2018

Bytecode-based Multiple Condition Coverage: An Initial Investigation

Srujana Bollina
University of South Carolina

Bytecode-based Multiple Condition Coverage: An Initial Investigation

by

Srujana Bollina

Bachelor of Technology
Jawaharlal Nehru Technological University 2013

Submitted in Partial Fulfillment of the Requirements

for the Degree of Master of Science in

Computer Science

College of Engineering and Computing

University of South Carolina

2018

Accepted by:

Gregory Gay, Director of Thesis

Csilla Farkas, Reader

John R. Rose, Reader

Cheryl L. Addy, Vice Provost and Dean of the Graduate School.

Dedication

I dedicate this thesis to

my mom,

Mrs. Nagamani Bollina,

my dad,

Mr. Nageshwara Rao Bollina,

my sister,

Ms. Spurthi Bollina,

my fiance,

Mr. Sandeep Vuppula,

and all of my friends.

Acknowledgments

I would first like to thank my advisor Dr. Gregory Gay for his constant guidance and support in completing my thesis. Without his motivation and careful supervision, this work would never have taken shape.

I would like to thank my committee members, Dr. Csilla Farkas, and Dr. Jason O'Kane for their time and valuable comments to better my work. I would like to thank Dr. Gordon Fraser for sharing his knowledge and helping me.

I would like to thank the members of my research group, Almulla Hussein K, Salahirad Alireza and Meng Ying for their ideas and discussions during the completion of my thesis.

Finally, I would like to thank my parents, my sister and my fiance for their unconditional love and trust during all times of my life. Without them, none of my success would have been achieved.

Abstract

Masking occurs when one condition prevents another condition from influencing the output of a Boolean expression. Logic-based adequacy criteria such as Multiple Condition Coverage (MCC) are designed to overcome masking at the within-expression level, but can offer no guarantees about masking in subsequent expressions. As a result, a Boolean expression written as a single complex statement will yield test cases that are more likely to overcome masking than when the expression is written as series of simple statements. Many approaches to automated analysis and test case generation for Java systems operate not on the source code representation of code, but on the bytecode. The transformation from source code to bytecode requires simplifying code elements, introducing the risk of masking.

We propose Bytecode-MCC, designed to group related Boolean expressions from the bytecode, reformulate the expressions into a single complex expression, and produce test cases satisfying each combination of conditions in the constructed expression. Bytecode-MCC should produce test obligations that—when satisfied—are more likely to reveal faults in the program logic than tests providing coverage of existing criteria over the simplified bytecode.

A preliminary study has hinted at the potential of this approach. However, Bytecode-MCC is more difficult to achieve than Branch Coverage, and means of increasing coverage are needed to truly test the fault-detection potential of this technique. We propose methods of improving Bytecode-MCC coverage through automated generation that we will explore in future work.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

"No product of human intellect comes out right the first time. We rewrite sentences, rip out knitting stitches, replant gardens, remodel houses, and repair bridges. Why should software be any different?" [79]

Proper verification practices are needed to ensure that developers deliver reliable software. *Testing* is an invaluable, widespread verification technique. Yet, for any reasonably complex software project, testing alone cannot prove the absence of faults. Therefore, developers seek advice on the factors likely to increase the probability of fault detection. As we cannot know what faults exist a priori, dozens of *adequacy criteria*—ranging from the measurement of structural coverage to the detection of synthetic faults [64, 66]—have been proposed to judge testing *adequacy*. In theory, if the goals set forth by such criteria are fulfilled, tests should be *adequate* at detecting faults related to the focus of that criterion. Adequacy criteria such as Statement or Branch Coverage have proven popular in both research and practice, as they are easy to measure, offer clear guidance to developers, and present an indicator of progress [34]. Adequacy criteria also play an important role in search-based test input generation, as they offer optimization targets and serve as strategies that shape the resulting test suite [2].

Masking occurs when one condition—an atomic Boolean variable or subexpression –prevents another condition from influencing the output of the expression. Even if a fault in a Boolean expression is triggered, other parts of that expression—or future expressions encountered along the path of execution—can prevent that fault from

being observed during test case execution.

Sophisticated logic-based adequacy criteria such as Multiple Condition Coverage (MCC) or Modified Condition/Decision Coverage (MC/DC) are designed to overcome masking at the within-expression level. However, as such criteria prescribe testing goals at the individual-expression level, they can offer no guarantees about masking in subsequent expressions. As a result, such criteria are sensitive to how programs are written [29]. A Boolean expression written as a single complex statement will yield robust test cases that are more likely to overcome masking and reveal faults than when the expression is written as series of simple statements.

Many approaches to automated analysis and test case generation for Java systems operate not on the source code representation of code, but on the bytecode [23, 77]. The transformation from source code to bytecode requires a transformation of code elements into a series of simple expressions. As we are always working with a simplified representation of the code, the risk of masking is introduced between expressions. This could limit the theoretical fault-finding potential of bytecode-based coverage criteria.

To overcome this limitation, we propose a new variant of Multiple Condition Coverage. Our approach, Bytecode-Based Multiple Condition Coverage (Bytecode-MCC), is designed to group related Boolean expressions from the bytecode, reformulate the expressions into a single complex expression, calculate all possible combinations of conditions within the constructed expression, and produce test cases satisfying each combination. Bytecode-MCC should produce test obligations that—when satisfied—are more likely to reveal faults in the program logic than tests providing simple Branch Coverage over the original simplified bytecode.

Bytecode-MCC can be used to measure the power of existing test suites or as a target for automated test generation. To examine both scenarios, we have implemented an algorithm to generate Bytecode-MCC test obligations and measure

coverage in the EvoSuite search-based test generation framework [23]. We have also implemented a fitness function within EvoSuite intended to enable the automated creation of Bytecode-MCC-satisfying test suites.

A preliminary study conducted over 109 faults from Defects4J—an extensive database of real faults extracted from Java projects [**Just14:Defects4J**]—has revealed the following insights:

- Bytecode-MCC is more difficult to achieve than Branch Coverage, and its fitness function does not offer sufficient feedback to guide test generation. Offering additional time to the search process does not guarantee higher levels of obligation satisfaction. This suggests that Bytecode-MCC may be best used as a method of judging test suite quality, rather than as a direct generation target.

- Simultaneously targeting Bytecode-MCC and Branch Coverage improves coverage of Bytecode-MCC and improves the likelihood of fault detection, as the fitness function for Branch Coverage yields more feedback for the search process. It may be possible to identify a set of generation targets that is effective at attaining Bytecode-MCC, even if Bytecode-MCC is not as useful as a direct generation target.

- Results attained for complex logical faults from the "Time" system, where targeting the combination of Bytecode-MCC and Branch Coverage yields an average of 92% Bytecode-MCC coverage, suggest the potential capabilities of Bytecode-MCC if we can improve coverage results. For these faults, the test suites have an average 32.50%-35.00% likelihood of fault detection—well over the overall average.

Ultimately, even if Bytecode-MCC attainment is theoretically able to overcome issues with masking, we cannot test its abilities without first finding ways to improve

coverage. In future work, we will explore various methods of improving Bytecode-MCC coverage through automated generation.

The remainder of this thesis is organized as follows. **Chapter 2** presents background material on adequacy criteria and search-based test generation. **Chapter 3** describes the masking problem and our proposed solution, Bytecode-MCC. **Chapter 4** details our preliminary study. In **Chapter 5**, we present and discuss the results of the study. **Chapter 6** describes threats to validity in our work. **Chapter 7** presents prior work on search-based test generation and the role of coverage in software testing. **Chapter 8** concludes the thesis and summarizes future work.

Chapter 2

Background

In this research, we are interested in test adequacy criteria and how they can be used as part of automated test data generation.

2.1 Adequacy Criteria

Adequacy criteria are important in providing developers with the guidance they need to test efficiently. As we do not know what faults exists before testing, we rely on an approximation of "we found all of the faults". Adequacy criteria are useful for this purpose, as they identify inadequacies in the test suite. For example, if a given test does not reach and execute a statement, it is inadequate for finding faults in that statement.

Each adequacy criterion prescribes a series of test obligations—goals that must be met for testing to be considered "adequate" with respect to that criterion. Often, such criteria are structured around particular program elements and faults associated with those elements, such as statements, branches of control flow, or boolean conditions [51, 63, 81]. When a coverage criterion has been satisfied, the system is considered to be adequately tested with respect to that element. These adequacy criteria have seen widespread use, as they offer objective, measurable checklists [35] and are used as an *exit criteria* in testing critical applications.

2.2 Structural Coverage Criteria

Testing activities are often divided into functional—based on the requirements—and structural—based on the source code—activities. Adequacy criteria have been proposed for both forms of testing [14]. Requirements-based criteria determine how well tests verify the implementation of the software requirements and establish traceability between software requirements and test cases. Structure-based criteria determine how thoroughly the code structure was executed by tests and establish traceability between the code structure and test cases [79].

Requirements coverage alone cannot be considered as a through test of software [49], as:

- The software requirements and design description may not provide a complete and accurate specification of all the behavior represented in the implemented code.
- The requirements of the software may not be written with sufficient coarseness to ensure that all the functionality implemented in the code is tested.
- Moreover, requirements-based testing cannot assure that the executable code contains no unintended functionality.

Therefore, structural coverage analysis is required to ensure the implemented code has been throughly tested[49].

The hypothesis of structural coverage analysis is that it provides a means to determine that the test cases exercise code structure in a manner more likely to expose faults associated with the chosen adequacy criterion. The main purposes of structural coverage analysis are as follows:

- Offers proof that the code structure was exercised to the degree required for applicable software level.

- Provides an assurance that the code structure is free from unintended functions.
- Uncovers the code structure that was not exercised during prior testing.

Numerous structural coverage criteria have been defined with respect to specific syntactic elements of the program [26, 30]. These have been utilized to measure suite adequacy—as an way to asses the quality of existing test suites and whether engineers can stop including new tests. They are, moreover, utilized as a target for automated test generations.

In this study, we are concerned with adequacy criteria defined over boolean conditions. A *decision* is any complex Boolean expression in a program. Decisions can be broken into a series of simple *conditions*—atomic Boolean variables or subexpressions—connected with operators such as `and`, `or`, `xor`, and `not`. In particular, we are focused on Decision Coverage, Branch Coverage, Basic Condition Coverage, and Multiple Conditional Coverage (MCC).

Decision Coverage: This simple criterion requires that all decision statements evaluate to both possible outcomes—`true` and `false`. For example, given the expression (*A or B*), the test suite `(TT), (FF)` attains decision coverage over that expression.

Branch Coverage: The source code of a program can be broken into basic blocks—sets of statements executed sequentially. Branches are decision statements that can decide which basic blocks are executed, such as `if`, `loop`, and `switch` statements. Branch coverage requires that the test suite cover each outcome of all branches. Branch coverage is the most common coverage criterion, with ample tool support and industrial adoption. Improving branch coverage is a common objective in automated test generation [57, 22].

Basic Condition Coverage: Basic condition coverage requires that each condition in each decision take on both outcomes—`true` and `false`—at least once. This allows decision statements to be more thoroughly exercised than through the use of decision

coverage alone. However, this criterion does not require the decision to take on outcomes. For example, for the decision (*A or B*), test suite `(TF),(FT)` satisfies basic condition coverage, but not decision coverage.

Multiple Condition Coverage: Multiple condition coverage (MCC) requires test cases that guarantee all possible combination of condition outcomes within the decision be executed at least once. Given expression (*A or B*), MCC coverage requires test suite `(TF),(TT),(FF),(FT)`. MCC is more expensive to attain than basic condition or decision coverage, but offers more potential fault-detection capability. Note that, in the presence of short-circuit evaluation, infeasible outcomes are not required. In the previous example, short-circuit evaluation would reduce the required test suite to `(FF),(FT),(T-)`.

2.3 Search-Based Software Test Generation

Selection of test input is generally an extremely costly and laborious manual task. However, given a measurable testing goal, input selection can be framed as a *search* for the input that achieves that goal. Automation of input selection can potentially reduce human effort and the time required for testing [7, 39].

Discover of the desired input by exhaustive search is infeasible even for a reasonable sized program, given the near-infinite space of potential options. Given that space, purely random methods are unlikely to discover the exact input needed either [39, 38].

Meta-heuristic search [1, 60] provides a possible solution for test data generation. Given scoring functions denoting *closeness to the attainment of those goals*—called *fitness functions*—optimization algorithms can sample from a large and complex set of options as guided by a chosen strategy (the *metaheuristic*). Metaheuristics are often inspired by natural phenomena. For example, genetic algorithms evolve a group of candidate solutions by filtering out bad "genes" and promoting fit solutions [23].

Due to the non-linear nature of software, resulting from branching control structures, the search space of a real-world program is large and complex. Metaheuristic search—by strategically sampling from that space—can scale effectively to large problems. Such approaches have been applied to a wide variety of testing scenarios [2].

Adequacy criteria are ideal as test generation targets, as such criteria can be straightforwardly translated into the fitness functions used to guide the search [41, 57, 65, 3]. In order to use an optimization algorithm for automation of test input generations, a criterion must be translated into a fitness function that guides the search. For example, if our goal is to execute all branches with in a program, then our fitness function calculation can be formed by picking branches and calculating a test score based on how close a test or sets of tests was to executing the targeted branches. Multiple fitness function formulations exists for branch coverage, but each was designed to give feedback to help the search converge rapidly on a solution. Identifying suitable fitness function representations of adequacy criteria is critical to the success of the search.

Chapter 3

Bytecode-Based Multiple Condition Coverage

In this chapter, we will discuss the central problem motivating this research—*masking* and how it is exacerbated by the translation from source code to bytecode—and formally define our solution, Bytecode-based Multiple Condition Coverage (Bytecode-MCC). Finally, we will describe how we extended the EvoSuite test generation framework to generate test cases satisfying Bytecode-MCC.

3.1 Motivation: Masking and Bytecode

Masking occurs when a condition, within a decision statement, has no effect on the value of the decision as a whole. As an example, consider the trivial program fragments in Figure 3.1. The program fragments have different structures, but are functionally equivalent. Version 1 is defined using intermediate variable `expr_1`, and Version 2 is inlined with no intermediate variables. Given a decision of the form `in_1 or in_2`, the truth value of `in_1` is irrelevant if `in_2` is true, so we state that `in_1` is *masked out*. A condition that is not masked out has *independent effect* for the decision—able to decide the value of the decision as a whole. Masking can have negative consequences on the testing process by preventing the effect of a fault from propagating to a visible failure in the attempted test cases.

Certain coverage criteria are more susceptible to masking than others. Branch and Decision Coverage simply require that the entire expression evaluate to particular outcomes. A fault in the definition of a single condition can be masked, preventing that fault from affecting the result of the entire expression. Similarly, Basic Condition

Version 1: Non-inlined Implementation

```
expr_1 = in_1 or in_2;     //stmt1
out_1 = expr_1 and in_3;   //stmt2
```

Version 2: Inlined Implementation

```
out_1 = (in_1 or in_2) and in_3;
```

Sample Test Sets for (in_1, in_2, in_3):

```
TestSet1 = {(TFF),(FTF),(FFT),(TTT)}
TestSet2 = {(TFT),(FTT),(FFT),(TFF)}
```

Figure 3.1: Behaviorally equivalent implementations with different structures.

Coverage mandates that each condition be varied, but places no constraints on the values of the other conditions. Even if the faulty condition is varied, masking may prevent that condition from affecting the output for that particular test case.

Advanced condition coverage criteria, such as MC/DC and MCC, are able to overcome masking within a single expression. MCC requires that all possible combinations of condition values be attempted, meaning that non-masking test cases must exist—if possible. In practice, MCC is often infeasible for manual test case creation, as a much larger number of tests will be required for satisfaction. However, automation may make it feasible to achieve coverage. MC/DC is an attempt to form a reasonable subset of MCC with similar fault-detection capability. It does so by only requiring that independent impact of each atomic condition be shown on the decision as a whole.

This means that such criteria are sensitive to the structure of the code. Based on the definition of MC/DC, `TestSet1` in Figure 3.1 provides MC/DC over Version 1, but not over Version 2. The test cases with `in_3 = false` contribute towards coverage of the expression `in_1 or in_2` in Version 1 but not over Version 2 since the masking effect of `in_3 = false` is revealed in Version 2.

In contrast, MC/DC over the inlined version requires a test suite to take the

masking effect of `in_3` into consideration as seen in `TestSet2`. This disparity in the MC/DC obligations over the two versions can have significant ramifications with respect to fault finding of test suites. Suppose the code fragment in Figure 3.1 is faulty and the correct expression should have been `in_1 and in_2` (which was erroneously coded as `in_1 or in_2`). `TestSet1` would be incapable of revealing this fault since there would be no change in the observable output, `out_1`. On the other hand, any test set providing coverage of the inlined implementation would be able to reveal this fault.

Previous research has shown that the efficacy of test suite satisfying structural coverage criteria defined over specific program elements such as control-flow branches, conditions or decisions can be highly sensitive to how expressions are written [29, 30, 72, 71]. The non-inlined version has simpler expressions. This means that coverage criteria like MC/DC can be more trivially satisfied, with fewer test cases, than cases where the code is structured into fewer, more complex expressions. The inlined version will have more complex test obligations and will generally require more test cases, but those test cases will generally have more fault revealing power.

Many approaches to automated analysis and test case generation for Java systems operate not on the source code representation of code, but on the bytecode [23, 77]. The bytecode representation of a program is often easier to instrument than the source code—for instance, it can be obtained without the source code being present—and bytecode-based techniques are often more efficient and scalable than source code-based techniques [77]. Existing state-of-the-art techniques compute code coverage and generate test cases by monitoring the execution of the instrumented bytecode to determine which coverage obligations are satisfied [23].

However, the transformation from source code to bytecode requires a simplification of code elements into a non-inlined form similar to that shown in Figure 3.1. Consider the class depicted in Figure 3.2. In this example, the source code is shown on the left

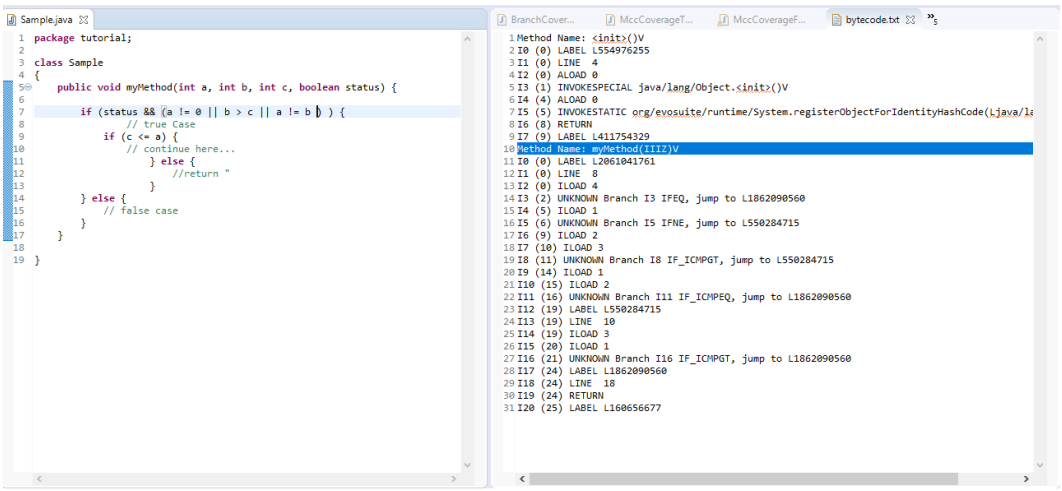

Figure 3.2: A simple Java class containing a complex Boolean expression. Source code is shown on the left, and its equivalent bytecode is shown on the right.

and the bytecode transformation is shown on the right. The complex if-statement on the left is translated into a series of simple non-inlined expressions in the bytecode. This translation is required, given how statements are defined at the bytecode level. However, as we are always working with a non-inlined representation of the code, the risk of masking is introduced between expressions. As all expressions are maximally simplified, a straight-forward implementation of MCC or MC/DC would not be of benefit—they would be equivalent to Branch Coverage over each individual statement.

In the past, researchers have expressed concern over whether code coverage attained over bytecode accurately predicts coverage over the source code [54]. Our own past research indicates that tests that attain MC/DC coverage over a simple, non-inlined version of the code may not cover the complex inlined version [29]. This indicates that existing approaches to bytecode-based test generation may produce tests that do not actually cover the source code, and that are limited with respect to their fault-detection capabilities.

3.2 Bytecode-Based Multiple Condition Coverage

To overcome the limitations imposed by the translation to a simple, non-inlined program structure and to overcome the risk of masking, we propose a new variant of Multiple Condition Coverage. Our approach, Bytecode-Based Multiple Condition Coverage (Bytecode-MCC), is designed to group related Boolean expressions from the bytecode, reformulate the expressions into a single complex expression, calculate all possible combinations of conditions within the constructed expression, and produce test cases satisfying each combination.

In cases where complex inlined source code is translated into simple, non-inlined bytecode, Bytecode-MCC should produce test obligations that—when satisfied—are more likely to reveal faults in the program logic than tests providing simple Branch Coverage over the non-inlined bytecode. Even if the original code was also written in a non-inlined style, this approach can automatically produce a more complex inlined representation that can form the basis of a robust test suite.

Bytecode-MCC can be used to measure the power of existing test suites. A suite that provides high coverage of Bytecode-MCC should have high fault-revealing potential for faults associated with Boolean expressions. Bytecode-MCC can also be used as a target for automated test generation. To examine both scenarios, we have implemented an algorithm to generate test obligations and measure coverage in the EvoSuite test generation framework [23]. We have also implemented a fitness function within EvoSuite intended to enable the automated creation of Bytecode-MCC-satisfying test suites.

The following subsections describe the EvoSuite framework, the test obligation generation algorithm, and the fitness function used for test creation.

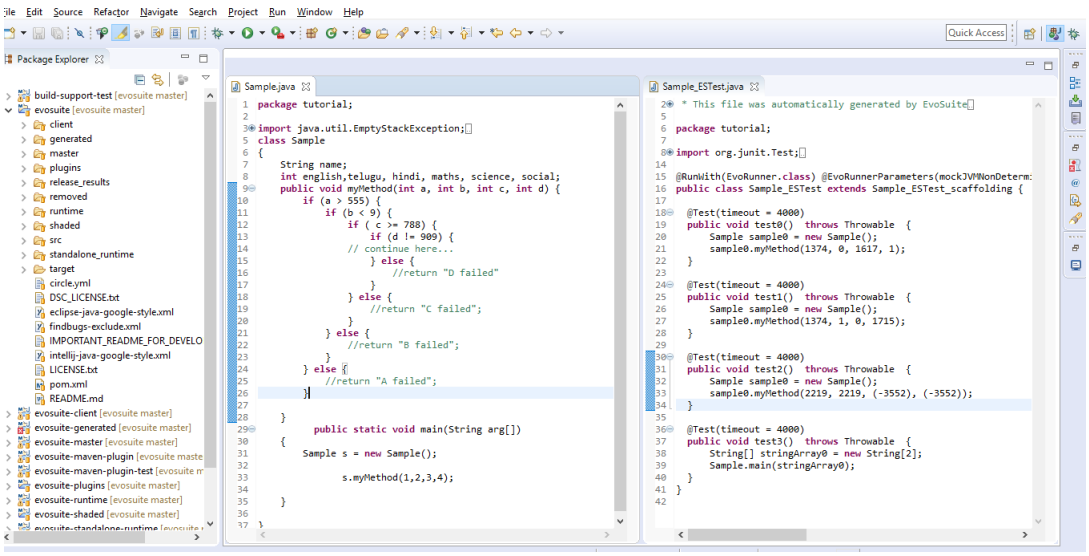

Figure 3.3: EvoSuite producing coverage test suites for Java classes.

3.2.1 The EvoSuite Test Generation Framework

Unit testing is required for object-oriented programs to identify faults and to capture and understand program behavior. EvoSuite is a framework that automatically produces test suites, using search-based test generation [21]. Figure 3.3 depicts a simple Java class and the test suites produced by EvoSuite.

The EvoSuite framework uses a genetic algorithm to evolve test suites over a series of generations, forming a new population by retaining, mutating, and combining the strongest solutions. Genetic algorithms are motivated from the natural process of evolution. An initial population is randomly generated of candidate solutions. Based on their fitness values, parents are selected to create new population members through the crossover and mutation operators. Mutation consists of adding new (random) test cases to a test suite, and modifying existing tests (e.g., by adding, removing, or changing some of the statements). The crossover operator creates new test suites by combining features of parent suites. These operators are applied until a new generation of individuals has been produced, and this then becomes the next generation.

In EvoSuite, each solution is a test suite. The chromosomes of this solution are test cases, made up of one or more method calls to the class-under-test. The fitness function estimates how close a candidate solution is to satisfying a coverage goal. The initial population is generated randomly with fixed number of input values. Chosen representation determines which operators need to be used in the evolution process. Fitness values of the population are improved until either an optimal solution has been found, or some other stopping condition has been met. For example, a maximum time limit or a certain number of fitness evaluations. At the end of the search, the resulting test suite goes through several post processing steps such as minimization which is removal of redundant statements or assertion generations which is addition of JUnit assert statements to check the observed behavior. The search algorithm and the post-processing steps are both applicable regardless of whether the underlying Java class under test is Java SE or JEE code.

EvoSuite is actively maintained and has been successfully applied to a variety of projects [69, 20]. EvoSuite allows users to select one or more fitness functions to guide test case creation, primarily modeled after structural coverage criteria. It also can measure coverage of any of its included criteria, even for test suites not generated originally through EvoSuite.

Traditional approaches to search-based test generation center around coverage of individual test obligations. An obligation is targeted, solutions—test suites or single test cases—are generated, and the fitness score denotes distance to coverage of that particular goal. However, some goals are more difficult to cover than others. Given a fixed search budget, the success of traditional approaches depend significantly on the order in which obligations are considered [4]. Sometimes, a test obligation may be infeasible, i.e., there are no inputs that would cover them. Thus, targeting infeasible obligations will only lead to wastage of effort and testing budget. Further, this approach makes it difficult to predict the size of resulting test suite as test cases

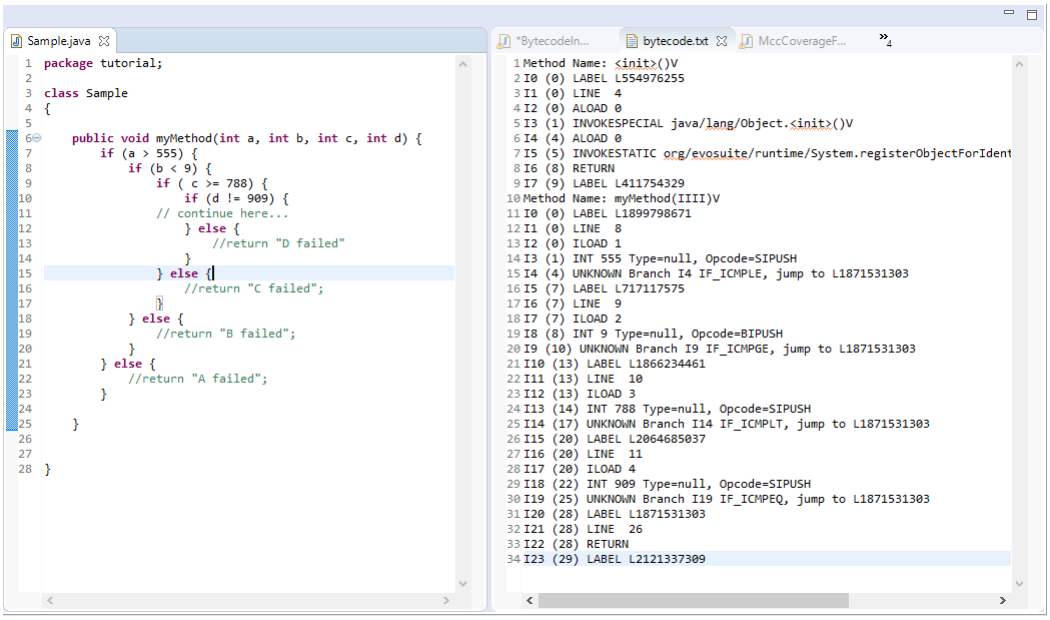

Figure 3.4: A simple Java class. Source code is shown on the left, and its equivalent bytecode is shown on the right.

generated for one goal may be dependent on other goals.

To overcome problems with traditional techniques, EvoSuite's fitness functions are based around the idea of *whole test suite generation* [22, 4]. In this approach, each solution generated by the algorithm is a whole test suite, and the generation algorithm evolves all the test cases in a test suite at same time using feedback from a fitness function focused on attainment of all testing goals simultaneously instead of satisfying each obligation individually. Forking EvoSuite version 1.0.5, we have implemented an algorithm to generate the obligations for Bytecode-MCC and a fitness function that can be used to measure coverage and guide whole test suite generation.

3.2.2 Test Obligation Generation Algorithm

EvoSuite collects all the necessary information for generating tests at bytecode level through Java reflection, meaning that Evosuite does not require the source code of the program. Instrumentation of bytecode is done in Evosuite in order to gather all required information for calculating fitness values for the chosen fitness function.

EvoSuite considers each top level class at a time, during test generation. To calculate the obligations for Bytecode-MCC, we need to formulate a list of test obligations over a series of if operations from bytecode, potentially distributed over a series of subsequent labels. For example, Figure 3.4 depicts a simple Java class. In this example, the source code is shown on the left, and its equivalent bytecode is shown on the right.

In order to formulate the test obligations for Bytecode-MCC, we must:

1. Monitor the bytecode, searching for Boolean expressions.
2. When an expression is detected, begin building a group of related expressions.
3. Add any subsequent Boolean expressions in the same ByteCode label—a basic block of sequentially executed expressions—to the grouping.
4. When a new label is reached, add any new Boolean expressions to that grouping.
5. Stop when a label is reached with no Boolean expressions.
6. Formulate a truth table containing all possible evaluations of the gathered expressions.
7. Translate each row of the truth table into a test obligation.

3.2.3 Monitoring and Grouping of Boolean Expressions

For a given class and method, we inspect the bytecode to gather set of related Boolean expressions. In the bytecode, expressions are grouped into labels. A label indicates the start of a series of sequentially-executed expressions, and is a point that another control-altering expression can jump to.

While monitoring the bytecode, we start a grouping when we detect a Boolean expression. Each Boolean expression in bytecode is represented using a form of `if-statement` where a `true` outcome causes a jump to another label. We add this

```
I0 (0) LABEL L1899798671
I4 (4) UNKNOWN Branch I4 IF_ICMPLE, jump to L1871531303
I5 (7) LABEL L717117575
I9 (10) UNKNOWN Branch I9 IF_ICMPGE, jump to L1871531303
I10 (13) LABEL L1866234461
I14 (17) UNKNOWN Branch I14 IF_ICMPLT, jump to L1871531303
I15 (20) LABEL L2064685037
I19 (25) UNKNOWN Branch I19 IF_ICMPEQ, jump to L1871531303
I20 (28) LABEL L1871531303
I23 (29) LABEL L2121337309
```

Figure 3.5: Grouped labels and Boolean expressions for the class depicted in Figure 3.4.

Table 3.1: Table of Boolean expressions and label information for the grouping identified in Figure 3.5.

Expression	Location	True Jump Location	False Jump Location
I4	L1899798671	L1871531303	L717117575
I9	L717117575	L1871531303	L1866234461
I14	L1866234461	L1871531303	L2064685037
I19	L2064685037	L1871531303	L1871531303

if-statement to our grouping, noting the label that is jumped to if the statement evaluates to `true` and where we resume execution if the statement evaluates to `false`. We then continue to iterate over the code in the current label, if any, adding additional `if-statements` to the table. We continue parsing any labels jumped to by recorded statements for additional `if-statements`, and subsequent labels. Once we reach a label without additional *if-statements*, we stop collecting for that group.

For example, for the sample code in Figure 3.4, we extract the grouping listed in Figure 3.5. Next, we can connect the grouped statements through the ordering they are executed in based on their evaluation.

- We record the current label, where the expression resides.
- We record the label that is jumped to if the expression evaluates to `true`.
- We record where execution resumes if the expression evaluates to `false`. This is either a continuation of the current label, or a new label that is reached immediately after the current expression.

. Table 3.1 lists the extracted information for the statements gathered in Figure 3.5.

Table 3.2: Generated truth table for the grouping identified in Table 3.1.

I4	**I9**	**I14**	**I19**	**Outcome Jump Location**
True	-	-	-	L2089187484
False	True	-	-	L2089187484
False	False	True	-	L2089187484
False	False	False	True	L2089187484
False	False	False	False	L2089187484

3.2.4 Formation of Truth Table and Generating Obligations

The information gathered in the last stage indicates the ordering in which expressions are evaluated, and the outcome once they are evaluated. Using this information, we can form a truth table containing all possible paths through the gathered expressions. Table 3.2 depicts the truth table extracted for the grouping from Table 3.1.

Each row of this truth table corresponds to a concrete test obligation that we impose for the Bytecode-MCC criterion. In order to achieve Bytecode-MCC, we need to cover all of the tows of the table. In this case, the test obligations for the simple class in Figure 3.4 are as follows:

$(I4 \ = \ True)$

$(I4 \ = \ False \wedge I9 \ = \ True)$

$((I4 \ = \ False \wedge I9 \ = \ False) \wedge I14 \ = \ True)$

$(((I4 \ = \ False \wedge I9 \ = \ False) \wedge I14 \ = \ False) \wedge I19 \ = \ True)$

$(((I4 \ = \ False \wedge I9 \ = \ False) \wedge I14 \ = \ False) \wedge I19 \ = \ False)$

3.2.5 Automated Test Generation to Satisfy Bytecode-MCC

A test suite satisfies Branch Coverage, as implemented by the EvoSuite framework, if the produced test suite contains at least one test whose execution evaluates each Boolean expressions in the bytecode to `true`, and at least one whose execution evaluates the expression to `false`.

If measuring coverage, a simple proportion of goals covered to total goals can be reported. However, effective approaches to search-based generation instead require a fitness function that reports not just the percentage of goals covered, but *how close*

the suite is to covering the remaining goals. This feedback allows the search to more effectively converge on a solution that achieves maximum coverage of the chosen criterion.

In the case of Branch Coverage, the fitness function calculates the *branch distance* from the point where the execution path diverged from a targeted expression outcome. If an undesired outcome is reached, the function describes how "close" the targeted predicate was to the desired outcome. The fitness value of a test suite is measured by executing all of its tests while tracking the branch distances $d(b, Suite)$ for each expression.

$$F_{BC}(Suite) = \sum_{b \in B} v(d(b, Suite)) \tag{3.1}$$

Note that $v(...)$ is a normalization of the distance $d(b, Suite)$ between 0-1. The value of $d(b, Suite)$, then, is calculated as follows:

$$d(b, Suite) = \begin{cases} 0 & \text{if the branch is covered,} \\ v(d_{min}(b, Suite)) & \text{if the predicate has been executed at least twice,} \\ 1 & \text{otherwise.} \end{cases} \tag{3.2}$$

The cost function used to attain the distance value follows a standard formulation based on the branch predicate [56]. Note that an expression must be executed at least twice, because EvoSuite must cover the `true` and `false` outcomes of each expression. If the expression is only executed once, the search could jump between each outcome [66].

The implementation of Bytecode-MCC consists of three main classes:

- *MccCoverageSuiteFitness*: This class calculates the Bytecode-MCC coverage for a test suite.

- *MccCoverageTestFitness*: This class is used to determine whether an each individual coverage goal is covered by the test suite.

- *MccCoverageFactory*: This class produces the set of test obligations.

In order to measure coverage of Bytecode-MCC and generate test cases intended to satisfy the produced obligations, we can make use of the same branch distance calculation. To obtain the fitness of a test suite, we calculate the branch distances for each expression (and desired outcome) involved in each obligation. Then, the fitness for an individual obligation is the summation of fitness values of all expressions (and desired outcomes) present in the obligation.

Algorithm 1 Calculating fitness values for a test suite.

```
1: function getFitness(TestChromosome solution, ExecutionResult result)
2:     BranchDistanceMap<int, distance> ← getDistance (result)
3:     for each value ∈ BranchDistanceMap do
4:         fitness ← fitness + value
5:     end for
6:     updateIndividual (this, solution, fitness)
7:     return fitness
8: end function
```

Algorithm 2 Calculating branch distance for each expression within an obligation.

```
1: function getDistance(ExecutionResult result)
2:     for each expression ∈ obligation do
3:         branchDistance ← getBranchDistance(expression, className,
                                methodName)
4:         Map<expression, branchDistance>
5:     end for
6:     return Map
7: end function
```

Algorithms 1 and 2 illustrate this calculation. In EvoSuite, a TestChromosome represents a single solution—a test suite generated by the framework. An ExecutionResult tracks the expressions evaluated by executing a test suite, including the outcome of each Boolean expression.

For each Boolean expression, we can calculate the minimal branch distance achieved by that suite. For each obligation, we calculate the branch distance for each targeted expression and outcome, then score that obligation as the sum of the branch distances for its targeted expression and outcome combinations. As execution comes closer to satisfying the obligation, the fitness should converge to zero. This fitness formulation can be used as a test generation target, or to measure coverage of existing test suites.

3.3 Accessing the Implementation

Our implementation of Bytecode-MCC obligation and test generation is open-source and freely available as a fork of the EvoSuite project. It may be downloaded from `https://github.com/Srujanab09/evosuite`.

Chapter 4

Study

We hypothesize that the simplified nature of bytecode instructions introduces limitations into the efficacy of tests generated to satisfy coverage criteria imposed over the bytecode representation, and that Bytecode-MCC-satisfying tests will be effective at overcoming the masking effect.

We have performed an initial experimental evaluation centered around a typical test generation scenario, in which a user-set amount of time is given to the test generation process. After repeating this process multiple times, we can assess the efficacy of test suites—as judged by the likelihood of detecting a given fault—and the coverage attained during that time period. Using this information, we wish to address the following research questions:

1. Given a typical fixed search budget, is EvoSuite able to satisfy the obligations of Bytecode-MCC?

2. Given a typical fixed search budget, are test suites generated to satisfy Bytecode-MCC more effective at detecting faults than suites satisfying traditional coverage criteria like Branch Coverage?

3. Given a typical fixed search budget, is a multi-objective combination of Bytecode-MCC and Branch Coverage more effective than either alone?

4. Given a typical fixed search budget, is a multi-objective combination of Bytecode-MCC and Branch Coverage able to attain higher Bytecode-MCC coverage than targeting Bytecode-MCC alone?

To address these questions, we have performed the following experiment:

1. **Collected Case Examples:** We have used real faults, from five Java projects, as test generation targets (Section 4.1).
2. **Generated Test Cases:** For each fault, we generated 10 suites targeting Bytecode-MCC, Branch Coverage, and a combination of Bytecode-MCC and Branch Coverage using the fixed version of each class-under-test (CUT). We perform this process with both a two-minute and a ten-minute search budget per CUT (Section 4.2).
3. **Removed Non-Compiling and Flaky Tests:** Any tests that do not compile, or that return inconsistent results, are automatically removed (Section 4.2).
4. **Assessed Effectiveness Against Real Faults:** For each fault and criterion, we measure the proportion of test suites that detect the fault to the number generated (Section 4.3).
5. **Assessed Effectiveness Against Mutations:** For each fault and criterion, we seed mutations—synthetic faults—into each CUT and measure the proportion of test suites that detect each mutant to the number generated (Section 4.3).
6. **Recorded Bytecode-MCC Coverage:** For each suite targeting Bytecode-MCC or the combination of Branch Coverage and Bytecode-MCC, fault, and budget, we measure the Bytecode-MCC Coverage attained over the fixed version of the CUT (Section 4.3).

4.1 Case Examples

Defects4J is an extensible database of real faults extracted from Java projects [43]. Currently, it consists of 395 faults from six projects: Chart (26 faults), Closure (133 faults), Lang (65 faults), Math (106 faults), Time (27 faults), and Mockito (38 faults).

In order to perform this initial evaluation, we have selected a subset of 109 faults: Chart (1), Closure (66), Lang (28), Math (11), and Time (4). These examples were selected because their source code contain either a large number of Boolean expressions (at least 30), complex Boolean expressions (at least three conditions long), or both.

Each fault is required to meet three properties. First, a pair of code versions must exist that differ only by the minimum changes required to address the fault. The "fixed" version must be explicitly labeled as a fix to an issue, and changes imposed by the fix must be to source code, not to other project artifacts such as the build system. Second, the fault must be reproducible—at least one test must pass on the fixed version and fail on the faulty version. Third, the fix must be isolated from unrelated code changes such as refactorings. For each fault, Defects4J provides access to the faulty and fixed versions of the code, developer-written test cases that expose the faults, and a list of classes and lines of code modified by the patch that fixes the fault.

4.2 Test Suite Generation

The EvoSuite framework uses a genetic algorithm to evolve test suites over a series of generations, forming a new population by retaining, mutating, and combining the strongest solutions. It is actively maintained and has been successfully applied to a variety of projects [69].

In this study, we used our variant of EvoSuite version 1.0.5. We generate tests targeting both Bytecode-MCC and Branch Coverage[1]. EvoSuite can also simultaneously target multiple criteria, with fitness evaluated as a single combined score. Therefore, we have also targeted a combination of Bytecode-MCC and Branch Coverage in order

[1]Specifically, the "onlybranch" fitness function, which omits branchless methods. This was chosen as our implementation of Bytecode-MCC also omits branchless methods.

to evaluate whether the combination can achieve either a higher likelihood of fault finding or higher Bytecode-MCC coverage.

Test suites are generated that target the classes reported as relevant to the fault by Defects4J. Tests are generated from the fixed version of the CUT and applied to the faulty version in order to eliminate the oracle problem. In practice, this translates to a regression testing scenario, where tests guard against future issues.

Two search budgets were used—two minutes and ten minutes per class. This allows us to examine whether an increased search budget benefits coverage or fault detection efficacy. These values are typical of other testing experiments [28]. To control experiment cost, we deactivated assertion filtering—all possible regression assertions are included. All other settings were kept at their default values. As results may vary, we performed 10 trials for each fault, criterion, and search budget.

Generation tools may generate flaky (unstable) tests [69]. For example, a test case that makes assertions about the system time will only pass during generation. We automatically remove flaky tests. First, all non-compiling test suites are removed. Then, each remaining test suite is executed on the fixed version five times. If the test results are inconsistent, the test case is removed. This process is repeated until all tests pass five times in a row. On average, less than one percent of the tests are removed from each suite.

4.3 Data Collected

To evaluate the fault-finding effectiveness of the generated test suites, we execute each test suite against the faulty version of each CUT. The **likelihood of fault detection** of each generation portfolio, for each fault, is the proportion of suites that successfully detect the fault to the total number of suites generated for that fault.

We also evaluate fault-finding effectiveness using the **mutation score**. We have used the Major mutation framework to generate a set of mutants—synthetic faults—

for each CUT [44]. This process creates a series of versions of the CUT, where each has a single fault inserted through code transformation into the fixed implementation. The generated test suites are executed against each mutation, and the mutation score is calculated as the ratio of number of mutants detected to the number generated for that CUT.

Just et al. suggest that a significant correlation exists between mutant detection and real fault detection [46]. Therefore, the mutation score is used as a secondary indication of the potential efficacy of the generated suites.

Using EvoSuite's coverage measurement capabilities, we have measured the **Bytecode - MCC coverage** achieved by each suite when executed over both the fixed version of each CUT. We measure this for suites targeting Bytecode-MCC and for suites targeting the combination of Branch Coverage and Bytecode-MCC.

Chapter 5

Results and Discussion

The goal of our preliminary study is to determine whether search-based test generation —as performed by the EvoSuite framework—is able to satisfy the test obligations of Bytecode-MCC within a typical search budget. We also wish to evaluate the fault-detection performance of the suites generated under that budget, regardless of the attained level of coverage.

5.1 Attained Coverage

Table 5.1 lists the average Bytecode-MCC coverage attained by EvoSuite given a two-minute search budget and a ten-minute search budget and using Bytecode-MCC as the optimization target. We also list the Bytecode-MCC coverage attained when simultaneously targeting both Branch Coverage and Bytecode-MCC under the same budgets.

From Table 5.1, we can see that the attained Bytecode-MCC coverage is generally quite low. Overall, only 23.31% of obligations are covered on average under a two-minute budget, and only 25.53% under the ten-minute budget. On a per-system

Table 5.1: Average Bytecode-MCC coverage (%) attained over the CUT, broken down by system and optimization targets.

	Two-Minute Budget		**Ten-Minute Budget**	
System	**MCC**	**MCC/BC**	**MCC**	**MCC/BC**
Overall	23.31	40.47	25.53	43.36
Chart	8.70	35.20	15.40	39.70
Closure	13.52	20.81	16.33	23.97
Lang	31.97	66.07	32.55	68.89
Math	41.00	67.04	44.22	69.22
Time	69.58	92.88	70.68	93.33

Table 5.2: Average Branch Coverage (%) attained over the CUT when Branch Coverage is the optimization target, broken down by system.

System	Two-Minute Budget	Ten-Minute Budget
Overall	39.95	47.67
Chart	33.10	54.41
Closure	13.30	21.36
Lang	81.59	87.99
Math	73.00	77.36
Time	68.50	86.27

basis, the average ranges from 8.70% (Chart) - 69.58% (Time) under the two-minute budget and 15.40% (Chart) - 70.68% (Time) under the ten-minute budget.

We can compare the attained Bytecode-MCC coverage when targeting Bytecode-MCC to the attained Branch Coverage when Branch is the optimization target, as detailed in Table 5.2. Given two minutes for generation, we attain an average of 71.39% higher Branch Coverage. Under a ten-minute budget, we attain an average of 86.72% higher Branch Coverage.

Fundamentally, Bytecode-MCC is a more difficult criterion to satisfy than Branch Coverage. Given the same period of time, we can naturally expect higher attainment of Branch Coverage than Bytecode-MCC coverage. Therefore, "typical" generation time frames may not be enough to attain reasonable levels of Bytecode-MCC coverage.

However, there is also not a substantial improvement from offering five times the search budget—moving from two minutes to ten minutes. On average, there is only a 9.52% improvement in attained Bytecode-MCC coverage, compared to an average overall improvement of 19.32% in Branch Coverage. The low comparative improvement suggests that an increased budget alone may not be enough to overcome the difficulty of satisfying Bytecode-MCC obligations.

To further examine the question of search budget, we chose a small number of classes and tested budgets ranging from one to twenty minutes. The results for Time fault 15 are shown in Figure 5.1. Regardless of budget, the attained Bytecode-MCC coverage for this class range between 61-63%, and do not consistently rise as more budget is allocated. This suggests that merely allocating more budget may not assist

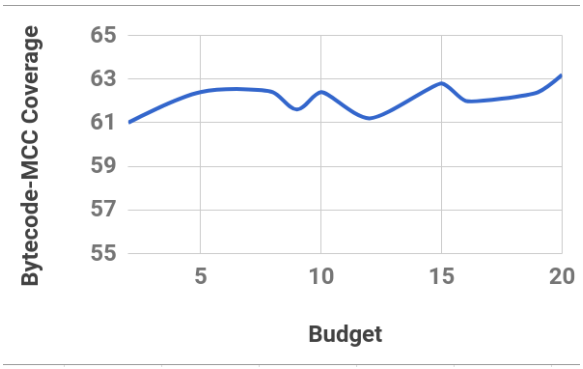

Figure 5.1: Attained Bytecode-MCC coverage as budget (in minutes) increases for Time fault 15.

in covering more goals. Rather, the fitness function does not offer sufficient feedback to drive coverage of the goals.

This idea is further reinforced by examining the Bytecode-MCC coverage results when Branch Coverage and Bytecode-MCC are targeted simultaneously, as listed in Table 5.1 for each system and budget. Overall, targeting both Bytecode-MCC and Branch Coverage simultaneously yields an average 40.47-43.36% Bytecode-MCC coverage. Per-system, coverage ranges from 20.81% (Closure)-92.88% (Time) under a two-minute budget, and 23.97% (Closure) to 93.33% (Time). Overall, targeting Branch and Bytecode-MCC coverage simultaneously yields a 73.62% increase in Bytecode-MCC coverage under a two-minute budget over targeting Bytecode-MCC on its own, and a 69.84% improvement under a ten-minute budget.

Targeting Branch Coverage in addition to Bytecode-MCC offers a series of easier-to-cover intermediate goals that, ultimately, results in improved Bytecode-MCC coverage. Overall, coverage is still lower than desired, but the situation is improved over single-target optimization of Bytecode-MCC.

5.2 Fault and Mutant Detection

Table 5.3 lists the average likelihood of fault detection for suite generated to target Bytecode-MCC, Branch Coverage, and a combination of Bytecode-MCC and Branch

Table 5.3: Average likelihood of fault detection (%) for each generation target and budget, broken down by system and overall.

	Two-Minute Budget			Ten-Minute Budget		
System	**MCC**	**Branch**	**MCC/BC**	**MCC**	**Branch**	**MCC/BC**
Overall	4.27	21.20	16.67	3.47	22.13	19.33
Chart	20.00	100.00	90.00	1.00	100.00	90.00
Closure	0.00	1.33	1.00	0.00	3.67	2.33
Lang	8.28	41.72	28.28	7.24	43.79	32.41
Math	5.46	20.91	16.36	3.64	16.36	19.09
Time	0.00	2.50	32.50	0.00	0.00	35.00

Coverage, divided by system and overall, for each search budget. The likelihood of fault detection is the proportion of generated suites that detect each fault to the total number generated for that fault.

Overall, Branch-targeting suites have a 21.20-22.13% likelihood of detection. This is consistent with previous experiments using this set of faults, and reflects the complex nature of the studied faults [28]. Overall, Bytecode-MCC-targeting tests only have a 4.27% average likelihood of detection under a two-minute budget, and a 3.47% average likelihood of detection under a ten-minute budget—far lower than when Branch Coverage is targeted.

This drop is likely due to the low coverage attained of Bytecode-MCC when it is selected as the sole optimization target. Results again improve when Bytecode-MCC and Branch Coverage are targeted simultaneously. Targeting both yields an overall average likelihood of detection of 16.67% (two-minute budget) and 19.33% (ten-minute budget). Still, this average is lower than when Branch Coverage is targeted alone. Previous research indicates that multi-objective optimization can be more difficult than single-objective optimization [28], and it is likely that the additional burden of satisfying Bytecode-MCC results in lower Branch Coverage as well when both are targeted together.

However, if we examine results on a per-system basis, we can see that Bytecode-MCC satisfaction may have some promise for improvement in fault-detection. For the Time examples, targeting the combination of Branch Coverage and Bytecode-

Table 5.4: Average percent of mutants killed (%) for each generation target and budget, broken down by system and overall.

	Two-Minute Budget			Ten-Minute Budget		
System	**MCC**	**Branch**	**MCC/BC**	**MCC**	**Branch**	**MCC/BC**
Overall	10.26	25.34	23.03	11.73	26.99	25.48
Chart	6.75	41.20	28.80	5.89	41.33	28.10
Closure	4.59	9.95	8.63	5.35	12.53	10.34
Lang	18.96	52.36	46.93	20.70	52.15	52.16
Math	27.99	48.65	49.81	28.12	50.52	49.17
Time	4.86	23.14	24.64	18.42	23.98	24.74

MCC yields an average of over 92% Bytecode-MCC coverage. For this system, the combination also has an average likelihood of detection of 32.50-35.00%—well over the overall average. In this case, targeting the combination makes it possible to detect faults completely missed when targeting Branch Coverage alone.

The Time examples offer complex logical behavior. While the total number of decisions in the source code—34.75—is somewhat lower than the overall average in the studied examples (48.82), the individual statements are more complex with an average of 3.25 conditions per decision (compared to an overall average of 2.29). These are not trivial examples, and the performance when targeting the combination of Branch Coverage and Bytecode-MCC is promising.

The efficacy results when using synthetic faults offer similar conclusions. Table 5.4 lists the average percent of mutants detected overall and for each system, for each generation target and search budget. Overall, suites generated targeting Bytecode-MCC perform worse than those targeting Branch Coverage (10.26 versus 24.34% under a two-minute budget and 11.73 versus 26.99% with a ten-minute budget). Once again, the combination generally yields slightly worse results. However, the combination does show some improvement for the Time examples.

5.3 Discussion

Ultimately, even if Bytecode-MCC attainment is theoretically able to overcome issues with masking, we cannot test its abilities without first finding ways to improve

coverage. The Time examples were the only ones where Bytecode-MCC coverage was reasonably high—particularly with the boost offered by simultaneously targeting Branch Coverage. In order to judge whether there are fault-detection benefits from overcoming masking in bytecode, it is clear that we must find a way to improve coverage.

Some criteria are inherently more difficult to satisfy than others [31, 78, 64]. Regardless of whether tests are hand-created or automatically generated, it is understood that it will be more difficult—and will require more test cases—to satisfy MCC over Branch Coverage. Regardless of the employed test generation technique, it may not be reasonable to expect equal coverage of Branch Coverage and Bytecode-MCC given the same time period.

Still, there may be means of improving coverage and making use of the theoretical fault-detection capabilities of Bytecode-MCC. Moving forward, we intend to explore each of the following topics.

5.3.1 Reformulating the fitness function

An additional complicating factor in search-based test generation comes from the fitness function, and its ability to offer feedback to the search. When attempting to achieve Branch Coverage with search-based generation, the raw percentage of obligations covered could serve as an optimization target—it is a numeric score. However, the branch distance is used instead because it offers further feedback. The branch distance does not simply capture progress, but also suggests whether one solution is closer to covering the remaining obligations over another.

It is possible that a fitness formulation other than the one employed in this work would yield better results. Each Bytecode-MCC obligation is actually a combination of smaller Boolean conditions. Currently, fitness is measured by scoring each condition independently and linearly combining the resulting scores. Progress towards covering

any of the individual conditions will yield an better fitness score. This, in theory, should offer strong feedback. However, there may exist cases where the independent subgoals conflict depending on the selection of input, and a bad choice of input may improve the distance towards one condition while increasing the distance for another condition.

A linear combination of conditions may not be the best mechanism for judging fitness for Bytecode-MCC combinations, and other fitness formulations may yield better results. For example, it may be better to apply a weight to conditions based on the order they must be satisfied in. Alternatively, rather than combining the distances for the conditions into a single score, each obligation could be treated as a set of distances—optimized independently. This would be a more complex approach, but could potentially yield better results in cases where goals conflict. Moving forward, we will consider alternate formulations of the fitness function.

5.3.2 Use Bytecode-MCC in an advisory role instead of as a direct generation target

Some criteria could yield powerful test cases, but lack sufficient feedback mechanisms to drive a search towards high levels of coverage. For example, Exception Coverage rewards test suites that throw additional exceptions. Suites that render the system more likely to throw exceptions can be very powerful tools for detecting faults [27]. However, there is no feedback mechanism that suggests "closeness" to throwing more exceptions.

To give another example, consider mutations. Detection of mutation is often used to judge the fault-detection potential of a test suite. In particular, it forms the basis of two adequacy criteria—Weak and Strong Mutation Coverage [19]. Weak Mutation requires that a statement containing a mutation be reached and executed in such as way that the wrong answer is achieved. Strong Mutation requires, in

addition, that the corruption introduced by the mutation reach the program output, and that the program output be incorrect. The former task is much easier than the latter. More relevantly, the former task offers a reasonable feedback mechanism to drive search-based generation—the concept of branch distance can be applied with slight modification to help guide search-based generation. However, ensuring that the program output is corrupted is a much more difficult task, without a straightforward guidance mechanism.

In cases such as Exception and Strong Mutation Coverage, the criteria have great utility as a means of judging adequacy, and as a stopping criterion. We could target other criteria, but use Bytecode-MCC to judge the final test suites, and we could use coverage of Bytecode-MCC to decide when to cease the generation process.

For example, past research has found that targeting Branch Coverage or a combination of Branch and Exception Coverage tends to yield far higher Exception Coverage than targeting Exception Coverage as a direct generation target. In this case, we could choose alternative optimization targets, then measure the attained Bytecode-MCC. Through experimentation, we plan to explore combinations of criteria and search budgets in order to determine which tend to give the highest Bytecode-MCC coverage. If we can find optimization targets and budgets that yield higher levels of Bytecode-MCC, we will be better able to evaluate the potential of the criterion for overcoming masking and improving the fault-detection potential of test suites.

One topic we are interested in exploring is automated selection of a set of optimization targets for a given task. To select this set of targets, a self-adaptive test case generation algorithm could use reinforcement learning [73] to actively search for the most appropriate set of optimization targets for a chosen CUT and task, adapting to the conditions encountered during the generation process. Given a chosen reward function, reinforcement learning will alternate between exploring new sets of targets and exploiting sets known to yield the largest increase in the reward score [48].

If a tester has a particular goal in mind—for instance, satisfaction of Bytecode-MCC—they could use Bytecode-MCC coverage as the reward function. Although Bytecode-MCC is hard to cover through direct optimization, this approach could be used to suggest a set of other criteria that, optimized together, are better able to satisfy Bytecode-MCC than suites generated to directly target the criterion. We plan to explore the use of this approach to optimize coverage of Bytecode-MCC.

We may wish to also consider other forms of test generation, beyond search-based generation. For example, (dynamic) symbolic execution techniques use sophisticated Boolean solvers to attain input designed to drive program execution towards particular paths [2]. Such approaches suffer from limitations in terms of the type of programs and language features they can handle, and in terms of scalability [24]. However, they can be very effective at producing the input needed to traverse specific paths—which is required for Bytecode-MCC satisfaction. The use of symbolic execution—or approaches that combine search and symbolic execution—may be required to achieve high levels of Bytecode-MCC coverage.

Chapter 6

Threats to Validity

Internal Validity: Because EvoSuite's test generation process is non-deterministic, we have conducted each experiment several times. To control experiment cost, we have only generated ten test suites for each combination of fault, budget, and fitness function. It is possible that larger sample sizes may yield different results. However, we believe that this is a sufficient number to draw stable conclusions.

External Validity: Our study has focused on five systems—a relatively small number. Nevertheless, we believe that such systems are representative of, at minimum, other small to medium-sized open-source Java systems. We believe that we have chosen enough fault examples to gain a basic understanding of Bytecode-MCC, and that our results are generalizable to other, sufficiently similar projects.

In this study, we have only implemented Bytecode-MCC within a single test generation tool—EvoSuite. Results may differ using a different test generation algorithm. However, given the complexity of implementation, we cannot compare different frameworks at this time. However, we believe that EvoSuite is sufficiently powerful to gain a clear understanding of our initial ideas at this time.

Conclusion Validity: We ensure that base assumptions are satisfied when performing statistical analysis. We use non-parametric methods, as distribution characteristics are not generally known a priori, and normality cannot be assumed.

Chapter 7

Related Work

In this chapter we will discuss prior work on software based search generations, role of coverage in test generation, test generation tools and other topics related to this work.

7.1 Search Based Test Generations

Some of the techniques available for generating test cases at unit level are Search Based Software testing (SBST) [56] , Dynamic Symbolic Execution (DSE) [32, 10] and Constraint solving [50, 13, 12]. These approaches are capable of generating tests achieving high code coverage but there exists limitations of each of them due to the dynamic nature of software [58]. For example, existence of unbounded loops and dynamic memory references, i.e., use of pointers. There are also techniques to randomly generate test cases to detect program crashes [11] or to find contract violations [62]. Dynamic symbolic execution [32] were presented to overcome the limitations of random testing [32], but these techniques seem to provide low coverage. Some software based search techniques [56, 53, 70, 59] like symbolic execution and constraint solving requires the tester to select the path for test generation. First search based technique to remove this requirement was goal -oriented approach [52], but it suffered from path problems [58]. In contrast, our tool EvoSuite implements a combination of SBST and DSE [25], to overcome the problems listed above. Implementation of test generation and assertion generations are discussed in details in [21] and the technical details can be found in [18]. One common approach while generating test cases is to consider

one coverage goal at a time, where as EvoSuite generates entire test suite targeting all the coverage goals at once [39].

7.2 Role of Coverage in Test Generation

Numerous studies exist comparing the structural coverage criteria with random testing, with mixed results. A survey provided by Juristo explains the existing work [42]. Few studies by Hutchins [40] explain that branch coverage is better than random testing [32], while the works by Frankl and Weiss [16] disagree that. Work by Namis and Andrews explains the positive correlation between coverage and fault finding effectiveness in a test suite [61]. Although, the recent study by Inozemtseva and Holmes claim that there is only a low to moderate correlation when the number of test cases are controlled for and that stronger forms of coverage may not lead to stronger fault finding results [45]. Studies by Weyuker and Jeng [76], and Chen and Yu [8], indicate that partition is not always better than random testing. Moreover, Hemlet and Taylor determined that partition testing is most ineffective [37]. Although, the work by Gutjahr [36] provides a more desirable case for partition testing.

Recent work by Arcuri et al. [5] shows that random testing is cost effective and is expected to reach high code coverage than expected. Authors also claim that random testing is beneficial than adaptive random testing, when cost is taken into consideration. Most of the recent works on automation of test case generation are centered on how to generate test cases quickly and/or to improve coverage [5]. Work by Gopinath et al. demonstrates a comparison between automated and manually written test suites for statements, block, branch and path coverages and found that statement coverage is effective in finding faults compared to others [33]. Other studies apart from Gopinath's are in concolic execution [68, 32]. Work by Majumdar and Sen [55] on concolic execution merged random testing and symbolic execution, but their evaluation focused only on two case examples and did not involve fault finding

effectiveness.

7.3 Automatic Test Generation Tools

Pex [74] is used to generate test cases for C code by dynamic symbolic execution. It also generates assertions based on return values of methods, but the problem is it cannot support classes that requires complex method sequences. There exists various other tools like Randoop [62] for generating JUNIT test cases. Randoop [62] randomly tests software without guidance in covering complex code structures, reports violations of predefined contracts.The other related tools are TestFul [6] and eToc [75] which uses search-based approach for generating JUnit test suites to maximize structural coverage. However, eToc does not have most advance features in recent times. Therefore, it is a old tool that has not been updated for several years. On the other hand, TestFul is not fully automated when compared to EvoSuite. For example, Testful allows manual editing of XML files for each class under test, which can evaluate just 15 classes whereas Evosuite is fully automated and it can evaluate on thousands of classes [17] as same as Randoop. Randoop [62] allows annotation of the source code to identify observer methods to be used for assertion generation. Another tool named Orstra [80] generates assertions based on observed return values and object states and adds assertions to check future runs against these observations. While such approaches can be used to derive efficient oracles, they do not serve to identify which of these assertions are actually useful, and such techniques are therefore only found in regression testing. In contrast, through the μTest tool, EvoSuite uses mutation testing to select an effective subset of assertions.

7.4 MC/DC and MCC

Despite the importance of advance coverage criteria like MC/DC and MCC [9, 67], there exists very few studies on their effectiveness. Various structural coverage crite-

ria are studied by Yu and Lau found that MC/DC is cost effective when compared to other coverage measures [82]. Evaluation of MC/DC for an example from automotive domain was conducted by Kandl and Kirner found very impressive fault findings result [47]. Works by Dupuy and Leveson in evaluating MC/DC as complement to functional testing explains that the use of MC/DC improves quality of tests [15]. When generating test inputs automatically, it is not sufficient to just maximize structural coverage metrics. It is important to understand how coverage is achieved [30]. To our knowledge, there aren't any comparisons made checking the effectiveness of MC/DC with random tests. Therefore, we cannot claim that the test suite satisfying MC/DC is truly effective. Stronger criteria like MC/DC is also prone to masking over the in-lined program versions as seen above in this paper. Our bytecode based MCC approach is likely to provide a solution for the above problem. More concerning than the negative results regarding the ability of structural coverage to enhance fault finding is the overall lack of consensus one way or the other. Certain coverage metrics are used as though their use guarantees effective testing when, in practice, there is no universal evidence of their utility.

Chapter 8

Conclusions and Future Work

Sophisticated logic-based adequacy criteria such as Multiple Condition Coverage (MCC) are designed to overcome masking at the within-expression level. However, such criteria are sensitive to how programs are written. A Boolean expression written as a single complex statement will yield robust test cases that are more likely to overcome masking and reveal faults than when the expression is written as series of simple statements. The transformation from source code to bytecode requires a transformation of code elements into a series of simple expressions. As we are always working with a simplified representation of the code, the risk of masking is introduced between expressions. This could limit the theoretical fault-finding potential of bytecode-based coverage criteria.

To overcome this limitation, we propose a new variant of Multiple Condition Coverage. Our approach, Bytecode-Based Multiple Condition Coverage (Bytecode-MCC), is designed to group related Boolean expressions from the bytecode, reformulate the expressions into a single complex expression, calculate all possible combinations of conditions within the constructed expression, and produce test cases satisfying each combination. Bytecode-MCC should produce test obligations that—when satisfied—are more likely to reveal faults in the program logic than tests providing simple Branch Coverage over the original simplified bytecode.

Bytecode-MCC can be used to measure the power of existing test suites or as a target for automated test generation. To examine both scenarios, we have implemented an algorithm to generate Bytecode-MCC test obligations and measure

coverage in the EvoSuite search-based test generation framework [23]. We have also implemented a fitness function within EvoSuite intended to enable the automated creation of Bytecode-MCC-satisfying test suites.

A preliminary study conducted over 109 faults from Defects4J—an extensive database of real faults extracted from Java projects [**Just14:Defects4J**]—has revealed the following insights:

- Bytecode-MCC is more difficult to achieve than Branch Coverage, and its fitness function does not offer sufficient feedback to guide test generation. Offering additional time to the search process does not guarantee higher levels of obligation satisfaction. This suggests that Bytecode-MCC may be best used as a method of judging test suite quality, rather than as a direct generation target.

- Simultaneously targeting Bytecode-MCC and Branch Coverage improves coverage of Bytecode-MCC and improves the likelihood of fault detection, as the fitness function for Branch Coverage yields more feedback for the search process. It may be possible to identify a set of generation targets that is effective at attaining Bytecode-MCC, even if Bytecode-MCC is not as useful as a direct generation target.

- Results attained for complex logical faults from the "Time" system, where targeting the combination of Bytecode-MCC and Branch Coverage yields an average of 92% Bytecode-MCC coverage, suggest the potential capabilities of Bytecode-MCC if we can improve coverage results. For these faults, the test suites have an average 32.50%-35.00% likelihood of fault detection—well over the overall average.

Ultimately, even if Bytecode-MCC attainment is theoretically able to overcome issues with masking, we cannot test its abilities without first finding ways to improve

coverage. In future work, we will explore various methods of improving Bytecode-MCC coverage through automated generation. In particular, we plan to:

1. Explore alternative formulations of the fitness function for Bytecode-MCC, such as applying weights based on the order that subobligations must be solved.

2. Examine the use of Bytecode-MCC as a way to judge test suites generated targeting other criteria, as well as its use as a stopping condition for test generation.

3. Investigate the use of reinforcement learning to automatically identify alternative generation targets that will yield higher attainment of Bytecode-MCC than direct targeting of Bytecode-MCC during test generation.

4. Vary the algorithms used to generate Bytecode-MCC-covering test suites, substituting dynamic symbolic execution approaches for search-based approaches.

Bibliography

[1] Shaukat Ali et al. "A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation". In: *IEEE Trans. Softw. Eng.* 36.6 (Nov. 2010), pp. 742–762. ISSN: 0098-5589. DOI: 10.1109/TSE.2009.52. URL: http://dx.doi.org/10.1109/TSE.2009.52.

[2] Saswat Anand et al. "An Orchestrated Survey on Automated Software Test Case Generation". In: *Journal of Systems and Software* 86.8 (Aug. 2013), 1978–2001. DOI: 10.1016/j.jss.2013.02.061.

[3] Andrea Arcuri. "It really does matter how you normalize the branch distance in search-based software testing". In: *Softw. Test., Verif. Reliab.* 23.2 (2013), pp. 119–147. DOI: 10.1002/stvr.457. URL: https://doi.org/10.1002/stvr.457.

[4] Andrea Arcuri and Gordon Fraser. "On the Effectiveness of Whole Test Suite Generation". English. In: *Search-Based Software Engineering*. Ed. by Claire Le Goues and Shin Yoo. Vol. 8636. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 1–15. ISBN: 978-3-319-09939-2. DOI: 10.1007/978-3-319-09940-8_1. URL: http://dx.doi.org/10.1007/978-3-319-09940-8_1.

[5] Andrea Arcuri, Muhammad Zohaib Iqbal, and Lionel Briand. "Formal Analysis of the Effectiveness and Predictability of Random Testing". In: *Proceedings of the 19th International Symposium on Software Testing and Analysis*. ISSTA '10. Trento, Italy: ACM, 2010, pp. 219–230. ISBN: 978-1-60558-823-0. DOI: 10.1145/1831708.1831736. URL: http://doi.acm.org/10.1145/1831708.1831736.

[6] L. Baresi and M. Miraz. "TestFul: automatic unit-test generation for Java classes". In: *2010 ACM/IEEE 32nd International Conference on Software Engineering*. Vol. 2. May 2010, pp. 281–284. DOI: 10.1145/1810295.1810353.

[7] Boris Beizer. *Software Testing Techniques (2Nd Ed.)* New York, NY, USA: Van Nostrand Reinhold Co., 1990. ISBN: 0-442-20672-0.

[8] T. Y. Chen and Y. T. Yu. "On the expected number of failures detected by subdomain testing and random testing". In: *IEEE Transactions on Software Engineering* 22.2 (Feb. 1996), pp. 109–119. ISSN: 0098-5589. DOI: 10.1109/32.485221.

[9] J. J. Chilenski and S. P. Miller. "Applicability of modified condition/decision coverage to software testing". In: *Software Engineering Journal* 9.5 (Sept. 1994), pp. 193–200. ISSN: 0268-6961. DOI: 10.1049/sej.1994.0025.

[10] L. A. Clarke. "A System to Generate Test Data and Symbolically Execute Programs". In: *IEEE Trans. Softw. Eng.* 2.3 (May 1976), pp. 215–222. ISSN: 0098-5589. DOI: 10.1109/TSE.1976.233817. URL: http://dx.doi.org/10.1109/TSE.1976.233817.

[11] Christoph Csallner and Yannis Smaragdakis. "JCrasher: an automatic robustness tester for Java." In: *Softw., Pract. Exper.* 34.11 (Nov. 16, 2004), pp. 1025–1050. URL: http://dblp.uni-trier.de/db/journals/spe/spe34.html#CsallnerS04.

[12] Richard A. DeMillo and A. Jefferson Offutt. "Constraint-Based Automatic Test Data Generation". In: *IEEE Trans. Softw. Eng.* 17.9 (Sept. 1991), pp. 900–910. ISSN: 0098-5589. DOI: 10.1109/32.92910. URL: http://dx.doi.org/10.1109/32.92910.

[13] Richard A. DeMillo and A. Jefferson Offutt. "Experimental Results from an Automatic Test Case Generator". In: *ACM Trans. Softw. Eng. Methodol.* 2.2

(Apr. 1993), pp. 109–127. ISSN: 1049-331X. DOI: 10.1145/151257.151258. URL: http://doi.acm.org/10.1145/151257.151258.

[14] *DO-178B: Software Considerations in Airborne Systems and Equipment Certification*. Radio Technical Commission for Aeronautics (RTCA), 1982.

[15] A. Dupuy and N. Leveson. "An empirical evaluation of the MC/DC coverage criterion on the HETE-2 satellite software". In: *19th DASC. 19th Digital Avionics Systems Conference. Proceedings (Cat. No.00CH37126)*. Vol. 1. 2000, 1B6/1–1B6/7 vol.1. DOI: 10.1109/DASC.2000.886883.

[16] Phyllis G. Frankl and Stewart N. Weiss. "An Experimental Comparison of the Effectiveness of the All-uses and All-edges Adequacy Criteria". In: *Proceedings of the Symposium on Testing, Analysis, and Verification*. TAV4. Victoria, British Columbia, Canada: ACM, 1991, pp. 154–164. ISBN: 0-89791-449-X. DOI: 10.1145/120807.120821. URL: http://doi.acm.org/10.1145/120807.120821.

[17] G. Fraser and A. Arcuri. "Evolutionary Generation of Whole Test Suites". In: *2011 11th International Conference on Quality Software*. July 2011, pp. 31–40. DOI: 10.1109/QSIC.2011.19.

[18] G. Fraser and A. Arcuri. "EvoSuite: On the Challenges of Test Case Generation in the Real World". In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. Mar. 2013, pp. 362–369. DOI: 10.1109/ICST.2013.51.

[19] Gordon Fraser and Andrea Arcuri. "Achieving scalable mutation-based generation of whole test suites". In: *Empirical Software Engineering* 20.3 (2014), pp. 783–812.

[20] Gordon Fraser and Andrea Arcuri. "EvoSuite at the Second Unit Testing Tool Competition". In: *Future Internet Testing*. Ed. by Tanja E.J. Vos, Kiran Lakho-

tia, and Sebastian Bauersfeld. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 95–100.

[21] Gordon Fraser and Andrea Arcuri. "EvoSuite: Automatic Test Suite Generation for Object-oriented Software". In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ESEC/FSE '11. Szeged, Hungary: ACM, 2011, pp. 416–419. ISBN: 978-1-4503-0443-6. DOI: 10.1145/2025113.2025179. URL: http://doi.acm.org/10.1145/2025113.2025179.

[22] Gordon Fraser and Andrea Arcuri. "Whole Test Suite Generation". In: *IEEE Trans. Softw. Eng.* 39.2 (Feb. 2013), pp. 276–291. ISSN: 0098-5589. DOI: 10.1109/TSE.2012.14. URL: http://dx.doi.org/10.1109/TSE.2012.14.

[23] Gordon Fraser et al. "Does Automated White-box Test Generation Really Help Software Testers?" In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ISSTA. Lugano, Switzerland: ACM, 2013, pp. 291–301. ISBN: 978-1-4503-2159-4. DOI: 10.1145/2483760.2483774. URL: http://doi.acm.org/10.1145/2483760.2483774.

[24] J. P. Galeotti, G. Fraser, and A. Arcuri. "Improving search-based test suite generation with dynamic symbolic execution". In: *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. Nov. 2013, pp. 360–369. DOI: 10.1109/ISSRE.2013.6698889.

[25] J. P. Galeotti, G. Fraser, and A. Arcuri. "Improving search-based test suite generation with dynamic symbolic execution". In: *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. Nov. 2013, pp. 360–369. DOI: 10.1109/ISSRE.2013.6698889.

[26] G. Gay. "The Fitness Function for the Job: Search-Based Generation of Test Suites That Detect Real Faults". In: *2017 IEEE International Conference on*

Software Testing, Verification and Validation (ICST). Mar. 2017, pp. 345–355. DOI: 10.1109/ICST.2017.38.

[27] Gregory Gay. "Generating Effective Test Suites by Combining Coverage Criteria". In: *Proceedings of the Symposium on Search-Based Software Engineering*. SSBSE 2017. Paderborn, Germany: Springer Verlag, 2017.

[28] Gregory Gay. "The Fitness Function for the Job: Search-Based Generation of Test Suites that Detect Real Faults". In: *Proceedings of the International Conference on Software Testing*. ICST 2017. Tokyo, Japan: IEEE, 2017.

[29] Gregory Gay et al. "The Effect of Program and Model Structure on the Effectiveness of MC/DC Test Adequacy Coverage". In: *ACM Trans. Softw. Eng. Methodol.* 25.3 (July 2016), 25:1–25:34. ISSN: 1049-331X. DOI: 10.1145/2934672. URL: http://doi.acm.org/10.1145/2934672.

[30] G. Gay et al. "The Risks of Coverage-Directed Test Case Generation". In: *IEEE Transactions on Software Engineering* 41.8 (Aug. 2015), pp. 803–819. ISSN: 0098-5589. DOI: 10.1109/TSE.2015.2421011.

[31] G. Gay et al. "The Risks of Coverage-Directed Test Case Generation". In: *Software Engineering, IEEE Transactions on* PP.99 (2015). ISSN: 0098-5589. DOI: 10.1109/TSE.2015.2421011.

[32] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: Directed Automated Random Testing". In: *SIGPLAN Not.* 40.6 (June 2005), pp. 213–223. ISSN: 0362-1340. DOI: 10.1145/1064978.1065036. URL: http://doi.acm.org/10.1145/1064978.1065036.

[33] Rahul Gopinath, Carlos Jensen, and Alex Groce. "Code Coverage for Suite Evaluation by Developers". In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: ACM, 2014, pp. 72–

82. ISBN: 978-1-4503-2756-5. DOI: 10.1145/2568225.2568278. URL: http://doi.acm.org/10.1145/2568225.2568278.

[34] Alex Groce, Mohammad Amin Alipour, and Rahul Gopinath. "Coverage and Its Discontents". In: *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. Onward!'14. Portland, Oregon, USA: ACM, 2014, pp. 255–268. ISBN: 978-1-4503-3210-1. DOI: 10.1145/2661136.2661157. URL: http://doi.acm.org/10.1145/2661136.2661157.

[35] Alex Groce, Mohammad Amin Alipour, and Rahul Gopinath. "Coverage and Its Discontents". In: *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. Onward! 2014. Portland, Oregon, USA: ACM, 2014, pp. 255–268. ISBN: 978-1-4503-3210-1. DOI: 10.1145/2661136.2661157. URL: http://doi.acm.org/10.1145/2661136.2661157.

[36] Walter J. Gutjahr. "Partition Testing vs. Random Testing: The Influence of Uncertainty". In: *IEEE Trans. Softw. Eng.* 25.5 (Sept. 1999), pp. 661–674. ISSN: 0098-5589. DOI: 10.1109/32.815325. URL: http://dx.doi.org/10.1109/32.815325.

[37] Dick Hamlet and Ross Taylor. "Partition Testing Does Not Inspire Confidence (Program Testing)". In: *IEEE Trans. Softw. Eng.* 16.12 (Dec. 1990), pp. 1402–1411. ISSN: 0098-5589. DOI: 10.1109/32.62448. URL: http://dx.doi.org/10.1109/32.62448.

[38] Mark Harman, Yue Jia, and Yuanyuan Zhang. "Achievements, Open Problems and Challenges for Search Based Software Testing". In: *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on.*

IEEE, Apr. 2015, pp. 1–12. DOI: 10.1109/icst.2015.7102580. URL: http://www0.cs.ucl.ac.uk/staff/mharman/icst15.pdf.

[39] Mark Harman and Phil McMinn. "A Theoretical and Empirical Study of Search-Based Testing: Local, Global, and Hybrid Search". In: *IEEE Trans. Softw. Eng.* 36.2 (Mar. 2010), pp. 226–247. ISSN: 0098-5589. DOI: 10.1109/TSE.2009.71. URL: http://dx.doi.org/10.1109/TSE.2009.71.

[40] Monica Hutchins et al. "Experiments of the Effectiveness of Dataflow- and Controlflow-based Test Adequacy Criteria". In: *Proceedings of the 16th International Conference on Software Engineering*. ICSE '94. Sorrento, Italy: IEEE Computer Society Press, 1994, pp. 191–200. ISBN: 0-8186-5855-X. URL: http://dl.acm.org/citation.cfm?id=257734.257766.

[41] Eunkyoung Jee et al. "A Data Flow-based Structural Testing Technique for FBD Programs". In: *Inf. Softw. Technol.* 51.7 (July 2009), pp. 1131–1139. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2009.01.003. URL: http://dx.doi.org/10.1016/j.infsof.2009.01.003.

[42] Natalia Juristo, Ana M. Moreno, and Sira Vegas. "Reviewing 25 Years of Testing Technique Experiments". In: *Empirical Software Engineering* 9.1 (Mar. 2004), pp. 7–44. ISSN: 1573-7616. DOI: 10.1023/B:EMSE.0000013513.48963.1b. URL: https://doi.org/10.1023/B:EMSE.0000013513.48963.1b.

[43] René Just, Darioush Jalali, and Michael D. Ernst. "Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs". In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ISSTA 2014. San Jose, CA, USA: ACM, 2014, pp. 437–440. ISBN: 978-1-4503-2645-2. DOI: 10.1145/2610384.2628055. URL: http://doi.acm.org/10.1145/2610384.2628055.

[44] Rene Just, Franz Schweiggert, and Gregory M. Kapfhammer. "MAJOR: An Efficient and Extensible Tool for Mutation Analysis in a Java Compiler". In: *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. ASE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 612–615. ISBN: 978-1-4577-1638-6. DOI: 10.1109/ASE.2011.6100138. URL: http://dx.doi.org/10.1109/ASE.2011.6100138.

[45] René Just et al. "Are Mutants a Valid Substitute for Real Faults in Software Testing?" In: *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. Hong Kong, China: ACM, 2014, pp. 654–665. ISBN: 978-1-4503-3056-5. DOI: 10.1145/2635868.2635929. URL: http://doi.acm.org/10.1145/2635868.2635929.

[46] René Just et al. "Are mutants a valid substitute for real faults in software testing?" In: *FSE 2014, Proceedings of the ACM SIGSOFT 22nd Symposium on the Foundations of Software Engineering*. Hong Kong, Nov. 2014, pp. 654–665.

[47] Susanne Kandl and Raimund Kirner. "Error Detection Rate of MC/DC for a Case Study from the Automotive Domain". In: *Proceedings of the 8th IFIP WG 10.2 International Conference on Software Technologies for Embedded and Ubiquitous Systems*. SEUS'10. Waidhofen/Ybbs, Austria: Springer-Verlag, 2010. ISBN: 3-642-16255-X, 978-3-642-16255-8. URL: http://dl.acm.org/citation.cfm?id=1927882.1927902.

[48] Michael N Katehakis and Arthur F Veinott Jr. "The multi-armed bandit problem: decomposition and computation". In: *Mathematics of Operations Research* 12.2 (1987), pp. 262–268.

[49] Hayhurst Kelly J. et al. *A Practical Tutorial on Modified Condition/Decision Coverage*. Tech. rep. 2001.

[50] James C. King. “Symbolic Execution and Program Testing”. In: *Commun. ACM* 19.7 (July 1976), pp. 385–394. ISSN: 0001-0782. DOI: 10.1145/360248.360252. URL: http://doi.acm.org/10.1145/360248.360252.

[51] Edward Kit and Susannah Finzi. *Software Testing in the Real World: Improving the Process*. New York, NY, USA: ACM Press/Addison - Wesley Publishing Co., 1995. ISBN: 0-201-87756-2.

[52] B. Korel. “A dynamic approach of test data generation”. In: *Proceedings. Conference on Software Maintenance 1990*. Nov. 1990, pp. 311–317. DOI: 10.1109/ICSM.1990.131379.

[53] B. Korel. “Automated Software Test Data Generation”. In: *IEEE Trans. Softw. Eng.* 16.8 (Aug. 1990), pp. 870–879. ISSN: 0098-5589. DOI: 10.1109/32.57624. URL: http://dx.doi.org/10.1109/32.57624.

[54] N. Li et al. “Is bytecode instrumentation as good as source code instrumentation: An empirical study with industrial tools (Experience Report)”. In: *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. Nov. 2013, pp. 380–389. DOI: 10.1109/ISSRE.2013.6698891.

[55] Rupak Majumdar and Koushik Sen. “Hybrid Concolic Testing”. In: *Proceedings of the 29th International Conference on Software Engineering*. ICSE ’07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 416–426. ISBN: 0-7695-2828-7. DOI: 10.1109/ICSE.2007.41. URL: http://dx.doi.org/10.1109/ICSE.2007.41.

[56] Phil McMinn. “Search-based Software Test Data Generation: A Survey”. In: *Software Testing, Verification and Reliability* 14 (2004), pp. 105–156.

[57] Phil McMinn. “Search-based Software Test Data Generation: A Survey: Research Articles”. In: *Softw. Test. Verif. Reliab.* 14.2 (June 2004), pp. 105–156.

ISSN: 0960-0833. DOI: 10.1002/stvr.v14:2. URL: http://dx.doi.org/10.1002/stvr.v14:2.

[58] Phil McMinn et al. "The Species Per Path Approach to SearchBased Test Data Generation". In: *Proceedings of the 2006 International Symposium on Software Testing and Analysis*. ISSTA '06. Portland, Maine, USA: ACM, 2006, pp. 13–24. ISBN: 1-59593-263-1. DOI: 10.1145/1146238.1146241. URL: http://doi.acm.org/10.1145/1146238.1146241.

[59] W. Miller and D. L. Spooner. "Automatic Generation of Floating-Point Test Data". In: *IEEE Transactions on Software Engineering* SE-2.3 (Sept. 1976), pp. 223–226. ISSN: 0098-5589. DOI: 10.1109/TSE.1976.233818.

[60] Melanie Mitchell. *An Introduction to Genetic Algorithms*. Cambridge, MA, USA: MIT Press, 1998. ISBN: 0262631857.

[61] Akbar Siami Namin and James H. Andrews. "The Influence of Size and Coverage on Test Suite Effectiveness". In: *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*. ISSTA '09. Chicago, IL, USA: ACM, 2009, pp. 57–68. ISBN: 978-1-60558-338-9. DOI: 10.1145/1572272.1572280. URL: http://doi.acm.org/10.1145/1572272.1572280.

[62] Carlos Pacheco and Michael D. Ernst. "Randoop: Feedback-directed Random Testing for Java". In: *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*. OOPSLA '07. Montreal, Quebec, Canada: ACM, 2007, pp. 815–816. ISBN: 978-1-59593-865-7. DOI: 10.1145/1297846.1297902. URL: http://doi.acm.org/10.1145/1297846.1297902.

[63] William Perry. *Effective Methods for Software Testing, Third Edition*. New York, NY, USA: John Wiley & Sons, Inc., 2006. ISBN: 9780764598371.

[64] M. Pezze and M. Young. *Software Test and Analysis: Process, Principles, and Techniques*. John Wiley and Sons, Oct. 2006.

[65] Sanjai Rayadurgam and Mats P.E. Heimdahl. "Coverage Based Test-Case Generation Using Model Checkers". In: *Proceedings of the 8th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2001)*. IEEE Computer Society, Apr. 2001, pp. 83–91.

[66] Jose Miguel Rojas et al. "Combining Multiple Coverage Criteria in Search-Based Unit Test Generation". English. In: *Search-Based Software Engineering*. Ed. by Marcio Barros and Yvan Labiche. Vol. 9275. Lecture Notes in Computer Science. Springer International Publishing, 2015, pp. 93–108. ISBN: 978-3-319-22182-3. DOI: 10.1007/978-3-319-22183-0_7. URL: http://dx.doi.org/10.1007/978-3-319-22183-0_7.

[67] John Rushby. "New Challenges in Certification for Aircraft Software". In: *Proceedings of the Ninth ACM International Conference on Embedded Software*. EMSOFT '11. Taipei, Taiwan: ACM, 2011, pp. 211–218. ISBN: 978-1-4503-0714-7. DOI: 10.1145/2038642.2038675. URL: http://doi.acm.org/10.1145/2038642.2038675.

[68] Koushik Sen, Darko Marinov, and Gul Agha. "CUTE: A Concolic Unit Testing Engine for C". In: *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ESEC/FSE-13. Lisbon, Portugal: ACM, 2005, pp. 263–272. ISBN: 1-59593-014-0. DOI: 10.1145/1081706.1081750. URL: http://doi.acm.org/10.1145/1081706.1081750.

[69] Sina Shamshiri et al. "Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges". In: *Proceedings of the*

30th IEEE/ACM International Conference on Automated Software Engineering (ASE). ASE 2015. Lincoln, NE, USA: ACM, 2015.

[70] Praveen Ranjan Srivastava and Tai-hoon Kim. *Application of Genetic Algorithm in Software Testing*.

[71] M. Staats, M. W. Whalen, and M. P. E. Heimdahl. "Better testing through oracle selection: (NIER track)". In: *2011 33rd International Conference on Software Engineering (ICSE)*. May 2011, pp. 892–895. DOI: 10.1145/1985793.1985936.

[72] Matt Staats, Gregory Gay, and Mats P. E. Heimdahl. "Automated Oracle Creation Support, or: How I Learned to Stop Worrying About Fault Propagation and Love Mutation Testing". In: *Proceedings of the 34th International Conference on Software Engineering*. ICSE '12. Zurich, Switzerland: IEEE Press, 2012, pp. 870–880. ISBN: 978-1-4673-1067-3. URL: http://dl.acm.org/citation.cfm?id=2337223.2337326.

[73] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. Vol. 1. 1. MIT press Cambridge, 1998.

[74] Nikolai Tillmann and Jonathan De Halleux. "Pex: White Box Test Generation for .NET". In: *Proceedings of the 2Nd International Conference on Tests and Proofs*. TAP'08. Prato, Italy: Springer-Verlag, 2008, pp. 134–153. ISBN: 3-540-79123-X, 978-3-540-79123-2. URL: http://dl.acm.org/citation.cfm?id=1792786.1792798.

[75] Paolo Tonella. "Evolutionary Testing of Classes". In: *SIGSOFT Softw. Eng. Notes* 29.4 (July 2004), pp. 119–128. ISSN: 0163-5948. DOI: 10.1145/1013886.1007528. URL: http://doi.acm.org/10.1145/1013886.1007528.

[76] Elaine J. Weyuker and Bingchiang Jeng. "Analyzing Partition Testing Strategies". In: *IEEE Trans. Softw. Eng.* 17.7 (July 1991), pp. 703–711. ISSN: 0098-5589. DOI: 10.1109/32.83906. URL: http://dx.doi.org/10.1109/32.83906.

[77] Michael W Whalen et al. “A flexible and non-intrusive approach for computing complex structural coverage metrics”. In: *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press. 2015, pp. 506–516.

[78] M. Whalen et al. “Observable Modified Condition/Decision Coverage”. In: *Proceedings of the 2013 Int’l Conf. on Software Engineering*. ACM, May 2013.

[79] Lauren Wiener. *Digital Woes: Why We Should Not Depend on Software*. Addison - Wesley Publishing Company, 1993. ISBN: 9780201626094. URL: https://books.google.com/books?id=LqhQAAAAMAAJ.

[80] Tao Xie. “Augmenting Automatically Generated Unit-test Suites with Regression Oracle Checking”. In: *Proceedings of the 20th European Conference on Object-Oriented Programming*. ECOOP’06. Nantes, France: Springer-Verlag, 2006, pp. 380–403. ISBN: 3-540-35726-2, 978-3-540-35726-1. DOI: 10.1007/11785477_23. URL: http://dx.doi.org/10.1007/11785477_23.

[81] Michal Young and Mauro Pezze. *Software Testing and Analysis: Process, Principles and Techniques*. John Wiley & Sons, 2005. ISBN: 0471455938, 9780471455936.

[82] Yuen Tak Yu and Man Fai Lau. “A Comparison of MC/DC, MUMCUT and Several Other Coverage Criteria for Logical Decisions”. In: *J. Syst. Softw.* 79.5 (May 2006), pp. 577–590. ISSN: 0164-1212. DOI: 10.1016/j.jss.2005.05.030. URL: http://dx.doi.org/10.1016/j.jss.2005.05.030.