation operators to five key mutation operators. The reduction technique is referred to as *constraint mutation* and the five key mutation operators are called *expression-selective mutation operators*. Offutt et al. [OLR+96] showed by experimentation on 10 small example programs that a test bench which detects all mutants generated from the expression-selective mutation operators, detects 99.5% of the mutants generated from all 22 mutation operators.

Agrawal et al. [ADH+06] provided a comprehensive set of 77 mutation operators for ANSI-C. The mutation operators are designed to cover all features of the syntax of the ANSI-C programming language. Research tools often consider a subset of those mutation operators because implementing all 77 mutation operators is expensive and cumbersome. Usually, the mutation operators corresponding to the expressive-selective mutation operators are considered.

A more recent trend when considering more expressive languages like Java, C#, or C++ is to design mutation operators for the Compiler's intermediate language rather than for the more expressive source language. Gruen et al. [GSZ09] applied mutation operators to Java bytecode. Dereziska and Kowalski [DK11] applied mutation operators to the .NET Common Intermediate Language (CIL) to cope with C#, whereas Riener et al. [RBF11] provided mutation operators for Low Level Virtual Machine (LLVM). This approach, however, changes the granularity of a fault, i.e., due to the expressiveness of languages like Java, C#, or C++ a first order fault in a program at the level of the original source language may correspond to a complex fault in the translated program at the intermediate level. Moreover, a mutation at the intermediate level may not correspond to a syntactic valid mutation at the level of the source language.

Moreover, the implementation of mutation operators in different research tools vary. King and Offutt [KO91] found that trivial equivalent mutants can be determined when applying mutation operators to a program. For instance, a variable with value 0 will always result in an equivalent mutant when replaced by the constant 0. As a consequence, a powerful static analysis can avoid the creation of several equivalent mutants. King and Offutt [KO91] and Hu et al. [HLO11] target the avoidance of the generation of equivalent mutants. They provided *equivalence conditions* for mutation operators for imperative and object-oriented programming languages, that are, conditions which when satisfied result in the generation of an equivalent mutant.

The variety in the set of mutation operators used in research (and in their implementation) makes the comparison of experimental results complicated. Mutation scores depend on the set of mutation operators and their specific implementation. Details on the mutation operators and their implementation is rarely available in publications.

In this thesis, we apply mutation operators to the intermediate representation LLVM similar to Riener et al. [RBF11]. We discuss the fault model, the mutation operators, and outline the relation between ANSI-C and LLVM in Section 2.5.5.

2.3.5 Meta-Mutant

Early mutation testing systems interpreted the program under test and its mutants. Since building an interpreter for programming language is complicated and interpreting programs is slow, one improvement over the interpretative approach is to compile each mutant to an executable. The number of mutants, however, is typically very high and the compilation then becomes the bottleneck of mutation testing. Untch et. al. [UOH93] proposed the idea of a *meta-mutant* (or *mutant schemata*), i.e., a single program that encodes a set of mutants and additional control logic for testing purpose into one source-level program. The additional control logic provides a special input variable, e.g., a global variable `FAULT_ID` which is used to enable and disable individual faults. The source-level meta-mutant is then compiled only once and the behavior of one of the encoded mutants is selected at run-time by assigning a value to the global variable `FAULT_ID`.

For instance, consider the program fragment in Figure 2.3 that encodes the mutants from Figure 2.1 into one program. We assign an id to each mutant by consecutively numbering the mutants in Figure 2.1 from left to right. The id `0` is reserved for the original program. The value of the variable `FAULT_ID` is controlled by the tester denoted by the pseudo-code function `input()` and either enables the behavior of one of the encoded mutants (if the value is between 1 and 6) or disables all mutants (if the value is 0 or greater than 6).

```
unsigned long FAULT_ID = input();
[ ... ]
if (FAULT_ID == 1 ? a > b - c :
      (FAULT_ID == 2 ? a > b * c :
        (FAULT_ID == 3 ? a > b / c :
          a > b + c))){
  a = (FAULT_ID == 4 ? 2 + 2 :
         (FAULT_ID == 5 ? 2 - 2 :
            (FAULT_ID == 6 ? 2 / 2 :
              2 * 2)));
}
```

Figure 2.3: Meta-mutant encoding the mutants from Figure 2.1.

2.4 Bounded Model Checking

In this section, we describe SAT-based Bounded Model Checking (BMC). BMC was originally introduced by Biere et al. [BCCZ99] as a symbolic model checking approach complementary to BDD-based symbolic model checking [BL92]. It searches for counterexamples of finite length in a given *state-transition system* which serves as a model of the system under design. If none exist the system is proven correct. The focus of BMC lies typically on "hunting bugs", i.e., falsifying properties of a formal specification rather than proving correctness of a state-transition systems with paths of finite length. More recently, BMC was applied to software programs [CKOS04, ISGG05]. We borrow from Kroening [BHvMW09] in terms of terminology and the conceptual framework to introduce BMC, where BMC is described to verify sequential software programs.

Definition 6 (State-Transition System)**.** *A* state-transition system *is a triple (S, S_0, R), where S is a set of states, $S_0 \subseteq S$ is a set of initial states, and $R \subseteq S \times S$ is a transition relation.*

A *state* $s \in S$ consists of a valuation of the *program counter*, a valuation of the (global and local) *program variables*, and a valuation of a *function call stack*. The program counter is a special variable pointing to the program location which is executed next. The function $pc : S \rightarrow \mathcal{L}$ maps a state to its program location, where $\mathcal{L}$ is the set of all program locations. We denote the program's entry point — typically the first instruction of the main function of a program — by the distinguished program location $l_{entry} \in \mathcal{L}$. The program variables form a finite set V over the domain D. The valuation of the program variables is partial, i.e., not all program variables are defined in each state. We use the symbol $\perp$ to denote an undefined value. The function call stack is an unbounded stack which is used to handle function calls. Thus, the elements of the function call stack are either valuations of the program counter or valuations of local variables.

The *initial states* $s \in S_0$ represent all valuations of the program variables, where the function call stack is empty and the program counter points to the label l_{entry}. The *transition relation* $R(s, s')$ encodes the program semantics, i.e., $R(s, s')$ holds iff the state s' is the successor of the state s when the instruction at location $pc(s)$ is executed.

The state-transition system serves as a model of a program which is checked for correctness with respect to a formal specification. Typically, the formal specification is a *reachability property*, i.e., it describes a set of bad states in the state-transition system that should not be reached if the implementation conforms to the specification. We use a set of error locations $\mathcal{L}_E \subseteq \mathcal{L}$ to denote the program locations of the bad states described by the formal specification.

Definition 7 (Counterexample)**.** *A* counterexample *is a sequence of states $s_0, s_1, \ldots, s_n$ which ends in a bad state, where $s_0 \in S_0$, $s_j \in S$ for $0 \leq j \leq n$, $R(s_i, s_{i+1})$ holds for $0 \leq i \leq n-1$, and $pc(s_n) \in \mathcal{L}_E$.*

Given a natural number k called *unrolling bound*. BMC searches for counterexample of finite length. Due to input-dependent loops, the length of a path in a given state-transition system may be infinite. Thus, BMC uses an unfolding technique to transform the state-transition system into a model with finite paths. A straight forward unfolding approach is to replicate the transition relation k-times. The size of the resulting state-transition system is k-times the original state-transition system. A more space-efficient approach is *loop unrolling*, i.e., a restricted state-transition system is constructed, where the loop body is $(k+1)$-times replicated, the true-edge of the loop condition in the i-th replicate is directed to $(i+1)$-th replicate for $0 \leq i \leq k$, and the false-edge of the loop condition in the i-th replicate is directed to the next instruction to be executed after the loop. The source locations of the $(k+1)$-th replicate of the loop body are marked as bad state.

In SAT-based BMC, both the state-transition system and the formal specification are transformed into a conjoined logic formula $\varphi = \varphi_P \wedge \varphi_S$ in a formal logic, e.g., propositional logic. The first part φ_P of the logic formula encodes the paths of the state-transition system. The second part φ_S of the logic formula encodes the formal specification which describes a set of bad states that should not be reached.

The logic formula is then handed to a theory solver with respect to the logic theory in use which checks whether the formula is satisfiable. A satisfying assignment obtained from the theory solver corresponds to an assignment to all input and program variables to force an execution along a particular path which reaches a bad state. From the assignment a counterexample (or test case) can be extracted which leads to the violation of the formal specification. If no satisfying assignment exists the formal specification holds.

2.5 LLVM Intermediate Language

2.5.1 Overview of LLVM

Low Level Virtual Machine (LLVM) is a strongly typed, RISC-like assembly language and provides a virtual instruction set which abstracts from machine-specific details such as the number of registers, the organization of data in the memory, the instruction set provided by the target architecture, etc.

The main characteristics of LLVM are as follows: (1) LLVM provides an infinite number of (virtual) *registers*. The registers can be used to store data of a particular data type. LLVM is a strongly typed language, i.e., it is only allowed to store data in a register if their data types conform. (2) The registers exhibit the *single-assignment property* [AWZ88], i.e., each register is assigned only once. (3) The transfer between the registers and the memory is accomplished by `load` and `store` instruction.

LLVM is a complete programming language, however, it was designed as an intermediate representation for the *LLVM compiler framework*. The LLVM compiler framework provides a set of compilers for a variety of programming languages and target architectures. Similar to the GNU compiler collection, the LLVM compiler framework is organized in *compiler front-ends* and *compiler back-ends*. Each compiler front-end translates programs from a specific programming language to LLVM programs. Each compiler back-end translates LLVM programs to code for a specific target architecture. The concept of compiler front-ends and compiler back-ends reduces the combinatorial explosion when n programming language are allowed to be compiled to m target architectures. Instead of $n \cdot m$ compilers only n compiler front-ends and m compiler back-ends are needed. LLVM serves as a unique, intermediate representation in between the compiler front-ends and the compiler back-ends. We refer to this stage as the *compiler middle-end*.

LLVM comes with a rich tool support which allows for disassembling (`llvm-dis`), debugging (`lldb`), interpreting (`lli`), and optimizing (`opt`) of LLVM programs.

2.5.2 Structure of an LLVM program

An LLVM program is built from instructions. The instructions are organized in *basic blocks*. Each basic block is a maximal sequence of consecutive instructions with a unique label, where control flow enters the basic block at the first instruction and leaves the basic block at the last instruction. The last instruction of a basic block is a *terminator instruction* which defines the transfer of the control flow to one or more successor basic blocks.

The basic blocks are organized in *functions*. The *Control Flow Graph* (CFG) of a function is a (directed) graph. The nodes of the CFG denote the basic blocks of the function with two

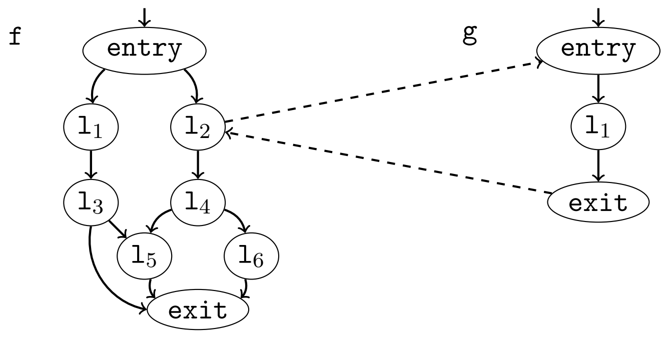


Figure 2.4: The control flow graphs of a program consisting of two function `f` and `g`. Solid edges denote control edges and dashed edges denote call edges.

additional basic blocks `entry` and `exit`. The edges of the CFG represent function calls and the control flow within the function. There is a *control edge* from basic block X to basic block Y if control transfers from X to Y defined by the terminator instruction of X. Additionally, there is a control edge from the basic block `entry` of a function to each basic block at which the function can be entered and there is a control edge to the basic block `exit` from each basic block the function is left.

The *call edges* are defined as follows. Consider a function call, where function g is called from an instruction in basic block X in function f. There is a call edge from basic block X to the basic block `entry` of function g and a call edge from the basic block `exit` of function g to basic block X.

Figure 2.4 shows the control flow graphs of a program consisting of two functions `f` and `g`. The labels on the nodes of each graph are unique. Solid edges denote control edges and dashed edges denote call edges. The dashed edge from basic block $\mathtt{l}_2$ of function `f` to basic block `entry` of function `g` denotes a function call and the dashed edge from basic block `exit` of function `g` to basic block $\mathtt{l}_2$ of function `f` denotes the return to the caller-site, i.e., basic block $\mathtt{l}_2$ in function `f`.

2.5.3 LLVM Instruction Set

In the following, we define the syntax and semantics of a simplified, subset of the LLVM instruction set. LLVM is a strongly typed language, i.e., each register has a particular type and each assignment and value is preceded by a type specifier which explicitly names the type of the values and registers used. For the sake of simplicity, we omit the type specifiers in our description. Additionally, we do not discuss function and parameter attributes, linkage types, calling conventions and LLVM's intrinsic functions because these features are only useful in very specific situations. For a detailed description of those language features, we refer to the *LLVM Language Reference Manual* [LA10].

LLVM instructions are built from *values*. A value denotes either denotes a constant or the address of a register. We use Value to denote the set of all values. Suppose r_{dest} denotes a register, v, v_{op_1}, v_{op_2}, ..., v_{op_n} denote values, *addr* denotes the address of a register, *ty* denotes a type, and l, l_{true}, l_{false} denote labels. We define the syntax and semantics of the subset of LLVM instructions used in the following sections.

Memory Access and Allocation

The unlimited number of strongly-typed registers in LLVM can be interpreted as program variables. The LLVM language, however, additionally uses a load-store architecture to access the main memory. In the following, we define the syntax and semantics of the instructions used to dynamically allocate, read, and write memory.

- The alloca instruction is of the form

  $$r_{dest}\ \texttt{= alloca}\ ty\texttt{,}\ v.$$

  The instruction allocates memory on the stack frame of the currently executed function. The register r_{dest} is a new register. The type ty specifies the type of the data stored in the register and the value v specifies the number of elements stored in the register. After the alloca instruction was executed, the register r_{dest} can be used like any other virtual register. The allocated memory is automatically released when the function is left.

- The load instruction is of the form

  $$r_{dest}\ \texttt{= load}\ addr.$$

  The register r_{dest} is assigned the value at memory address *addr* when the load instruction is executed.

- The store instruction is of the form

  $$\texttt{store}\ v\texttt{,}\ addr.$$

  The value at memory address *addr* is assigned the value v when the store instruction is executed.

Control Flow Manipulation

Terminator instructions are used to manipulate the control flow of a function. The last instruction of each basic block is a terminator instruction. The terminator instruction defines the successors of the basic block, i.e., the set of basic blocks from which one is executed when the current basic block is finished. Most frequently the terminator instruction is a branching instruction.

- The branching instruction is either of conditional

  $$\texttt{br}\ v\texttt{, label}\ l_{true}\texttt{, label}\ l_{false}$$

  or unconditional form

  $$\texttt{br label}\ l.$$

  The conditional branching instruction sets the value of the program counter to the address of l_{true} when v equals 1 and otherwise to the address of l_{false}. The unconditional branching instruction sets the program counter to the address of label l when executed.

Additionally, the switch instruction is a terminator instruction. For the sake of simplicity, we do not discuss switch instructions but assume they have been replaced by a sequence of branching instructions with equal semantics.

Binary Operators and Comparison

- The binary operator instruction is of the form

$$r_{dest}\ \texttt{= binop}\ v_{op_1}\texttt{,}\ v_{op_2},$$

  where `binop` is one of the mnemonics **add**, **sub**, **mul**, **div**, **rem**, **shl**, **shr**, **and**, **or**, or **xor**. The binary operator instruction assigns the register r_{dest} the value of the function $f_{binop}(v_{op_1}, v_{op_2})$ when executed, where f_{binop} is a binary function representing the semantics of the binary operation denoted by the mnemonic, respectively.

  We define

$$f_{binop}(x,y) = \begin{cases} x+y, & \text{binop} = \textbf{add} \\ x-y, & \text{binop} = \textbf{sub} \\ x/y, & \text{binop} = \textbf{div} \\ x\%y, & \text{binop} = \textbf{rem} \\ x << y, & \text{binop} = \textbf{shl} \\ x >> y, & \text{binop} = \textbf{shr} \\ x\&y, & \text{binop} = \textbf{and} \\ x|y, & \text{binop} = \textbf{or} \\ x \wedge y, & \text{binop} = \textbf{xor} \end{cases}$$

  where the assigned expression should be interpreted as ANSI-C expressions.

- The comparison instruction is of the form

$$r_{dest}\ \texttt{=}\ \textbf{\texttt{icmp}}\ \texttt{cond}\ v_{op_1}\texttt{,}\ v_{op_2},$$

  where `cond` is one of the mnemonics **eq**, **ne**, **gt**, **ge**, **lt**, or **le**. The comparison instruction assigns the register r_{dest} the value of the function $f_{cond}(v_{op_1}, v_{op_2})$ when executed, where f_{cond} is a binary function representing the semantics of the comparison instruction denoted by the mnemonic, respectively.

  We define

$$f_{cond}(x,y) = \begin{cases} x == y, & \text{cond} = \textbf{eq} \\ x! = y, & \text{cond} = \textbf{ne} \\ x > y, & \text{cond} = \textbf{gt} \\ x >= y, & \text{cond} = \textbf{ge} \\ x < y, & \text{cond} = \textbf{lt} \\ x <= y, & \text{cond} = \textbf{le} \end{cases}$$

  where the assigned condition should be interpreted as ANSI-C conditions.

Function Calls

- The call instruction is of the form

$$r_{dest}\ \texttt{=}\ \textbf{\texttt{call}}\ \texttt{f([}v_{op_1}\texttt{,}\ v_{op_2}\texttt{,}\ \texttt{...,}\ v_{op_n}\texttt{]).}$$

  The call instruction assigns the register r_{dest} the return value of the function `f` executed on the sequence of arguments v_{op_1}, v_{op_2}, $\ldots$, v_{op_n}. The brackets `[, ]` denote that the sequence of arguments may be optional and depends on the declaration of the function `f`, respectively.

- The ret instruction is of the form

  $$\texttt{ret}\ v.$$

  Assume the ret instruction is defined in a function `f` and function `f` is called by a function `g`. The instruction then returns the value v from function `f` to the caller side, function `g`, when executed.

Call and ret instructions define the call edges of the CFG of functions.

Static Single-Assignment Form and Phi Nodes

The (virtual) registers of LLVM exhibit the single-assignment property, where each register is assigned only once. Programs with this property are said to be in *static single-assignment* (SSA) form [AWZ88].

We say a register (or value) is *defined* if it is assigned by an instruction and a register (or value) is *referenced* if it is used in an instruction. Usually, all registers and values on the left-hand side of an instruction are defined and all registers and values on the right-hand side of an instruction are referenced. However, one exception is the store instruction, where the left operand is referenced and the right operand is defined. Additionally, we call the instruction that defines a register (or value) a *definition* of the register (or value) and the instruction that references an instruction a *use* of the register (or value). A program is in SSA-form if each register is defined at most once. SSA-form is desirable because it simplifies the design of algorithms for analyzing and manipulating LLVM programs.

The virtual registers of LLVM are always in SSA-form but registers allocated with the alloca instruction are generally not in SSA-form, i.e., they are defined more than once. However, LLVM provides standard mechanisms to transform memory accesses into SSA-form, called memory-to-register promotion. Memory-to-register promotion is easy in case of a straight-line program without conditional branching instructions. Then, registers that are used more than once are replicated and renamed such that each replicate has a unique name. In case of programs where multiple definitions from different branches reach a specific use of a register, phi instructions are introduced to merge the values of the definition from the different branches. Suppose k definitions of a register reach a particular use of the register, the registers are renamed such that each definition has is unique. Finally, a phi instruction is added to merge the values of the definitions. The phi instruction chooses one value from the all definitions dependent on the branch which was executed before. An efficient algorithm for the transformation of a program into SSA-form was given by Cytron et al. [CFRW91].

- The phi instructions enable the transformation of LLVM programs into SSA form because the instructions merge data from different branches. A phi instruction is of the form

  $$r_{dest}\ \texttt{=}\ \texttt{phi}\ \texttt{[}v_{op_1}\texttt{,}l_1\texttt{]}\texttt{,}\texttt{[}v_{op_2}\texttt{,}l_2\texttt{]}\texttt{,}\ldots\texttt{,}\texttt{[}v_{op_n}\texttt{,}l_n\texttt{]}.$$

  We say that the current basic block is the basic block containing the phi instruction. The basic blocks with labels l_1, l_2, ..., l_n are direct predecessors of the current basic block. The phi instruction assigns the register r_{dest} the value v_{op_i} if the basic block labeled with l_i, $1 \leq i \leq n$, was immediately executed before the current basic block.

2.5.4 LLVM Metadata

The LLVM language[1] provides support for *metadata*, i.e., arbitrary values that can be attached to individual LLVM instructions. Metadata is a mechanism to communicate specific information from a front-end to the optimizer or debugger. For instance, a compiler front-end may tag variables that are guaranteed not to alias which is exploited in optimizations. Another example is the usage of metadata to encode source locations into the LLVM intermediate representation which can be used during debugging.

In particular, metadata is an umbrella term for *metadata nodes* and *named metadata nodes*. Suppose v_n, v_1, v_2, and v_k denote values, and *str* is an identifier, a metadata node is a structure-like value of the form

$$\texttt{!}v_n\ \texttt{=}\ \textbf{\texttt{metadata}}\ \texttt{!}\{v_1\}.$$

Notice that a metadata node v_n itself is a value such that metadata nodes can be recursively nested. The named metadata node represents a collection of metadata nodes. It is of the form

$$\texttt{!}\mathit{str}\ \texttt{=}\ \textbf{\texttt{metadata}}\ \texttt{!}\{v_1\texttt{,}\ v_2\texttt{,}\ \texttt{...,}\ v_k\}.$$

The identifier *str* gives the named metadata node a unique, global name which can be used to look up a collection of metadata nodes by the unique name assigned to the corresponding node.

Individual metadata nodes can be attached to LLVM instructions which is denoted by the pattern

$$\mathit{instruction}\texttt{,}\ \textbf{\texttt{metadata}}\ \texttt{!}v_n,$$

where *instruction* represents an arbitrary LLVM instruction and v_n is a metadata node. We will use metadata nodes and named metadata nodes to attach information about seeded faults to an LLVM program.

2.5.5 A Simple Fault Model for LLVM

For mutation, we use a fault model similar to the fault models proposed in [OLR+96] and [GSZ09]: we introduce syntactic changes into arithmetic, relational, and bitwise operators, and inject values into expressions used in load instructions. Our decision procedures for the detection of equivalent and non-equivalent mutants, however, are not tied on this fault model.

We formalize the fault model using mutation operators applied to the level of the LLVM intermediate representation. The fault model considers four mutation operators: the replacement of arithmetic and bitwise binary operators, the replacement of comparison instructions, and the manipulation of values used in load instructions. Each mutation operator is denoted by a three-letter acronym. We use the acronyms `AOR`, `BOR`, `RCI`, and `IVI` — a detailed description follows below.

The `RCI` mutation operator, for instance, replaces each occurrence of a mnemonic of a comparison instruction by another valid mnemonic, e.g., the mutation operators replaces the mnemonic for lower than (**`lt`**) against the mnemonics for equality (**`eq`**), inequality (**`ne`**), greater

[1]The metadata feature was incorporated into LLVM version 2.7.

than (**gt**), lower than (**le**), and greater equal (**ge**), resulting in five mutants of the source program.

Next, we describe the technical details of the mutation operators AOR, BOR, RCI, and IVI. Suppose the source code of a program is a sequence of strings separated by white-spaces. We call these strings *tokens* and denote the set of all tokens by Tokens. We use $\mathrm{Occ}(T, P)$ to denote the set of occurrences of the tokens $T \subseteq \mathsf{Tokens}$ in the source code of program P, where T is a set of tokens which are searched and replaced by a mutation operator. The replacement of the token t by the token t' in program P is denoted by the expression $P[t/t']$.

- **Replacement of Arithmetic Operators (AOR)**: The AOR mutation operator is a mapping

$$\begin{aligned} \mathsf{t}_{\mathsf{AOR}} :& \mathcal{P} \to 2^{\mathcal{P}}, \\ & P \mapsto \{P[t/t'] \mid t \in \mathrm{Occ}(\mathsf{AOp}, P), t' \in \mathsf{AOp} \setminus t\} \end{aligned}$$

  which mimics a mistake in an arithmetic binary operator, where $\mathsf{AOp} := \{$`add`, `sub`, `mul`, `div`, `rem`$\}$.

- **Replacement of Bitwise Operators (BOR)**: The BOR mutation operator is a mapping

$$\begin{aligned} \mathsf{t}_{\mathsf{BOR}} :& \mathcal{P} \to 2^{\mathcal{P}}, \\ & P \mapsto \{P[t/t'] \mid t \in \mathrm{Occ}(\mathsf{BOp}, P), t' \in \mathsf{BOp} \setminus t\} \end{aligned}$$

  which mimics a mistake in a bitwise binary operator, where $\mathsf{BOp} := \{$`and`, `or`, `xor`, `shl`, `shr` $\}$.

- **Replacement of Comparison Instructions (RCI)**: The RCI mutation operator is a mapping

$$\begin{aligned} \mathsf{t}_{\mathsf{RCI}} :& \mathcal{P} \to 2^{\mathcal{P}}, \\ & P \mapsto \{P[t/t'] \mid t \in \mathrm{Occ}(\mathsf{ROp}, P), t' \in \mathsf{ROp} \setminus t\} \end{aligned}$$

  which mimics a mistake in a relational operator, where $\mathsf{ROp} := \{$`eq`, `ne`, `gt`, `ge`, `lt`, `le`$\}$.

- **Integral Value Injection (IVI)**: The IVI mutation operator is a mapping

$$\begin{aligned} \mathsf{t}_{\mathsf{IVI}} :& \mathcal{P} \to 2^{\mathcal{P}}, \\ & P \mapsto \{P[t/t+1], P[t/t-1], P[t/0] \mid t \in \mathrm{Occ}(\mathsf{Value})\} \end{aligned}$$

  which mimics off-by-one faults and the injection of zero values. In order to encode the off-by-one faults for values representing register addresses, the syntax of LLVM requires the insertion of additional add and sub binary operator instructions.

The four mutation operators used are similar to Offutt's set of *expression-selective mutation operators* [OLR+96, KO91] for Fortran and the mutation operators used in [GSZ09]. Some of the operators in [GSZ09] are only marginally described which prevents us from making an in-depth comparison. We took the AOR and RCI mutation operators from [KO91] and adapted them for the LLVM instruction set. Moreover, we supplemented the BOR mutation operator because in [KO91] no bitwise operators for Fortran are described. The IVI mutation operator is similar to Offutt's UOI mutation operator and covers all mutations from the category "replace numerical constants" in [GSZ09].

```
l1:   r1 = load i1
      r2 = icmp lt i2, i1
      br r2 label l2, label l3
l2:   r1 = load i2
      br label l3
l3:   o1 = load r1
```

Figure 2.5: An example program conforming to the subset of LLVM described in Section 2.5.3, that calculates the minimum of two given variables `i1` and `i2` and saves the resulting output in the variable `o1`.

2.5.6 Meta-Mutant Construction

Given a program P and a list of faults to be seeded into the program according to a fixed fault model, first, each fault gets a unique id, e.g., by consecutively numbering the faults. We start numbering at 1 and reserve the id 0 for the original program behavior. The faults are systematically seeded into the program. For each fault, the basic block that is seeded with the fault is duplicated and then mutated. Thus, the meta-mutant contains the original and the new, mutated basic block.

We add a global variable `FAULT_ID` and additional control logic per fault to the program. The control logic enables one mutated basic block at a time if the value of `FAULT_ID` is equal to the id of the fault and the original basic block otherwise.

Figure 2.5 gives a short example program that conforms to the subset of LLVM described in Section 2.5.3. The example program calculates the minimum of two given program inputs and returns the result as program output. We denote the program inputs by the input registers `i1`, `i2`, and the program output by the output register `o1`. For the sake of simplicity, the example program is not in SSA form (the register `r1` is assigned twice).

Suppose a potential fault in the basic block `l1` with id 1, where the mnemonic of the comparison instruction **`lt`** (lower than) is turned into the mnemonic **`le`** (lower equal). In order to construct the meta-mutant, the basic block `l1` is duplicated and mutated. The meta-mutant is shown in Figure 2.6. The basic block `mut1` corresponds to the faulty duplicate of basic block `l1` and the basic block `chk` serves as control logic, which, based on the value of the variable `FAULT_ID`, either enables the mutant in basic block `mut1` or the original program behavior in basic block `l1`.

We use metadata nodes to mark a seeded fault in the program with its fault id. In our meta-mutant construction, a seeded fault corresponds to a basic block. However, LLVM offers no mechanism to add metadata to a basic block. Thus, we add a metadata node to the terminator instruction of the basic block instead. In the example, in Figure 2.6, the terminator instruction of the basic block `mut1` is marked with the metadata node !1. The metadata node !1 shown at the bottom of Figure 2.6 contains a single value 1 corresponding to the fault id of the seeded fault.

Figure 2.6 contains only a single mutant, however, the meta-mutant construction can be extended to an arbitrary number of faults by adding more basic blocks, where each basic block contains one fault (similar to the basic block `mut1`) and control logic (similar to the basic block `chk`). However, we use a more subtle construction using the switch instruction contained in the full LLVM instruction set. The size of the resulting meta-mutant is linear in the size of the original program.

```
chk:  r1 = load FAULT_ID
      r2 = icmp eq r1, 1
      br r2 label mut1, label l1
mut1: r3 = load i1
      r4 = icmp le i1, i2
      br r4, label l2, label l3, metadata !1
l1:   r3 = load i1
      r5 = icmp lt i1, i2
      br r5 label l2, label l3
l2:   r3 = load i2
      br label l3
l3:   o1 = load r3

!1 = metadata !{1}
```

Figure 2.6: A meta-mutant of the example program given in Figure 2.5 with a single fault: the mnemonic of the comparison instruction **lt** in basic block **l1** is replace with mnemonic **le**.

2.6 Summary

In this chapter, we have introduced mutation testing, bounded model checking, and the LLVM intermediate representation. We have discussed a subset of the LLVM instruction set and have shown a simple fault model for LLVM using four mutation operators. The mutation operators seed faults into arithmetic, relational, and bitwise operators, and inject values into load instructions. Our fault model, however, is not explicitly tied on these mutation operators.

Moreover, we have presented the concept of a meta-mutant, i.e., a single program containing a set of mutants and additional control logic to enable and disable the individual behavior corresponding to the mutants. We have shown how to encode the meta-mutant into a single LLVM program. The meta-mutant construction uses LLVM metadata to mark the locations of the seeded faults in the program with their fault ids.

The consideration of mutation testing for LLVM programs and the construction of the meta-mutant is new. We use the discussed subset of LLVM and the meta-mutant construction in the remainder of the thesis and present approaches for the identification of equivalent and non-equivalent mutants.

Chapter 3

Detecting Equivalence

3.1 Introduction

In the following chapter, we describe an approach to detect equivalent (Section 3.3) and redundant mutants (Section 3.4) using static analysis methods leveraging optimizing source-to-source transformations.

Recall that equivalent mutants correspond to programs which are functionally equivalent to the original program and redundant mutants are functionally equivalent to another mutant. Equivalent mutants cannot be detected, whereas the afford to detect more than one of a set of redundant mutants is wasted.

We motivate our approach showing how the *GNU Compiler Collection (GCC)* and the Unix tool `diff` can be used to detect equivalent mutants (Section 3.2). Then, we present the Optimizer procedure (Section 3.3.1) that attempts to prove functional equivalence of a given program and one of its mutants using a sequence of optimizing source-to-source transformations. We give a brief description of some optimizing source-to-source transformations (Section 3.3.2) provided by the LLVM compiler infrastructure and present a standard optimization pipeline based on these transformations (Section 3.3.3). Afterwards, we present the MetaOptimizer procedure (Section 3.4.1) that generalizes Optimizer and, thus, detects redundant and equivalent mutants in a meta-mutant using a similar approach. Lastly, we summarize the chapter (Section 3.5).

3.2 Early Attempts and Motivation

In this section, we present a simple approach to detect functional equivalence of similar programs. This approach serves as motivation for the equivalence and redundancy detection procedures described in later parts of the chapter.

Consider two implementation of the same function `min` shown in Figure 3.1. The functions calculate the minimum of two given integer arguments denoted by the variables a and b: first, both functions arbitrarily assume that the value of b is the smallest input argument and initialize the value of a local variable m with the value of b. Then, the functions overwrite the value of the local variable m with the value of a if the value of a is actually smaller than the value of b. The two implementations differ only in the condition that checks whether the value of a is smaller than the value of b. The functions `min` shown in Figure 3.1 on the left and the right side use the conditions `a < b` and `a <= b`, respectively. However, both implementations are

```
int min(int a, int b){              int min(int a, int b){
  int m = b;                          int m = b;
  if (a < b) {                        if (a <= b) {
    m = a;                              m = a;
  }                                   }
  return m;                           return m;
}                                   }
```

Figure 3.1: Two implementation of a function `min` that calculates the value of two given inputs arguments `a` and `b`.

```
$ gcc -c -O1 org/min.c
$ gcc -c -O1 mut/min.c
$ diff -s org/min.o mut/min.o
Files org/min.o and mut/min.o are identical
```

Figure 3.2: Unix commands to decide whether the function `min` in Figure 3.1 are functionally equivalent using the GNU compiler collection (GCC) and `diff`.

functionally correct, i.e., the return value of the function `min` is in both cases $min\{a, b\}$. The two implementation are functionally equivalent because in case of equal input values $a = b$, it does not matter whether the function returns a or b.

We motivate our approach by using the GNU compiler collection (GCC) and the standard Unix tool *diff* to decide that the two `min` functions are functionally equivalent: let us assume that the function in Figure 3.1 on the left side corresponds to the original program and the function on the ride side is a first order mutant of the original program. We save both programs in separate source files, both called `min.c` but locate them in different directories `org/` and `mut/`, respectively. First, we use GCC to optimize the original program and the mutant. We specify the optimization level 1 with the flag `-O1` and enforce the compiler with the flag `-c` to not link the programs because they have no entry functions. GCC compiles and optimizes the source code of the two programs saving the results as object files `org/min.o` and `mut/min.o`. Finally, we compare the object files with the `diff` tool and request `diff` to output a message if the object files do not differ (with the flag `-s`). When the `diff` tool reports “identical files”, we conclude that the two given programs are functionally equivalent. Otherwise, we do not know whether the programs are actually different or GCC is not able to optimize them sufficiently.

Figure 3.2 lists the exact sequence of Unix commands executed. We choose the same name for both source files because GCC encodes the file name into the object file (but not the directory name). Thus, two functionally equivalent programs with different file names will always be reported as different.

3.2.1 Design Considerations

For the trivial example in Figure 3.1, GCC and `diff` are sufficient to prove that the mutant is functionally equivalent to the original program. In the following, we call this approach *the simple approach*. On basis of the simple approach, we give some design considerations for an equivalence detection procedure which leverages a standard compiler.

1. *Completeness*: The equivalent mutant problem is undecidable and thus the simple ap-

```
8b 45 08 mov 0x8(%ebp),%eax              8b 55 08 mov 0x8(%ebp),%edx
8b 55 0c mov 0xc(%ebp),%edx              8b 45 0c mov 0xc(%ebp),%eax
39 c2    cmp %eax,%edx                   39 d0    cmp %edx,%eax
0f 4e c2 cmovle %edx,%eax                0f 4d c2 cmovge %eax,%edx
```

Figure 3.3: Two fragments of x86 assembly source code: the usage of the registers `eax` and `edx` is exchanged in the left and right code fragments. Thus, the two source fragments are functionally equivalent.

proach is obviously incomplete. For instance, assume the two x86 assembly code fragments in Figure 3.3 are obtained from compiling two functionally equivalent but syntactically different programs. The two code fragments are functionally equivalent. However, we cannot prove this by syntactically comparing the object files with `diff`. The registers `eax` and `edx` are used in different order and thus the object files are syntactically different.

2. *Soundness*: Concluding functional equivalence with GCC and `diff` is correct iff the optimizations are correct, i.e., if the source program and the target program resulting from optimizing the source program are functionally equivalent. Moreover, aggressive optimizations are allowed to exploit undefined behavior in the semantics of a programming language to generate faster or smaller code, i.e., the optimizations can choose any suitable behavior for a program part that is undefined, e.g., in case of a division by zero. In mutation testing, we want to know whether a seeded fault affects the program semantics. An aggressive optimization, however, (arbitrarily) chooses one behavior in case of undefined behavior in the program. The optimization may choose behavior such that the mutant becomes functionally equivalent to the original program, although an other choice for the undefined behavior whould indeed affect the program semantics. As a result undefined behavior exploited by aggressive optimizations may lead to false-positives in detecting equivalent mutants.

3. *Robustness*: GCC is not designed for detecting functional equivalence but for code generation. The detection results may vary with different GCC versions. Moreover, future improvement to GCC may conflict with the goal of the simple approach, i.e., the development in GCC need not care about creating similar target code for similar source programs.

4. *Controllability*: In the simple approach, GCC is used as a black-box. It is hardly possible to improve the results when GCC and `diff` are unable to conclude functional equivalence.

5. *Scalability*: In the Figures 3.1 and 3.2 we consider one mutant. A generalization of the simple approach to detect equivalent mutants in sets of mutants requires one compilation for each mutant and a pairwise comparison of the resulting object files. However, since in mutation testing the number of mutants is typically high, such an approach will not scale for larger programs.

Due to the undecidability of program equivalence in general, there is no method to give a complete decision procedure for the detection of functional equivalence of arbitrary programs.

Notice that we have used optimization level 1 in Figure 3.2 rather than the more aggressive optimization levels 2 or 3. Optimization level 1 offers moderate optimizations which do not require a significant time overhead at compile-time and are safe in the sense that they do not exploit undefined behavior in the program [StGDC10]. Typically programmers omit undefined behavior in their programs. However, when we seed faults into a program, we cannot guarantee that a seeded fault may result in undefined behavior. Thus, we use a set of optimizations considered less aggressive. The avoidance of aggressive optimizations is an strategy to overcome the soundness problem caused by undefined behavior.

In the next section, we present a decision procedure that attempts to fulfill the other three design considerations (robustness, controllability, and scalability).

3.3 Detecting Equivalent Mutants using Code Optimization

3.3.1 Optimizer Procedure

In this section we present the Optimizer procedure that attempts to prove functional equivalence of a program and one of its mutants using static analysis methods implemented as a sequence of optimizing source-to-source transformations.

Procedure 1 gives the Optimizer procedure in pseudo code. For a given program P, one of its mutants M, and a sequence of optimizing source-to-source transformations T_1, T_2, …, T_n, Optimizer decides whether P is syntactically equivalent with respect to the sequence of transformations. First, Optimizer applies the sequence of optimizing transformations T_1, T_2, …, T_n consecutively to the program P and the mutant M shown in line 2 and 3, respectively. The obtained transformed programs P' and M' are then syntactically compared denoted by the pseudo code function `Compare` in line 4, i.e., the object files of the programs P' and M' are compared line by line. The procedure Optimizer returns 'Yes' if the programs P and M are syntactically equivalent and 'Don't Know' if P' and M' are different. In case of a 'Don't Know' decision, we do not know whether P and M are different or the optimizing source-to-source transformations applied are not sufficient to optimize P and M to the syntactically equivalent programs P' and M'.

Procedure 1: Optimizer

```
Input  : a program $P$, a mutant $M$ of the program, and a sequence $T_1, T_2, \ldots, T_n$ of
         optimizing source-to-source transformations
Output: 'Yes' if $P$ and $M$ are functionally equivalent and 'Don't Know' otherwise
1 begin
2     $P' = T_1(T_2(\ldots T_n(P)\ldots));$
3     $M' = T_1(T_2(\ldots T_n(M)\ldots));$
4     if Compare($P'$,$M'$) then
5         return 'Yes';
6     else
7         return 'Don't Know';
8     end
9 end
```

The Optimizer procedure is similar to the simple approach presented in Section 3.2. However, the procedure does not rely on GCC as a black-box but uses a set of standard optimizing source-to-source transformations.

3.3.2 Optimizing Source-to-Source Transformations

The sequence of optimizing source-to-source transformations T_i, $1 \leq i \leq n$, applied to the original program and one of its mutants is an input parameter of the Optimizer procedure. We leverage optimizing source-to-source transformations from the LLVM compiler infrastructure. These transformations are applied to LLVM source code. The implementation of a source-to-source transformation in LLVM is called a *compiler pass* or simply a *pass*. LLVM's list of analysis and transformation passes [SH11] shows all transformations supported by the LLVM compiler infrastructure. In the following, we discuss the standard optimization pipeline of LLVM for optimization level 1 because we want to avoid aggressive optimizations which may exploit undefined behavior.

The effectiveness of the Optimizer procedure depends on the ability of an optimizing source-to-source transformation to produce syntactically similar code from semantically similar code, i.e., an effective optimizing source-to-source transformation attempts to canonize the source code. On the one hand, for a given optimization criterion, e.g., code size, finding a globally optimal representation of a source program is undecidable. Additionally, in general there is no unique global optimal representation of a source program with respect to a fixed optimization criterion. Suppose we consider code size as optimization criterion, the x86 assembly code fragments in Figure 3.2 are both optimal with respect to the code size criterion. On the other hand, optimizing source-to-source transformations are designed to turn arbitrary source code into more structured code, i.e., compiler passes are usually applied in a particular order to find a locally optimal representation of a program.

We consider a fixed pipeline of optimizing source-to-source transformations. The pipeline is standard when optimization level 1 is considered. However, we are not tied to exactly these pipeline of transformations. In the remaining part of this section, we give a short description of each considered compiler pass based on [SH11] but without discussing technical or implementation details.

1. **Promote Memory to Register.** The pass implements a standard algorithm to construct Static Single Assignment (SSA) form from a given program, i.e., the pass removes alloca and load instruction, transforms store instructions to registers and adds phi instructions to the program when needed. The resulting program is in SSA form and thus each variable is assigned only once which simplifies the implementation of further analysis and optimization passes.

2. **Combine Redundant Instructions.** The pass combines instructions based on simple algebraic simplification patterns. For instance, two additions `%y = add %x,1` and `%z = add %y,1` are transformed to `%z = add %x, 2` assuming that value `%y` is not referenced in later parts of the program.

3. **Simplify the Control Flow Graph.** The pass implements dead code elimination and basic block merging. It removes basic blocks with no predecessors, merges a basic block with its predecessor if there is only one predecessor and the predecessor has no other successors, eliminates phi instructions for basic blocks with a single predecessor, and eliminates a basic block that contains only an unconditional branch.

4. **Re-Associate expressions.** The pass re-associates commutative expression by changing their order such that the new order allows better constant propagation. For instance, an expression `1 + (x + 2)` is transformed to `x + (1 + 2)` exploiting associativity of the

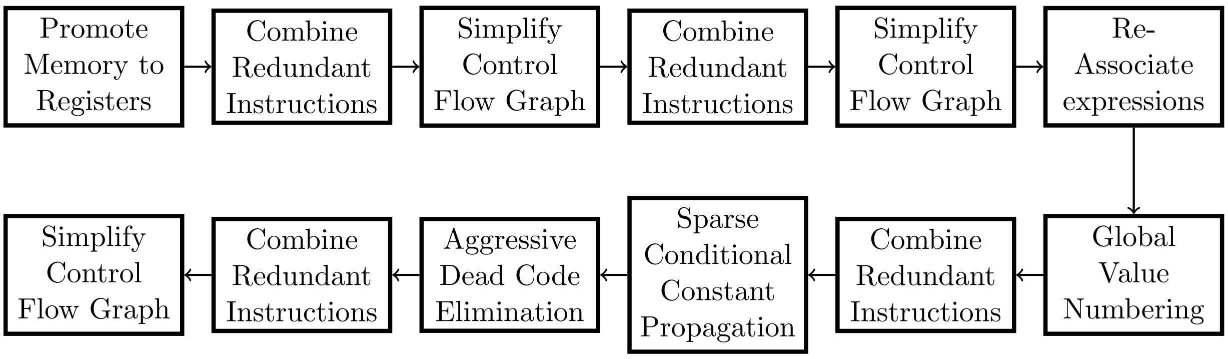

Promote Memory to Registers
Combine Redundant Instructions
Simplify Control Flow Graph
Combine Redundant Instructions
Simplify Control Flow Graph
Re-Associate expressions
Global Value Numbering
Combine Redundant Instructions
Sparse Conditional Constant Propagation
Aggressive Dead Code Elimination
Combine Redundant Instructions
Simplify Control Flow Graph

Figure 3.4: Optimization pipeline: a sequence of optimizing LLVM source-to-source transformation used to optimize our programs.

integer addition. Associative and commutative binary operators which can be re-associated in this way are **`add`**, **`mul`**, **`and`**, **`or`**, and **`xor`** when applied to integer types.

5. **Global Value Numbering.** The pass uses a global map to assign values to variables and expressions such that provable equivalent variables and expression are mapped to the same value. The mapping is then used to simplify variables and expressions by substituting them against the simplest representative from the map with respect to the value.

6. **Sparse Conditional Constant Propagation.** The pass propagates constants using abstract interpretation of the code. Each value that is proved constant is substituted by this constant.

7. **Aggressive Dead Code Elimination.** The pass removes instructions calculating values that are never used. The dead code elimination assumes each variable dead until proven otherwise. The compiler pass is recursively repeated until all values are proven to be used.

3.3.3 Optimization Pipeline

We have chosen an optimization pipeline consisting of 12 optimizing source-to-source transformations shown in Figure 3.4. The individual optimization passes are briefly described in the previous section.

Firstly, we transform the LLVM program into SSA form by applying the "Promote Memory to Registers"-pass. Secondly, we combine redundant instructions and simplify the control flow. Notice that the optimization pipeline uses the "Combine Redundant Instructions"-pass and the "Simplify Control Flow Graph"-pass more than once. Reapplying these transformations after other transformations may result in additional code reduction. We then re-associcate expressions and apply global value numbering, followed by constant propagation and dead code elimination. Lastly, we again combine redundant instructions and simplify the control flow graph.

The considered optimization pipeline is based on the standard LLVM optimization pipelines using optimization level 1. However, we are not tied on exactly these optimizations and, thus, the optimization pipeline allows for easy customization.

Recall that the Optimizer procedure is sound iff the optimizing source-to-source transformations do not affect the program semantics. We have chosen the standard LLVM optimization pipeline for optimization level 1 to avoid soundness problems of optimizations that exploit undefined behavior.

The decision procedure can be considered more robust that the simple approach in the sense that the sequence of optimizing source-to-source transformations is fixed and does not change with future LLVM releases. We assume that the task and the implemented algorithm of a particular compiler pass does not change in future LLVM releases but may be improved in precision or fixed when faulty. On contrast to the simple approach, the optimizations are applied to program code at the middle-end, i.e., Optimizer does not leverage target-specific optimization.

The Optimizer procedure is better controllable than the simple approach by customizing the optimization pipeline. Moreover, it is possible to implement new optimization passes to specifically improve the equivalence detection results. In the next section, we address the scalability issue and generalize the Optimizer procedure to take equivalence decisions for a set of mutants rather than for a single mutant of the original program.

3.4 Detecting Redundant Mutants using Code Optimization

3.4.1 MetaOptimizer Procedure

Procedure 2 shows the MetaOptimizer procedure, an extension of the Optimizer procedure. The MetaOptimizer procedure attempts to detect equivalent and redundant mutants encoded in a meta-mutant. The inputs of MetaOptimizer are a meta-mutant M and a sequence $T_1, T_2, \ldots, T_n$ of optimizing source-to-source transformations. The procedure maintains a map Φ that collects the information of detected equivalent and redundant faults. The map Φ maps a fault id i to a set of fault ids $j_1, j_2, \ldots, j_m$, where the faults represented by the fault ids j_k, $1 \leq k \leq m$, are functionally equivalent to the fault represented by the fault id i. The map is implemented using a union-find data-structure which effectively keeps track of partitions of a set of elements. The union-find data-structure provides two operations: `Find` and `Union`. In the following Φ is a union-find data-structure over the set of all fault ids. Given a specific fault id, the `Find(`Φ`,` id`)` operation looks up the set of fault ids in Φ to which id belongs. The `Union(`Φ`,` S_1`,` S_2`)` operation merges two sets of fault ids S_1 and S_2 into new a set and updates the map Φ.

Firstly, the procedure MetaOptimizer creates a map of all fault ids denoted by the pseudo code function `SetupMap` in line 2. Secondly, the procedure optimizes the meta-mutant M in line 3 using the transformations T_i, $1 \leq i \leq n$, similar to the Optimizer procedure. Then, the procedure compares each basic block bb line by line with each other basic block bb', where $bb' \neq bb$, denoted by the pseudo code function `Compare` in line 8. We denote the fault ids of the basic blocks bb and bb' by id and id', respectively. The mapping from a basic block to its fault id corresponds to a look up of the metadata attached to terminator instruction of the basic block, which is denoted by the pseudo code function `GetID` in line 5 and 7. If the two basic blocks bb and bb' are syntactically equivalent, we look up the sets of fault ids the corresponding ids id and id' belong, and merge the two sets of fault ids which is shown in line 9. Finally, the procedure returns the symmetric map Φ in line 13. The map contains the information of the detected redundant mutants. Recall that the fault id 0 is reserved for the original program behavior. Thus, the operation `Find(`Φ`,0)` returns the set of fault ids of detected equivalent mutants.

Procedure 2: MetaOptimizer

Input : a meta-mutant M and a sequence $T_1, T_2, \ldots, T_n$ of optimizing source-to-source transformations

Output: a map Φ that maps each fault id to a set of fault ids representing detected redundant faults

```
begin
    $\Phi$ = SetupMap();
    $M' = T_1(T_2(\ldots T_n(M)\ldots))$;
    foreach basic block $bb$ in $M'$ do
        id = GetID($bb$);
        foreach basic block $bb' \neq bb$ in $M'$ do
            id' = GetID($bb'$);
            if Compare($bb$,$bb'$) then
                Union($\Phi$, Find($\Phi$, id), Find($\Phi$, id'));
            end
        end
    end
    return $\Phi$;
end
```

3.5 Summary

In this chapter, we have shown a simple approach that attempts to detect equivalent and redundant mutants using source-to-source transformations. We have started with a motivation showing that the GNU compiler collection (GCC) and the Unix tool `diff` can be used to show functionally equivalence of similar programs. From this idea, we have developed two procedures Optimizer and MetaOptimizer. The first aims to show functionally equivalence of a program and one of its mutants. The latter generalizes the idea and attempts to detect equivalent and redundant faults encoded as a meta-mutant. Both procedures use static analysis techniques leveraging source-to-source transformations. Then, we have presented a standard optimization pipeline using source-to-source transformations provided by the LLVM compiler infrastructure. Finally, we have discussed some implementation details that need to be dealt with when the MetaOptimizer procedure is implemented using the LLVM compiler infrastructure and the meta-mutant construction from the previous chapter.

Chapter 4

Detecting Non-Equivalence

4.1 Introduction

The following chapter is based on Riener et al. [RBF11]. We present a symbolic procedure, called SymBMC which attempts to disprove functional equivalence of a program and a set of its mutants (Section 4.2). The set of mutants is represented by a meta-mutant. Our approach uses bounded model checking to search for one counterexample for each fault. The procedure unrolls the meta-mutant and encodes the equivalent mutant problem into a logic formula over the theory of bit-vectors. We start the description with a simplified version, called SimplifiedBMC which proves non-equivalence of a single mutant and the original program (Section 4.2.1). We then discuss unwinding (Section 4.2.2) and the encoding of an LLVM program as logic constraints (Section 4.2.3) in detail. Next, we present a complete procedure, called SymBMC (Section 4.2.4). Lastly, we discuss related work (Section 4.3 and summarize the chapter (Section 4.4).

4.2 Symbolic Bounded Model Checking

4.2.1 Simplified Symbolic Procedure

In Procedure 3, we give the pseudo code of the symbolic procedure "SimplifiedBMC" which attempts to show non-equivalence of the original program P and one of its mutants M with respect to a given unrolling bound k. The procedure first generates a model of the original program P and a model of the mutant M. Both programs are unrolled with respect to the unrolling bound k and then translated into the bit-vector formulae f_k and f'_k, respectively. In Procedure 3, the encoding task and translation into bit-vector formulae is denoted by the pseudo code function `Encode` in the lines 2 and 3. We denote the program inputs and outputs by fixed-length sequences of LLVM values. The sequences (i_j^P) and (i_j^M), $1 \leq j \leq n$, denote the program inputs of the original program and the mutant, where n is the number of program inputs. The sequences (o_l^P) and (o_l^M), $1 \leq l \leq m$, denote the program outputs, where m is the number of program outputs.

Next, SimplifiedBMC generates a formula g, called *propagation condition*, shown in line 4. The propagation condition asserts equal inputs of the program and the mutant, i.e., $i_j^P = i_j^M$ for all j, and at least one pair of different outputs, i.e., $o_l^P \neq o_l^M$ for some l. We call the conjunction of the propagation condition g and the encoded models f_k and f'_k, the *miter formula* s. The construction of the miter formula is shown in line 5.

Finally, we solve the miter formula s with an SMT solver represented by the pseudo code function `Solve` in line 6. A satisfying assignment corresponds to a distinguishing test case which results in a different external observable outputs for the original program P and the mutant M if executed. We extract the test case from the satisfying assignment by collecting the values of the program inputs and outputs, respectively, denoted by the pseudo code function `ExtractCEX`. If no satisfying assignment exists, i.e., the miter formula is unsatisfiable, then the mutant is functionally equivalent to the original program with respect to the unrolling bound k. Then, no counterexample of length k exists that shows non-equivalence of the original program and the mutant. In such a case, we generally have no knowledge whether the unrolling bound k was chosen too small or the mutant is functionally equivalent.

Procedure 3: SimplifiedBMC

Input : a program P, a mutant of the program M, and a maximum unrolling bound k
Output: a test case that kills the mutant M with respect to the maximum unrolling bound k if P and M are non-equivalent and EQUIVALENT otherwise

```
1  begin
2      $((i_j^P), (o_l^P), f_k) := \texttt{Encode}(P, k)$;
3      $((i_j^M), (o_l^M), f'_k) := \texttt{Encode}(M, k)$;
4      $g := \bigwedge_{j=1}^{n} (i_j^P = i_j^M) \wedge \bigvee_{l=1}^{m} (o_l^P \neq o_l^M)$;
5      $s := f_k \wedge f'_k \wedge g$;
6      if Solve(s) = SATISFIABLE then
7          return ExtractCEX($s$);
8      else
9          return EQUIVALENT;
10     end
11 end
```

4.2.2 Unwind the Program

We follow a bounded model checking approach that generates counterexamples of finite length, specified by the maximum unrolling bound k. Program unwinding is standard in software BMC. The procedure SimplifiedBMC unwinds the program by recursively unrolling loops in the program. Loop unrolling is implemented by the LLVM infrastructure as a compiler pass. However, the compiler pass unrolls loops iff the pass is able to determine the maximum unrolling number which in case of input-dependent loops is not possible. In the current implementation, the symbolic procedure quits if unrolling a loop entirely is impossible. In such a case, the implementation of the symbolic procedure is unable to check functional non-equivalence. During loop unrolling, we inline function calls. Thus, after unwinding the program consists of a single entry function with no loops.

4.2.3 Encoding the Unrolled Program

In the encoding task, we formulate logic constraints that represent the semantics of the unrolled program. In order to simplify the encoding, we first translate the unrolled program into

SSA form. Recall that the translation into SSA form is provided by the LLVM compiler infrastructure as a standard source-to-source transformation , called promote memory to registers (Section 3.3.2). We encode the program into a logic formula over the theory of quantifier-free bit-vectors. A symbolic variable is either a *bit-vector variable* or a *binary decision variable*. A bit-vector variable is a finite sequence of Boolean values with a fixed length. A binary decision variable is a single Boolean value. Bit-vector variables can be used to encode arbitrary information of finite length. We solve the bit-vector formula with an Satisfiability Modulo Theories (SMT) solver which supports the theory of bit-vectors. Such an SMT solver operates on top of a Satisfiability (SAT) solver and provides additional word-level operations (e.g. addition, if-then-else, etc.) for the manipulation of bit-vector variables.

For each program variable x, we introduce a bit-vector variable bv_x of corresponding size. For instance, a integer variable is encoded as a bit-vector of length 32 assuming a 32-bit processor architecture. For each basic block with label l in the program, we introduce a binary decision variable bb_l. The vale of the variable bb_l is 1 if and only if the basic block labeled with l has been executed.

Suppose $bv_{r_{dest}}$, bv_v, $bv_{v_{op_1}}$, $bv_{v_{op_2}}$, bv_{c_1}, bv_{c_2}, $\ldots$, bv_{c_n} are bit-vector variables denoting the register r_{dest} and the values v, v_{op_1}, v_{op_2}, c_1, c_2, $\ldots$, c_n, respectively. The encoding of the individual instruction types is straightforward.

- The load instruction

$$r_{dest} \texttt{ = } \mathbf{load}\ v$$

  in a basic block labeled with l is mapped to an implication

$$bb_l \rightarrow (bv_{r_{dest}} = bv_v).$$

- The binary operator instruction

$$r_{dest} \texttt{ = } \texttt{binop}\ v_{op_1},\ v_{op_2}$$

  in a basic block labeled with l is encoded by mapping the mnemonic `binop` of the instruction to its SMT counterpart bv_{binop} resulting in an implication

$$bb_l \rightarrow (bv_{r_{dest}} = bv_{binop}(bv_{v_{op_1}},\ bv_{v_{op_2}})).$$

- The branching instruction

$$\mathbf{br}\ v\ \mathbf{label}\ l_{true},\ \mathbf{label}\ l_{false}$$

  in a basic block labeled with l is encoded into a conjunction of implications

$$((bb_l \wedge v) \rightarrow bb_{l_{true}}) \wedge ((bb_l \wedge \neg v) \rightarrow bb_{l_{false}}).$$

- The phi instruction

$$r_{dest} = \mathbf{phi}\ [v_{op_1}, l_1], [v_{op_2}, l_2], \ldots, [v_{op_n}, l_n]$$

in a basic block labeled with l is encoded into a sequence of nested logic `ite` (if-then-else)-operations

$$\begin{aligned}
bb_l \rightarrow \texttt{(ite(}&bv_{c_1}\texttt{,}\ bv_{r_{dest}} \texttt{ = } v_{op_1}\texttt{,}\\
&\texttt{ite(}bv_{c_2}\texttt{,}\ bv_{r_{dest}} \texttt{ = } v_{op_2}\texttt{,}\\
&\texttt{...}\\
&\texttt{ite(}bv_{c_{n-1}}\texttt{,}\ bv_{r_{dest}} \texttt{ = } v_{op_{n-1}}\texttt{,}\ bv_{r_{dest}} \texttt{ = } v_{op_n}\texttt{)}\\
&\texttt{...}\\
&\texttt{)))}
\end{aligned}$$

where the value c_i, $1 \leq i \leq n$, denotes the logic condition under which the control flow transfers from the basic block labeled with l_i to the basic block labeled with l. The value c_i is calculated from the branching instructions of the basic block labeled with l_i.

Finally, we constrain the binary decision variable corresponding to the initial basic block of the program to be true which forces each execution of the program to enter the initial basic block. The resulting logic formula is satisfiable if and only if there is an execution path from the initial basic block to the program's exit and the unrolling bound k is sufficient to unroll the program.

4.2.4 Symbolic Procedure

The symbolic procedure SymBMC is shown in Procedure 4. The procedure is similar to SimplifiedBMC but decides the functional equivalence for a set of mutants and the original program: the inputs of the procedure are a meta-mutant M and a maximum unrolling bound k. The meta-mutant contains a set of faults each guarded by a condition that checks whether the specific fault should be enabled. The output of SymBMC is a set Ψ of test cases. The procedure generates a test bench from scratch, assuming that Ψ is initially empty, which is shown in line 2.

Firstly, SymBMC unrolls the meta-mutant M twice, shown in lines 3 and 4. The bit-vector variables id^P and id^M denote the value assigned to the global variable `FAULT_ID` in the logic formulae f_k^P and f_k^M. The formula f_k^P encodes the semantics of the original program and, thus, we fix id^P to 0. Consequently, in the formula f_k^P all seeded faults are disabled. The formula f_k^M encodes the semantics of all faulty programs considered by the meta-mutant. The symbolic variable id^M is kept unconstrained, which corresponds to an existential quantification of the variable id^M.

We then construct a miter formula s shown in lines 5 and 6 similar to the construction used in the procedure SimplifiedBMC. The miter formula s encodes the equivalent mutant problem. The formula is satisfiable if a counterexample exists that proofs non-equivalence of any of the faults represented by the meta-mutant. We hand the formula to an SMT solver that aims to produce a satisfying assignment. From the assignment, we extract the counterexample denoted by the pseudo code function `ExtractCEX` shown in line 8. A counterexample is an assignment for the symbolic variables denoting the program inputs and outputs. Additionally, we extract the fault id that has been detected, denoted by the pseudo code function `ExtractFaultID` in line 9.

We collect the counterexamples in Ψ. Actually, we strive for one satisfying assignment for each possible value of id^M. Thus, we add constraint to the miter function that forces a different value for id^M for the next satisfying assignment. The miter formula is then solved again in order to detect another fault. Finally, the miter formula becomes unsatisfiable and no remaining fault can be detected. The unsatisfiability of the formula proofs functional equivalence of all remaining faults with respect to the unrolled model, i.e., none of the remaining faults result in a different externally observable output with respect to the maximum unrolling bound k.

Procedure 4: SymBMC

```
Input  : a meta-mutant $M$ and a maximum unrolling bound $k$
Output: a list of test cases $\Psi$ that guaranteed kills all non-equivalent mutants of the
        meta-mutant $M$ with respect to the maximum unrolling bound $k$
begin
    $\Psi := \emptyset$;
    $((i_j^P), (o_l^P), f_k^P) :=$ Encode$(M, k) \wedge (id^P = 0)$;
    $((i_j^M), (o_l^M), f_k^M) :=$ Encode$(M, k)$;
    $g := \bigwedge_{j=1}^{n} (i_j^P = i_j^M) \wedge \bigvee_{l=1}^{m} (o_l^P \neq o_l^M)$;
    $s := f_k^P \wedge f_k^M \wedge g$;
    while Solve(s) = SATISFIABLE do
        $\Psi = \Psi \cup$ ExtractCEX$(s)$;
        $s = s \wedge (id^M \neq$ ExtractFaultID$(s))$;
    end
    return $\Psi$;
end
```

4.3 Related Work

The procedures SimplifedBMC and SymBMC are based on BMC and SMT. BMC is an effective technique to generate counterexamples of finite-length, which we interpret as test cases. SMT comes with efficient decision procedures to search for satisfying assignments.

Offutt et al. [Off88, DO91b] proposed a similar approach to automatically generate test cases based on solving a Constraint Satisfaction Problem (CSP). They formulated three conditions (reachability, necessity, and sufficiency) for each fault which need to be satisfied to detect it. The *reachability condition* describes the domain of assignments to the program variables which reach the location of the seeded fault. The *necessity condition* expresses the domain of assignments to the program variables that results in a different program state when executed on the original program and the mutant, respectively. These assignments are said to *infect* the program state. The *sufficiency condition* describes the domain of assignments to the program variables that propagate the infected program state to an externally observable output. The three conditions are encoded as algebraic constraints over the program variables. A satisfying assignment for the conjoined algebraic constraints (if one exists) is a distinguishing test case which results in a different externally observable output. However, CBT was proposed as an approximation technique that does not exploit the sufficiency condition [DO91b]. Thus, CBT is sound but incomplete, allowing the construction of “false negative” test cases that do not lead to the detection of a seeded fault. Experiments for CBT were applied to Fortran focusing on small programs [Off88, DO91b].

Our SymBMC procedure is sound and complete with respect to the given unrolling bound. The procedure generates a finite model of the meta-mutant of the program, i.e., the original program which contains all seeded faults and additional control logic to enable and disable the individual faults. The meta-mutant is encoded as a set of logic constraints over the theory of bit-vectors. We use an SMT solver to decide whether the logic constraints are satisfiable. The SMT solver reasons incrementally, i.e., the solver re-uses (learns) knowledge of the structure of the logic constraints when called more that ones.

Clarke et al. [CKOS04] implemented the *C Bounded Model Checker* (CBMC), i.e., a tool that uses bounded model checking to proof whether an ANSI-C program conforms to its specification given as a set of local assertions in the source code. CBMC was used in lots of applications including fault localization, functional equivalence checking, determining worst-case execution time, and test case generation. For instance, Clarke and Kroening used CBMC to check functional equivalence of an ANSI-C program and a VHDL description. Sinz and Post [PS09] attempted to prove functional equivalence of two difference ANSI-C implementations of the *Advanced Encryption Standard* (AES) cipher using CBMC. In both cases the programs are actually semantically equivalent and CBMC is queried to provide prove. Only in case of a real bug CBMC would find a counterexample. We use BMC to show the presence of bugs, i.e., we iterative check symbolic equivalence between the original program and one of the mutants encoded in a meta-mutant and collect the counterexamples as test cases. Our implementation of BMC is different from CBMC: we use a standard compiler to transform the source language into the Compiler's intermediate representation and transform the intermediate program into logic constraints. Thus we avoid issues like building a language parser. Moreover, our implementation can not only process ANSI-C programs but programs written in any source languages an LLVM front-end compiler exists (especially C++).

Our procedures combine testing and formal methods, i.e., a combination which is occasionally made: Ammann et al. [ABM98] presented a mutation testing approach which injects faults into specifications. A model checker is then used to produce a counterexample which serves as a test case. In this spirit, Okun et al. [OBY02] discusses two approaches to obtain a counterexample that propagates to an externally observable output. One of their approaches is *state-machine duplication* which is similar to our SymBMC procedure. State-machine duplication creates a duplicate of a given state-transition system, injects a fault into the transition graph, i.e., a change to a single transition, and adds a temporal logic formula asserting equal externally observable outputs for the original and the mutated state-transition system. A counterexample generated by a model checker is a distinguishing test case. State-machine duplication doubles the number of states for each seeded fault. We encode a set of mutants into a single meta-mutant. The size of the meta-mutant grows linearly with the size of the original program and linearly with the number of mutants.

Wotawa et al. [WNA10] generate distinguishing test cases from an ANSI-C program with a constraint solver and use the test cases for fault localization. However, the formalization of the logic constraints is different to our formalization. We use an SMT solver with a standardized input language [RT06, BST10]. Thus our implementation can be customized by replacing the SMT solver back-end. Any SMT solver can be used that supports the theory of bit-vectors. Constraint solvers support a much richer input language which makes the modeling of CSP simple but prevents a possible replacement of the solver back-end.

4.4 Summary

In this chapter, we have discussed symbolic bounded model checking to generate counterexamples that disprove functional equivalence of a mutant and the original program. Our procedure SymBMC unrolls a meta-mutant and encodes the unrolled model of the meta-mutant into a logic formula over the theory of bit-vectors. We incrementally solve the formula and constraint the symbolic variables to detect in each iteration another fault. Finally, the formula becomes unsatisfiable which corresponds to a prove of functional equivalence of all undetected faults.

Chapter 5

Experimental Evaluation

5.1 Introduction

In the following chapter, we present experimental results obtained with our prototype implementations of the procedures MetaOptimizer and SymBMC. Firstly, we briefly describe the environment setting (Section 5.2), the benchmark programs (Section 5.3), and the experimental evaluation of our case study (Section 5.4). We then list and discuss experimental results for the procedure MetaOptimizer (Section 5.5.1) and the procedure SymBMC (Section 5.5.2), where each procedure is separately applied to the meta-mutant of the benchmark programs, respectively. Finally, we describe experimental results (Section 5.5.3) when both procedures are applied to the meta-mutants of the benchmark programs one by another.

5.2 Environment Setting

All the experiments were conducted on a PC with an AMD Athlon$^{\mathrm{TM}}$ 64 X2 Dual Core Processor 6000+ with two 3 GHz cores and 4 GB RAM. The operating system was Linux 2.6.34.

We implemented the MetaOptimizer and the SymBMC procedure in two prototype tools written in C++ using the libraries Low Level Virtual Machine (LLVM) 2.8 [LA10] and Boost 1.46.0 [BCL10]. Additionally, the prototype implementation of SymBMC is based on an SMT solver. We present experimental results for the SMT solvers Boolector 1.4 [BB09] (with the PicoSAT back-end) and Z3 2.15 [MB08].

We use the API interface to exchange information between our prototype tool and the SMT solvers. Our interface to the SMT solvers is implemented as a generic wrapper class which respects the SMT library standard 1.2 [RT06] or 2.0 [BST10]. However, we do not exploit SMT commands provided by the individual solvers that extend the SMT library standard, e.g., Z3 provides the bit-vector `and` operation for an arbitrary number of operands but we rather use a sequence of nested bit-vector `and` operations with two operands when needed. Thus, our implementation can be extended to use any SMT solver which respects the SMT library standard.

5.3 Benchmark Programs

The benchmark programs are similar to the benchmark programs considered by Offutt et al. [OP97]. However, we have translated the programs from Fortran-77 to ANSI-C. The characteristics of the benchmark programs are listed in Table 5.1. The first column name the benchmark program. The next three columns give the number of symbolic input variables, the number of LLVM instruction, and the number of basic blocks of the original program, respectively. The last two columns focus on the meta-mutant. They list the number of instructions of the meta-mutant containing all faults with respect to our fault model and the number of seeded faults. We have not mentioned time for the construction of the meta-mutant because the construction took only a few seconds for each benchmark program.

In the following, we briefly describe the individual benchmark programs.

- The benchmark program `min` determines (and returns) the minimum of two given input variables.
- The program `islower` checks whether a given input value is lower cased.
- The benchmark program `findmin` searches the minimum value within an integer array of fixed size. Currently, our prototype tool does not support arrays. We use a custom transformation to rewrite the integer array into a set of individual integer variables. The numbers 3, 5, and 10 give the size of the array.
- Given three input values, the benchmark program `minmax` determines the minimum *min* and the maximum *max* of the given inputs and returns the third value in between of *min* and *max*.
- The benchmark program `trityp` classifies the type of a triangle: given three input values denoting the side lengths of a triangle, the program determines whether the triangle is equilateral, isosceles, scalene, or not a triangle, which is denoted by the constant integer values 1 to 4.
- The benchmark program `mult` calculates a 2x2 matrix multiplication $C = A \cdot B$, where A and B are matrices. The elements of A and B are provided as input parameters. The programs returns the sum of all elements of C.

We have not considered programs with recursion, arrays, pointers or floating point arithmetic. None of these ANSI-C constructs are currently supported by our prototype implementation of the SymBMC procedure.

5.4 Experimental Evaluation

Figure 5.1 sketches the experimental evaluation. We evaluate the procedures MetaOptimizer and SymBMC in three experiments which is represented in the figure by a set of boxes arranged in a tree-like structure. Each box denotes an action in an experiment and each maximal sequence of boxes (from top to bottom) denotes a complete experiment. In the figure, we number the experiments form 1 to 3.

We translate each program first into the LLVM intermediate representation and then construct the meta-mutant by seeding a set of artificial faults into the LLVM program with respect

Name	#Variables	#Instructions (Original Program)	#Basic Blocks	#Instructions (Meta-Mutant)	#Seeded Faults
min	2	24	4	144	17
islower	1	20	6	91	19
findmin3	3	40	8	238	33
findmin5	5	58	12	342	49
findmin10	10	103	22	602	89
minmax	3	52	12	189	46
trityp	3	116	30	1061	206
mult	8	40	2	1345	53

Table 5.1: Characteristics of the benchmark programs.

to our fault model. The translation to LLVM and the construction of the meta-mutant is represented in the Figure 5.1 by the box with label “fault instrumentation”.

We have used the fault model from Section 2.5.5 which provides four mutation operators. Detailed numbers are given in Table 5.1. The size of the resulting meta-mutant is linear in the size of the original LLVM program and linear in the number of seeded faults.

The number of seeded faults is lower than the number of instructions for most of the benchmark programs. The trityp benchmark program has several binary operator instruction which significantly increase the number of seeded faults according to our fault model.

In Figure 5.1, the MetaOptimizer procedure is denoted by the two boxes labeled “Optimize” and “Remove and count redundant faults” and the SymBMC procedure is denoted by the box labeled “Create Counterexamples”.

In our case-study, i.e., all three experiments, we attempt to classify the seeded faults for each program into three categories: “equivalent”, “non-equivalent”, and “redundant”. The faults in the category “equivalent” do not affect the program semantics. They are functionally equivalent to the original program. The faults in the category “non-equivalent” are guaranteed to affect the program semantics, i.e., we can effectively disprove functionally equivalence by generating a counterexample. Each fault in the category “redundant” is functionally equivalent to another fault, i.e., we need not determine functional equivalence or non-equivalence for the particular fault but we discard the fault from mutation testing. Moreover, when we detect two redundant faults it does not matter which fault is discarded from mutation testing. According to the definition of redundancy, the two faults are syntactically equivalent and differ only in the value of the fault id.

In general, a fourth category “unknown” is required because of the inherent incompleteness of BMC and our SymBMC procedure, respectively. For each fault in the category “unknown”, we do not know whether the fault is non-equivalent or functionally equivalent to the original program. However, for the benchmark programs under consideration, the number of paths is finite and unrolling yields a model containing the complete path information. Thus, after the SymBMC procedure has been applied to the meta-mutant, we exactly know whether a mutant

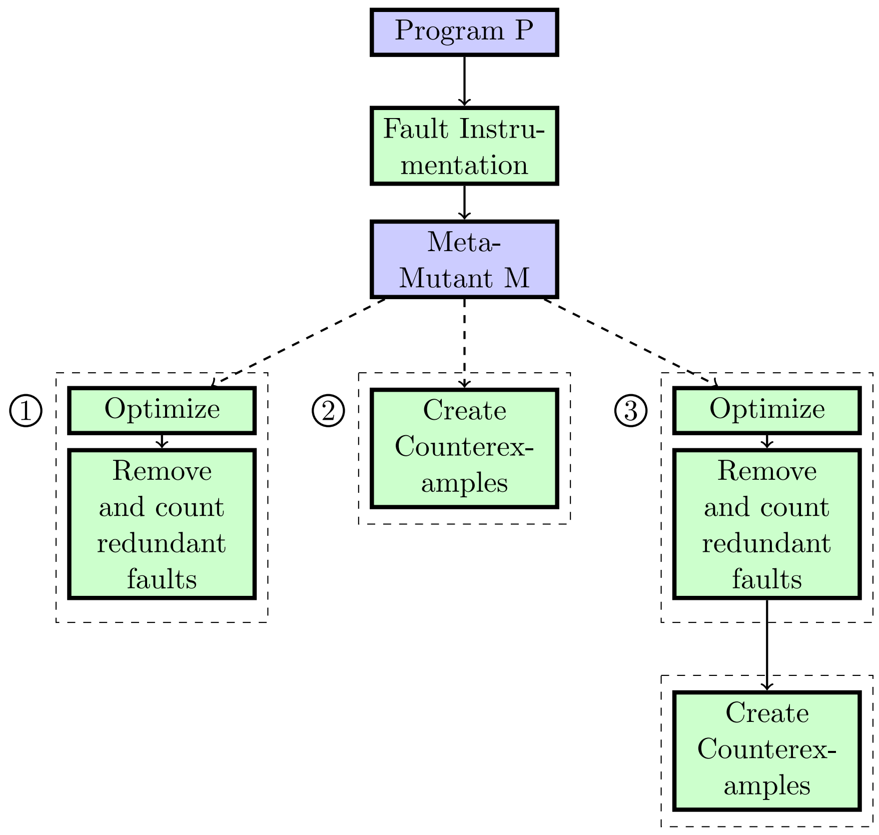


Figure 5.1: High-level overview of the experimental evaluation consisting of three experiments. A box either refers to an inputs (or a products) of the evaluation process or to an action performed on the inputs.

is functionally equivalent or can be detected by a test case.

Detecting redundant faults decreases the computational expense of mutation testing. From a maximal set of redundant faults, we generate only one test case. Moreover, notice that functionally equivalent faults are a special case of redundant faults, where the faults are functionally equivalent to the original program. Thus, detecting redundant faults, as well, decreases the number of equivalent faults. For instance, consider an equivalent fault and assume we have yet not proved that the fault is actually equivalent. We may not be able to decide functionally equivalence of this particular fault but when we determining that the fault is redundant, i.e., we show that the fault is functionally equivalent to another fault, we discard one of the faults from the mutation testing process. Consequently, the total number of functionally equivalent faults is decreased by one.

In the first experiment (marked in Figure 5.1 with number 1), we apply MetaOptimizer individually to the source code of the meta-mutant to find functionally equivalent and redundant faults. In the second experiment (marked in Figure 5.1 with number 2), we apply SymBMC individually to the source code of the meta-mutant to find non-equivalent faults. Finally, in the third experiment (marked in Figure 5.1 with number 3), we apply both procedures, first MetaOptimizer and then SymBMC, to the source code of the meta-mutant, which may change the results because MetaOptimizer transforms the meta-mutant into an optimized program. Intuitively, the optimized meta-mutant has fewer instructions and is, thus, encoded into a shorter

Name	#Instructions (After Optimization)	Compaction	#Seeded Faults	#Redundant Faults	#Unclassified Faults
min	92	0.639	17	0	17
islower	66	0.725	19	2	17
findmin3	146	0.613	33	3	30
findmin5	216	0.632	49	3	46
findmin10	391	0.650	89	3	86
minmax	189	0.612	46	3	43
trityp	625	0.589	206	23	183
mult	891	0.662	48	0	48

Table 5.2: Detected redundant faults with the MetaOptimizer procedure.

logic formula which makes solving the formula easier.

We check the validity of the experimental results obtained from MetaOptimizer and SymBMC with two approaches. Firstly, by comparing the detected faults of the MetaOptimizer and the SymBMC procedures. A fault proven to affect the program semantics with SymBMC must not be detected with MetaOptimizer. A fault detected as functionally equivalent with MetaOptimizer needs to remain undetected with SymBMC. An inconsistency would indicate a problem with one of our approaches. Secondly, we extract from each counterexample generated by SymBMC a test case, simulate the test case on the original program, and compare the outputs obtained by simulation with the outputs predicted by the test case. Our procedures passed both validity checks on each of the benchmark programs, respectively, i.e., the results of the SymBMC and MetaOptimizer procedures are consistent and the inputs of the test case result in outputs consistent with the test case when executed on the program.

5.5 Experimental Results

5.5.1 Results of Experiment 1

Table 5.2 shows the results of our first experiment where we apply the MetaOptimizer procedure using the optimization pipeline described in Section 3.3.3 to the meta-mutant. The table is similar to Table 5.1. The first column names the benchmarks program and the next two columns list the number of remaining instruction of the meta-mutant after optimization and the *compaction* of the meta-mutant, i.e., the ratio of the number of remaining instructions after optimization to the number of instructions before optimization. The last three columns give the number of seeded faults F_S, the number of detected redundant faults F_R and the number of unclassified faults $F_U = F_S - F_R$. The procedure MetaOptimizer detected some redundant faults but did not find any equivalent faults. We have omitted the column of detected equivalent faults. The time required for the static analysis accounts on average for less than a second and, thus, we have not reported the time in the table.

Name	F_S	Cex	$T_{Boolector}$ [s]	T_{Z3} [s]
min	17	16	0.47	0.43
islower	19	18	0.26	0.41
findmin3	33	23	6.56	23225.20
findmin5	49	37	31.76	?
findmin10	89	72	278.45	?
minmax	46	43	8.11	3.99
trityp	206	196	207.10	3259.70
mult	48	48	66.57	?

Table 5.3: Counterexample generation with the SymBMC procedure.

In contrast to the motivation from Section 3.2, we have not found any equivalent faults with MetaOptimizer. A possible reason is that the optimizing transformations provided by the LLVM compiler infrastructure are not effective enough. Effective optimizing transformations exploit several corner cases, e.g., the GCC was improved over years and there are still several reports on missing optimizations.

5.5.2 Results of Experiment 2

Table 5.3 gives the results for our second experiment that uses the SymBMC procedure. SymBMC leverages an SMT solver as back-end using the theory of bit-vectors, particularly, Boolector 1.4 [BB09] and Z3 2.15 [MB08].

The table from left to right names the benchmark program, gives the number of seeded faults F_S, the number of generated counterexamples (test cases) Cex, and the time $T_{Boolector}$ and T_{Z3} required to generate the counterexamples in seconds with the SMT solvers Boolector and Z3, respectively.

A question mark in the table denotes that the SMT solver was not able to classify all faults within 7 hours (25200 seconds). We do not report the time needed to encode the programs into bit-vector formulae which takes on average less than one second for each of the benchmark programs.

For the benchmark programs, SymBMC precisely detects all non-equivalent faults. The remaining faults are functionally equivalent to the original program. Thus, the differences between the number of seeded faults and the number of test cases generated equals the number of equivalent mutants. For, instance 10 of 206 seeded faults in the meta-mutant of the benchmark program trityp do not affect the program semantics.

Both SMT solvers generated counterexamples for the same set of faults. However, for some of the benchmark programs Boolector was significant faster in solving the SMT formulae. One possible reason for such a time gap is the heuristic used to select variables by the SMT solver. If the heuristic initially chooses the right variables and assigns values to them, solving the formula becomes polynomial in the number of variables. The existence of such backdoor variables has been shown in the context of the satisfiability problem [WGS03].

Figure 5.2, Figure 5.3, and Figure 5.4 present details for the counterexample generation with the procedure SymBMC for the benchmark program `trityp`. Figure 5.2 shows the counterexample generation over time. The solid and dashed lines denote the solving process with Boolector

Name	F_S	F_R	F_U	Cex	$T_{Boolector}$ [s]	T_{Z3} [s]
min	17	0	17	16	0.35	0.39
islower	19	2	17	16	0.17	0.29
findmin3	33	3	30	23	4.74	82.61
findmin5	49	3	46	37	45.92	?
findmin10	89	3	86	72	219.46	?
minmax	46	3	43	40	3.09	3.05
trityp	206	23	183	174	130.99	3120.67
mult	48	0	48	48	326.65	?

Table 5.4: Fault classification with MetaOptimizer and SymBMC.

and Z3, respectively. The t-axis gives the accumulated time needed by SymBMC to solve the individual SMT formulae. The F_D-axis give the total number of counterexamples generated by SymBMC. Figures 5.3 and Figure 5.4 concentrate on the SMT solver Boolector. Figure 5.3 is similar to Figure 5.2 but shows the time interval from 0 to 200 seconds in more detail. Figure 5.4 gives the time required to generate the individual counterexamples. We separated the t-axis into continuous intervals of one second. On the F_D-axis, we count the number of individual logic formulae solved by the SMT solver. For instance, for 124 seeded faults (60.19%) of the benchmark program `trityp`, solving the individual logic formulae takes less than or equal to one second per fault. Moreover, for only five faults (2.46%) the solving of the logic formulae takes five or more seconds per fault.

5.5.3 Results of Experiment 3

Table 5.4 gives hte results for our third experiment when both procedures, MetaOptimizer and SymBMC, are applied to the meta-mutant. We use the MetaOptimizer procedure to detect some redundant faults, then we constrain the detected redundant faults, and generate counterexamples for the optimized meta-mutant with the SymBMC procedure. The table shows results for both procedures and, thus, is built similar to Table 5.2 and Table 5.3.

Notice that solving the SMT formulae obtained from the optimized meta-mutant is faster. For instance, the time required to solve the SMT formulae of the benchmark program `trityp` with Boolector is reduced by 37%. Moreover, the time to solve the SMT formulae of the benchmark program `findmin3` is reduced by 99.6%. However, the time may significantly increases for some examples too, e.g., solving the SMT formulae of `mult` requires 4.9x more time than solving the SMT formulae before optimization.

5.6 Summary

In this chapter, we have presented empirical results for the MetaOptimizer and SymBMC procedures. We have applied each procedure separately to a set of benchmark programs and both procedures one by another to the same benchmark programs. The experimental results for MetaOptimizer are inconclusive, i.e., the procedure has not detected any equivalent fault but several redundant faults. The MetaOptimizer procedure, however, is reasonable fast and on none of the benchmark programs did MetaOptimizer need more than a second to detect the

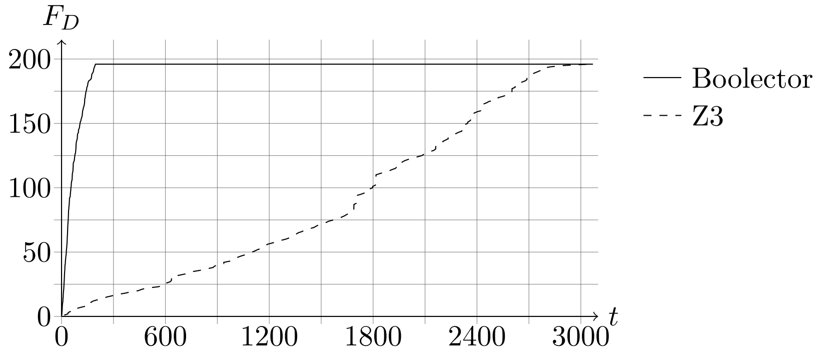


Figure 5.2: Counterexample generation for the benchmark program `trityp` over time. The t-axis shows the accumulated time needed by the procedure SymBMC in seconds. The F_D-axis gives the total number of faults detected by SymBMC.

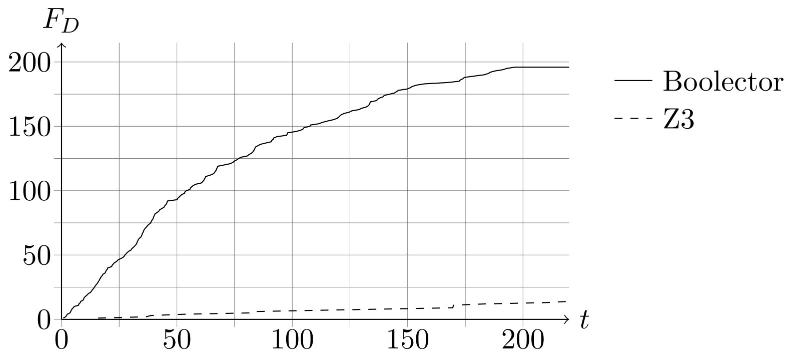


Figure 5.3: Detailed counterexample generation over time. The figure zooms the time interval from 0 to 200 seconds of Figure 5.2.

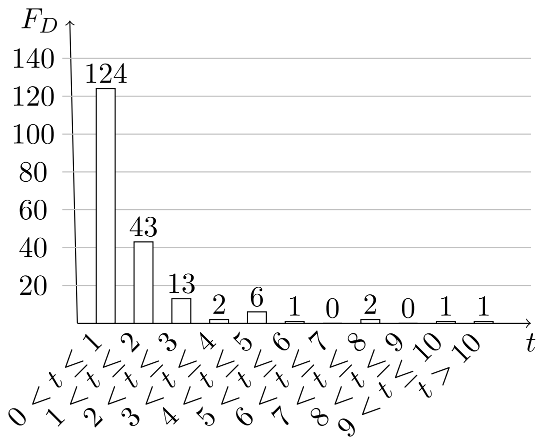


Figure 5.4: Counterexample generation for the benchmark program `trityp` over time. The t-axis separates the time into continues intervals of one second. The F_D-axis shows the number of the logic formulae solved by the SMT solver in the individual continues time intervals.

redundant faults.

The experimental results obtained with SymBMC are promising. We have generated counterexamples for all non-equivalent faults. The results, however, indicate that the SymBMC procedure may not scale to larger programs and its performance strongly depends on the SMT solver in use.

Moreover, using the optimized meta-mutant obtained from MetaOptimizer has sometimes a positive impact on the solving step of the procedure SymBMC, i.e., solving the SMT formula is faster. However, there are as well some cases in which the solving of the SMT formulae obtained from the optimized meta-mutant is more complicated.

Chapter 6

Conclusion and Future Work

In this thesis, we have focused on the equivalent mutant problem, i.e., deciding whether a single syntactic change relative to the original program affects the program semantics.

The problem is essential in mutation testing because without detecting equivalent mutants, we cannot determine how many faults remain undetected after testing. Consequently, the quality assessment by mutation analysis is in general under-estimated.

The equivalent mutant problem is undecidable and, thus, no sound and complete decision procedure exists to overcome the equivalent mutant problem. However, by sacrificing completeness, we can attempt to decide functional equivalence and non-equivalence for several cases.

We present two ideas to attack the equivalent mutant problem: a new code optimization approach which leverages optimizing source-to-source transformations of an optimizing compiler and the counterexample approach which uses symbolic bounded model checking to encode the equivalent mutant problem of the k-times unrolled model into a quantifier-free logic formula using the theory of bit-vectors. The resulting formula is then checked for satisfiability with the aid of an SMT solver.

The first approach, described by the MetaOptimizer procedure, attempts to detect equivalent and redundant faults, whereas the second approach, described by the SymBMC procedure, generates counterexamples which disprove functionally equivalence for particular seeded faults. For each counterexample a test case can be extracted that distinguishes the corresponding mutant from the original program. Thus, the second approach is a test case generation strategy which aims to creating a mutation adequate test bench.

We have implemented the procedures MetaOptimizer and SymBMC into C++ applications. Both procedures operate on the LLVM intermediate representation, i.e., a RISC-like assembly language. We show how to encode a set of faults into a meta-mutant on the level of the LLVM intermediate representation. Our prototype implementations of the two decision procedures then take their equivalence and non-equivalent decisions directly for the meta-mutant which serves as an effective data-structure to reason about a set of seeded faults.

The MetaOptimizer procedure is reasonable fast but can only detect some simple equivalent and redundant faults. However, in our case study, we have not detected any equivalent faults. One reason for this may be insufficient source-to-source transformations provided by the LLVM compiler infrastructure. Our approach was motivated by an experiment in which we have compared object files compiled with GCC and successfully detected functionally equivalent faults. In our experiments, we used GCC as a black-box. The MetaOptimizer procedure is a more flexible implementation based on the LLVM compiler framework but the optimizations are less effective in detecting equivalent and redundant faults. If we repeat the experiment from Section 3.2 with the LLVM C front-end compiler, we are unable to detect the equivalent mutant.

The GCC Dragonegg plugin allows for the replacement of GCC's optimizer and code generator by the optimizer and code generator provided by the LLVM framework. An extension of our tool which uses Dragonegg may result in better equivalent and redundant fault detection results when the code optimization approach is used. Additionally, by modifying Dragonegg, we may be able to leverage GCC's optimizer. The fault injection, i.e., the construction of the meta-mutant, then should be implemented on C source code rather than the LLVM intermediate representation. This conflicts with the idea of having a single mutation testing tool for all programming languages an LLVM compiler front-end exists.

The SymBMC procedure precisely generates counterexample for all non-equivalent faults. However, the procedure may not scale to larger programs but still appears reasonable when paired with an abstraction technique or in the local context of the program.

A major limitation of SymBMC is the restriction to a subset of the LLVM language. The procedure currently is unable to encode instructions using pointer and arrays. In order to allow studying of larger applications, the encoding needs to be extended. The SMT solvers currently in use by SymBMC (Boolector and Z3) support, additionally to the theory of bit-vectors, the theory of arrays which can be exploited to encode LLVM instructions operating on arrays directly. In order to support C pointer expressions the implementation of an untyped (C-like) memory model is needed. This memory model is easily implemented as a byte array and encoded in a formula using the theory of arrays. The resulting formula, however, grows fast when the array is manipulated by an instruction and is harder to solve.

Moreover, we have not considered simulation. A counterexample generated to detect a particular fault has the potential to disprove functionally equivalence of several other faults, too. Thus, simulating the generated counterexamples may improve the performance of the SymBMC procedure and meanwhile reduce the size of the number of generated counterexamples.

Possible future work includes the implementation of optimizing source-to-source transformations specialized to detect functionally equivalent faults. Offutt and Craft [OLR+94] presented some simple transformations. In Section 3.3.2, we have mentioned the source-to-source transformation Global Value Numbering (GVN). GVN implements an algorithm which assigns a number to each variable and expression of the program. Variables and expressions proven functionally equivalent are assigned with equal numbers. This information is then used to remove redundant code from the program. GVN is a light-weight method to detect functionally equivalent expressions based on a simple abstract interpretation mechanism. The abstract interpretation mechanism is kept simple because compiler optimizations should be reasonable fast. We use GVN in our optimization pipeline but do not exploit the information provided by its internal mapping of expressions to numbers. Additionally, replacing the light-weight abstract interpretation mechanism by an SMT solver makes GVN a more powerful tool for equivalence detection. This approach has the potential to result in much better equivalence detection results.

Bibliography

[ABL05] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th International Conference on Software Engineering*, pages 402–411, 2005.

[ABM98] P. E. Ammann, P. E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *IEEE International Conference on Formal Engineering Methods*, pages 46–54, 1998.

[Acr80] A. T. Acree. *On Mutation*. PhD thesis, Georgia Institute of Technology, 1980.

[ADH$^+$06] H. Agrawal, R. A. DeMillo, B. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. Spafford. Design of mutant operators for the C programming language. Technical Report SERC-TR-41-P, Georgia Institute of Technology, 2006. Revision 1.04.

[AMP09] A. Armando, J. Mantovani, and L. Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. *International Journal on Software Tools for Technology Transfer*, 11(1):69–83, 2009.

[AO08] P. Ammann and A. J. Offutt. *Introduction to Software Testing*. Cambridge University Press, Cambridge, 2008.

[AWZ88] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Symposium on Principles of Programming Languages*, pages 1–11, 1988.

[BA82] T. A. Budd and D. Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*, 18(1):31–45, 1982.

[BB09] R. Brummayer and A. Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 174–177, 2009.

[BCCZ99] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Fifth International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'99)*, pages 193–207, Amsterdam, The Netherlands, March 1999. LNCS 1579.

[BCL10] The Boost C++ library, 2010. Available from http://boost.org/. Last visit on 22nd of December, 2010.

[BHvMW09] A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*. IOS Press, 2009.

[BL92] J. R. Burch and D. E. Long. Efficient boolean function matching. In *Proceedings of the International Conference on Computer-Aided Design*, pages 408–411, Santa Clara, CA, November 1992.

[Boe81] B. W. Boehm. *Software Engineering Economics*. Prentice Hall PTR, 1981.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[BST10] C. Barett, A. Stump, and C. Tinelli. The smt-lib standard: Version 2.0, 2010. Available from `http://combination.cs.uiowa.edu/smtlib/papers/smt-lib-reference-v2.0-r10.12.21.pdf`. Last visited on 22nd of December, 2010.

[CFRW91] R. Cytron, J. Ferrante, B. K. Rosen, and M. N. Wegman. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.

[CKOS04] E. Clarke, D. Kröning, J. Ouaknine, and O. Strichman. Completeness and complexity of bounded model checking. In *Verification, Model Checking, and Abstract Interpretation*, pages 85–96, Venice, Italy, January 2004. Springer. LNCS 2937.

[DDH72] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming*. Academic Press, 1972.

[DK11] A. Dereziska and K. Kowalski. Object-oriented mutation applied in Common Intermediate Language programs originated from C#. In *International Conference on Software Testing, Verification and Validation Workshops*, 2011. To Appear.

[DLS78] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, 1978.

[DMM96] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur. Interface mutation: An approach for integration testing. *IEEE Transactions on Software Engineering*, 27(3):228–247, 1996.

[DO91a] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, 1991.

[DO91b] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, 1991.

[FW93] P. G. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering*, 19(8):774–787, 1993.

[GG75] J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. In *Proceedings of the International Conference on Reliable Software*, pages 493–510, 1975.

[GM01] S. Ghosh and A. P. Mathur. Interface mutation. *International Conference on Software Testing, Verification and Validation*, 11(4):227–247, 2001.

[GSZ09] B. J. M. Gruen, D. Schuler, and A. Zeller. The impact of equivalent mutants. In *International Conference on Software Testing, Verification and Validation*, pages 192–199, 2009.

[Ham77] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, SE-3(4):279–290, 1977.

[HLO11] J. Hu, N. Li, and J. Offutt. An analysis of OO muation operators. In *International Conference on Software Testing, Verification and Validation Workshops*, 2011. To Appear.

[How76] W. E. Howden. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering*, 2(3):208 – 215, 1976.

[ISGG05] F. Ivancic, I. Shlyakhter, A. Gupta, and M. K. Ganai. Model checkign C prorams using F-SOFT. In *International Conference on Computer Design*, pages 297–388, 2005.

[JH10] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions of Software Engineering*, To Appear, 2010.

[KO91] K. N. King and A. J. Offutt. A Fortran language system for mutation-based software testing. *Software — Practice & Experience*, 21(7):685–718, 1991.

[LA10] C. Lattner and V. Adve. LLVM language reference manual, 2010. Available from http://llvm.org/docs/LangRef.html. Last visit on 22nd of December, 2010.

[Lat02] C. Lattner. LLVM: An infrastructure for multi-stage optimization. Master's thesis, University of Illinois at Urbana-Champaign, 2002.

[Lio96] J. L. Lions. Ariane 5 flight 501 failure. Technical report, European Space Agency, 1996. Available from http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html.

[LPO09] N. Li, U. Praphamontripong, and A. J. Offutt. An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage. In *International Conference on Software Testing, Verification and Validation Workshops*, pages 220–229, 2009.

[LT93] N. Leverson and C. S. Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, 1993.

[MB08] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[McM94] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, MA, 1994.

[MUO10] P. R. Mateo, M. P. Usaola, and J. Offutt. Mutation at system and functional levels. In *International Conference on Software Testing, Verification and Validation Workshops*, pages 110–119, 2010.

[MW94a] A. P. Mathur and W. E. Wong. An empirical comparison of data flow and mutation-based test adequacy criteria. *Software Testing, Verification and Reliability*, 4(1):9–31, 1994.

[MW94b] A. P. Mathur and W. E. Wong. A theoretical comparison between mutation and data flow based test adequacy criteria. In *22nd Annual ACM Conference on Computer Science*, pages 38–45, 1994.

[Mye79] G. J. Myers. *The Art of Software Testing.* John Wiley & Sons, 1979.

[OBY02] V. Okun, P. E. Black, , and Y. Yesha. Testing with model checker: Insuring fault visibility. In *Proceedings of 2002 WSEAS International Conference on System Science, Applied Mathematics & Computer Science, and Power Engineering Systems*, pages 1351–1356, 2002.

[Off88] A. J. Offutt. *Automatic Test Data Generation.* PhD thesis, Georgia Institute of Technology, Atlanta GA, 1988.

[Off89] A. J. Offutt. The coupling effect: Fact or fiction. *ACM SIGSOFT Software Engineering Notes*, 14(8):131–140, 1989.

[Off92] A. J. Offutt. Investigation of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(1):5–20, 1992.

[OLR$^+$94] A. J. Offutt, A. Lee, G. Rothermel, R. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(2):99–118, 1994.

[OLR$^+$96] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. In *ACM Transactions on Software Engineering and Methodology*, pages 99–118, 1996.

[OP97] A. J. Offutt and J. Pan. Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification, and Reliability*, 7(3):165–192, 1997.

[OPTZ96] A. J. Offutt, J. Pan, K. Tewary, and T. Zhang. An experimental evaluation of data flow and mutation testing. *Software — Practice & Experience*, 26(2):165–176, 1996.

[OU00] A. J. Offutt and R. H. Untch. Mutation 2000: Uniting the orthogonal. In *Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, pages 45–55, 2000.

[OV96] A. J. Offutt and J. M. Voas. Subsumption of condition coverage techniques by mutation testing. Technical Report ISSE-TR-96-100, Department of Information and Software Systems Engineering, George Mason University, 1996.

[Pra95] V. R. Pratt. Anatomy of the pentium bug. In *Proceedings of the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 97–107, 1995.

[PS09] H. Post and C. Sinz. Proving functional equivalence of two AES implementations using bounded model checking. In *International Conference on Software Testing, Verification and Validation*, pages 31–40, 2009.

[RBF11] H. Riener, R. Bloem, and G. Fey. Test case generation from mutants using model checking techniques. In *International Conference on Software Testing, Verification and Validation Workshops*, 2011. To Appear.

[RR03] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *International Conference on Automated Software Engineering*, pages 30–39, Montreal, Canada, October 2003.

[RT06] S. Ranise and C. Tinelli. The smt-lib standard: Version 1.2, 2006. Available from `http://combination.cs.uiowa.edu/smtlib/papers/format-v1.2-r06.08.30.pdf`. Last visited on 22nd of December, 2010.

[SH11] R. Spencer and G. Henriksen. LLVM's analysis and tranform passes, 2011. Available from `http://llvm.org/docs/Passes.html`. Last visit on 4th of March, 2011.

[StGDC10] R. Stallmann and the GCC Developer Community. Using the GNU compiler collection, 2010. Available from `http://gcc.gnu.org/onlinedocs/`. Last visit on 22nd of December, 2010.

[SZ10] D. Schuler and A. Zeller. (Un-)covering equivalent mutants. In *International Conference on Software Testing, Verification and Validation*, pages 45–54, 2010.

[Tas02] G. Tassey. The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standards and Technology, 2002.

[UOH93] R. H. Untch, A. J. Offutt, and M. J. Harrold. Mutation analysis using mutant schemata. *ACM SIGSOFT Software Engineering Notes*, 18(3):139–148, 1993.

[Wah03] K. S. H. T. Wah. An analysis of the coupling effect I: Single test data. *Science of Computer Programming*, 48(2-3):119–161, 2003.

[WGS03] R. Williams, C. Gomesa, and B. Selman. Backdoors to typical case complexity. In *International Joint Conference on Artificial Intelligence*, pages 1173–1178, 2003.

[WNA10] F. Wotawa, M. Nica, and K. Aichernig. Generating distinguishing tests using the Minion constraint solver. In *International Conference on Software Testing, Verification and Validation Workshops*, pages 325–330, 2010.