Assessing Test Quality

David Schuler

Dissertation zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes
Saarbrücken, 2011

Day of Defense	
Dean	Prof. Holger Hermanns
Head of the Examination Board	Prof. Dr. Reinhard Wilhelm
Members of the Examination Board	Prof. Dr. Andreas Zeller,
	Prof. Dr. Sebastian Hack
	Dr. Valentin Dallmeier

Abstract

When developing tests, one is interested in creating tests of good quality that thoroughly test the program. This work shows how to assess test quality through mutation testing with impact metrics, and through checked coverage.

Although there a different aspects that contribute to a test's quality, the most important factor is its ability to *reveal defects*, because software testing is usually carried out with the aim to detect defects. For this purpose, a test has to provide inputs that execute the defective code under such conditions that it causes an infection. This infection has to propagate and result in a failure, which can be detected by a *check of the test*. In the past, the aspect of test input quality has been extensively studied while the quality of checks has received less attention.

The traditional way of assessing the quality of a test suite's checks is *mutation testing*. Mutation testing seeds artificial defects (mutations) into a program, and checks whether the tests detect them. While this technique effectively assesses the quality of checks, it also has two drawbacks. First, it places a huge demand on computing resources. Second, *equivalent mutants*, which are mutants that are semantically equivalent to the original program, dilute the quality of the results. In this work, we address both of these issues. We present the JAVALANCHE framework that applies several optimizations to enable automated and efficient mutation testing for real-life programs. Furthermore, we address the problem of equivalent mutants by introducing *impact metrics* to detect non-equivalent mutants. Impact metrics compare properties of tests suite runs on the original program with runs on mutated versions, and are based on abstractions over program runs such as dynamic invariants, covered statements, and return values. The intention of these metrics is that mutations that have a graver influence on the program run are more likely to be non-equivalent.

Moreover, we introduce *checked coverage*, an alternative approach to measure the quality of a test suite's checks. Checked coverage determines the parts of the code that were not only executed, but that actually contribute to the results checked by the test suite, by computing dynamic backward slices from all explicit checks of the test suite.

Zusammenfassung

Diese Arbeit stellt dar, wie die Qualität von Software Tests durch mutationsbasiertes Testen in Verbindung mit Auswirkungsmaßen und durch Checked Coverage beurteilt werden kann.

Obwohl unterschiedliche Faktoren die Qualität eines Tests beeinflussen, ist der wichtigste Aspekt die Fähigkeit Fehler aufzudecken. Dazu muss ein Test Eingaben bereitstellen, die den fehlerhaften Teil des Programms so ausführen, dass eine Infektion entsteht, d.h. der Programmzustand fehlerhaft wird. Diese Infektion muss sich so fortpflanzen, dass sie in einem fehlerhaften Ergebnis resultiert, welches dann von einer Test-Prüfung erkannt werden muss.

Die herkömmliche Methode um die Qualität von Test-Prüfungen zu beurteilen ist mutationsbasiertes Testen. Hierbei werden künstliche Fehler (Mutationen) in ein Programm eingebaut und es wird überprüft, ob diese von den Tests erkannt werden. Obwohl diese Technik die Qualität von Test-Prüfungen beurteilen kann, weist sie zwei Nachteile auf. Erstens hat sie einen großen Bedarf an Rechenkapazitäten. Zweitens verwässern äquivalente Mutationen, welche zwar die Syntax eines Programms ändern, jedoch nicht seine Semantik, die Qualität der Ergebnisse. In dieser Arbeit werden Lösungen für beide Probleme aufgezeigt. Wir präsentieren JAVALANCHE, ein System, das effizientes und automatisiertes mutationsbasiertes Testen für realistische Programme ermöglicht. Des Weiteren wird das Problem von äquivalenten Mutationen mittels Auswirkungsmaßen angegangen. Auswirkungsmaße vergleichen Eigenschaften zwischen einem normalen Programmlauf und einem Lauf eines mutierten Programms. Hierbei werden verschiedene Abstraktionen über den Programmlauf benutzt. Die zugrunde liegende Idee ist, dass eine Mutation, die eine große Auswirkung auf den Programmlauf hat, weniger wahrscheinlich äquivalent ist.

Darüber hinaus stellen wir Checked Coverage vor, ein neuartiges Abdeckungsmaß, welches die Qualität von Test-Prüfungen misst. Checked Coverage bestimmt die Teile im Programmcode, die nicht nur ausgeführt, sondern deren Resultate auch von den Tests überprüft werden.

Acknowledgements

First of all, I would like to thank my adviser Andreas Zeller for the guidance and support while working on my PhD. Many thanks also go to Sebastian Hack for being the second examiner of my thesis, and to

During my time at the Software Engineering chair, I have been fortunate to work with great colleagues, of whom many helped me with proofreading parts of this thesis. Thank you Valentin Dallmeier, Gordon Fraser, Florian Groß, Clemens Hammacher, Kim Herzig, Yana Mileva, Jeremias Rößler, and Kevin Streit. While working at this chair, I have been lucky to have great office mates. Thank you Christian Lindig, Rahul Premraj and Thomas Zimmermann. The research that I carried out would not have been possible without the work of our system administrators Kim Herzig, Sascha Just, Sebastian Hafner and Christian Holler who did a great job at maintaining the infrastructure at our chair. For parts of this research, I collaborated with Bernhard Grün and Jaechang Nam, whom I would like to thank. Furthermore, I would like to thank Arnika Marx and Anna Theobald for proofreading.

I am deeply grateful to my friends and family. Especially, I would like to thank Anna, Conny, Heiner, Bastian, Julia, Luzie, Peter and Timo for cooking together and the good time we spent together. Finally, my biggest thanks go to my parents Sigrid and Werner for always supporting me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software fails, and as software is part of our everyday lives, almost everyone has experienced a software failure. Failures are caused by defects in programs, which are accidentally introduced by developers because of the inherent complexity of modern software. It is impossible for a human to account for all possible scenarios that can arise during the execution of a program. Thus, defect free software is an illusion and almost all programs contain defects that lead to more or less severe failures.

Infamous defects include the Ariane 5 explosion in 1996 [20]. A conversion error from a 64-bit floating-point to 16-bit signed integer value caused the Ariane 5 launcher to veer off the planned flight path, which put too much force on the engines so that they were in danger to drop off, and eventually, the self-destruct mechanism was activated. Another severe defect was in the in the control software of the Therac-25 radio therapy machine [51]. Its electron beam has two operation modes, low and high power. A defect in the control software caused the usage of high power beam when the low power beam should have been used. In these cases the high power beam was used without the indispensable safety mechanisms. This caused a massive overdose of radiation for many patients.

A study by the National Institute of Standards and Technology [95] from 2002 quantifies the problem of software defects in terms of costs. It is estimated that software failures cause costs of $59.5 billion annually in the U.S., and that over one third of these costs could be avoided by using a better testing infrastructure.

Software testing techniques are concerned with detecting as many defects as early as possible. To this end, software is tested at different architectural levels and at different stages during the development process. Thereby, one is interested to test the program systematically. *Unit test* operate at the most basic level, and provide inputs for the program under test and compare the results to expected values. *Coverage metrics* impose requirements on the test inputs so that the program is systematically tested. However, they do not consider how well the results of the computations are checked. *Mutation testing*, which inserts artificial defects (mutations) into a program and checks whether tests detect them, assesses the quality of checks and test inputs. Besides these benefits mutation testing also has two major drawbacks. First, it is expensive in terms of computing resources. Second, *equivalent mutants*, which are mutants that do not differ in the program semantics, dilute the quality of its results.

This work makes the following contributions to mutation testing, detecting equivalent mutants, and assessing the quality of a test suite's checks.

JAVALANCHE We introduce the JAVALANCHE mutation testing framework that enables automated and efficient mutation testing for JAVA programs. It is the basis for further research presented in this work and has been developed with the intent to apply mutation testing to real-life programs. JAVALANCHE applies several optimizations adapted from previous mutation testing tools, and it introduces previously unimplemented optimizations.

Equivalent mutants We study and quantify the problem of equivalent mutants for real-life JAVA programs. Equivalent mutants have a different syntax than the original program but are semantically equivalent. As they cannot be detected by a test, they are presented to the developer among regular undetected mutants, and thereby, dilute the quality of mutation testing's results. We study their frequency among undetected mutants on real-life programs and test suites, and measure the time needed to identify them.

Impact metrics To address the problem of equivalent mutants, we introduce several impact metrics that measure the difference between a run of the test suite on the original program and a run of the test suite on a mutated version. The idea behind these approaches is that a mutant with a strong impact on the program run is more likely to be non-equivalent. In this work, we investigate several metrics based on different abstractions that characterize program runs. We show how to compute the impact of a specific mutation by comparing the abstractions over a run of the original program to a run of the mutated version.

Checked coverage We introduce checked coverage, a coverage metric that is designed to assess the quality of the inputs and checks of a test suite. Using dynamic slicing, checked coverage determines the statements actually contributing to the results checked by the test suite, in contrast to statements that are only executed. Thereby, checked coverage focuses on the explicit checks of the test suite, which verify the results of one concrete run and can make detailed assumptions. Thus, they have an important influence on the test quality.

1.1 Thesis Structure

The main contributions of this work are to apply mutation testing to real-life programs, introduce methods that help to identify non-equivalent mutants, and to present a new coverage metric that measures the quality of a test suite's checks. The results of our research are presented further on in the following order:

Chapter 2 gives a short introduction into software testing and program analysis. It describes different testing levels, and focuses on testing at the unit level because the techniques presented in this work operate at this level. The second part introduces static and dynamic program analysis, and explains dynamic analysis approaches relevant for this work in more detail.

Chapters 3 to 5 introduce and define mutation testing as a technique that assesses the quality of a test suite by inserting artificial defects and checking whether the test suite detects them. Chapter 4 presents the JAVALANCHE framework that enables efficient mutation testing for JAVA. With the help of JAVALANCHE, the extend of the equivalent mutant problem for real-life JAVA programs is studied in Chapter 5.

Chapters 6 and 7 present methods to detect non-equivalent mutants via the impact of a mutation. The impact of a mutation is the difference between a run of the test suite on the original version of the program and a run on the mutated version. Chapter 6 introduces invariant impact which is computed by inferring dynamic invariants from the original run and checking for violations in the mutated run. Chapter 7 presents impact measures based on the differences in code coverage and return values between a run of the original program and the mutated version.

Chapter 8 introduces *calibrated mutation testing*. A method which adapts mutant generation schemes to the defect history of a project. A mutant generation scheme produces a set of mutations calibrated to past defects by mining past fixes, extracting properties of them, and mapping these properties to mutations.

Chapter 9 investigates whether there is a correlation between the coverage level of a test suite and its defect detection capability for mutation testing and several coverage metrics, i.e. whether test suites with a higher coverage level are also more likely to detect defects.

Chapter 10 presents checked coverage, an alternative to mutation testing, which assesses the quality of a test suite's checks. This is done by focusing on those code features that actually contribute to the results checked by oracles. For this purpose, the dynamic backward slice from the checks is computed, and only statements that are on the slice are considered as covered.

Chapter 11 concludes with a summary of our results and ideas for future work.

1.2 Publications

This dissertation builds on the following papers (in chronological order):

- **David Schuler**, Valentin Dallmeier, Andreas Zeller. Efficient Mutation Testing by Checking Invariant Violations. In *ISSTA '09: Proceedings of the 18th International Symposium on Software Testing and Analysis*, pages 69–80, New York, NY, USA, 2009. ACM.

- Bernhard J. M. Grün, **David Schuler**, Andreas Zeller. The Impact of Equivalent Mutants. In *Mutation '09: Proceedings of the 3rd International Workshop on Mutation Analysis*, pages 192-199, Los Alamitos, CA, USA, 2010. IEEE Computer Society.

- **David Schuler**, Andreas Zeller. (Un-)Covering Equivalent Mutants. In *ICST '10: Proceedings of the 3rd International Conference on Software Testing, Verification and Validation*, pages 45–54, Los Alamitos, CA, USA, 2010. IEEE Computer Society.

- **David Schuler**, Andreas Zeller. Assessing Oracle Quality with Checked Coverage. In *ICST '11: Proceedings of the 4th International Conference on Software Testing, Verification and Validation*, pages 90–99, Los Alamitos, CA, USA, 2011. IEEE Computer Society.

- Jaechang Nam, **David Schuler**, Andreas Zeller. Calibrated Mutation Testing. In *Mutation '11: Proceedings of the 5th International Workshop on Mutation Analysis*.

Chapter 2

Background

This work presents approaches that measure the quality of software unit tests, and further extensions to these approaches that use dynamic program analysis techniques. The following chapter gives a short introduction to software testing and more details for testing on the unit level as most techniques presented later in this work operate at this testing level. Further on, there will be a focus on program analysis techniques, especially dynamic ones, which are exerted by various approaches that are presented later.

2.1 Software Testing

The goal of the software development process is to produce a piece of software that meets the requirements and contains no defects. In other words, the software should perform as expected under all circumstances. However, meeting this goal is in most cases impossible because not all requirements might be known beforehand and the requirements might be imprecise so that they allow room for interpretation. Furthermore, most software is so complex that one cannot make sure that it contains no defects. In order to gain confidence that the software behaves as expected, it is tested at different stages in the development process. Testing at different stages corresponds to testing at different levels of the program structure.

Acceptance testing addresses the question whether the complete software meets the users' requirements, i.e. if it really does what the users want. It is the last testing step before shipping the product, and is usually carried out by domain experts, who are often the customers themselves. Therefore, the software is tested in the environment of its users which can be executed in the following ways: for software that was developed for a specific customer, a test system is set up at the customer's site that may be tested with real production data. For software that has been developed for the mass market, a beta phase is started in which the software is handed out to selected customers.

System testing is concerned with the question whether the assembled system meets the specifications. The system is tested on the architectural level and the focus is put on the functional part of the software. It is assumed that the subsystems work correctly, and the complete system is tested with the aim of discovering problems that arise from discrepancies between the specification and the implementation, i.e. the specification is not implemented correctly. In contrast to acceptance testing which is carried out by customers, system testing is carried out by the producer of the software, usually by a separate testing team.

Integration testing assesses whether the modules communicate correctly via their interfaces. The aim of this testing phase is to find interaction and compatibility problems. When integrating the modules, different integration techniques are used. First, big-bang integration follows no real strategy because all components are integrated at once. Second, structure-oriented integration combines the modules incrementally by considering the structural dependencies between them. Third, function-oriented integration combines the modules according to specified functional criteria. Integration testing is carried out by the development team, and in contrast to system testing, the code structure is also considered.

Module testing (also called *unit testing*) checks the implementation of single program modules in isolation. The part of a program that is considered as a module can vary. For example, methods and functions can be considered as modules, or classes and files can be considered. In practice, they often cannot be tested in isolation because of the dependencies between modules. Module testing is always concerned with concrete implementations; therefore, it is carried out by the software developer. As the techniques presented in this work are concerned with testing at the unit level, we will present unit tests in more detail.

2.2 Unit Tests

Unit tests are small programs that exercise the *system under test* (SUT) under specified conditions. The execution of the SUT is observed. For example, exceptions that are raised during execution are logged. After the SUT has been exercised the results are checked with the help of *test oracles* and classified as *passing* or *failing*. When a test meets the expectations imposed by the oracle, it is considered to be passing and failing otherwise. A single unit test is referred to as a *test case*.

Definition 1 (Test Case) *A test case executes the system under test under specified conditions and with specified inputs, observes the execution, and checks the results via test oracles.*

Definition 2 (Test Oracle) *A test oracle determines whether the test passes or fails.*

The result of a test is determined by the test oracle and can be influenced by *explicit checks of the test suite*, *explicit checks of the program*, or *implicit checks of the program*. Explicit checks of the test suite usually come in the form of expectations about the computed results. Values returned by the program and the state of the program are compared against expected values. Explicit checks of the program are pre- and post-conditions of the methods and the assertions inside the program, e.g. many methods check their arguments and raise an exception if an illegal argument was passed. Implicit checks are carried out by the runtime system and are not explicitly stated. For example, the JAVA virtual machine implicitly checks whether an object is `null` before calling a method on it and if this is the case a `NullPointerException` is thrown. If all of the checks succeed, the test is considered as *passing*, otherwise, if one of the checks indicates an unexpected behavior, the test is considered as *failing*.

Definition 3 (Test Result) *A test result r for a test t is the result, as determined by the test oracle, of running the test on a version P of the program . The result can either be pass or fail.*

$$t(P) = r,\ r \in \{\text{PASS}, \text{FAIL}\}$$

The aim of a unit test is to test atomic units, which are the smallest possible parts that can be tested in isolation. The benefit of testing small parts comes into play when a test fails. In this case, the location of a bug can easily be pinpointed because only a

small part of the program was covered by the test. However, it is not always possible to test every unit of the program in isolation because of the dependencies between different units.

To test bigger parts of the system (than a single unit), test cases can be combined into test suites. The purpose of those test suites is to group tests for a specific feature or a part of the system. They can be organized in a hierarchical order, i.e. a master test suite might consist of several sub test suites.

Definition 4 (Test Suite) *A test suite is a set of several test cases.*

During the development process, unit tests are executed very frequently. Therefore, the execution of the tests should be, and in most cases is, automated. To this end, several testing frameworks for different programming languages have been developed. Most prominent is the JUNIT framework for JAVA.

These frameworks provide support to automatically run tests, group tests in several test suites. They separate setup code from testing code so that the setup code can be shared between several test cases, and introduce methods to check results, e.g. to check for equality of values or arrays. Furthermore, the frameworks give the developer an overview of the test results, and failing tests are reported together with associated failures and error messages.

The test suite is usually maintained in the same repository as the tested system and is written in the same programming language. It is run by the developer when changes have been made to the system, in order to gain confidence that the changes do not break existing tests. The practice of running tests after a modification to check whether it caused any defects previously experienced and detected by the tests, is called *regression testing*. Of course, tests cannot show the absence of defects. They can only show that the system does not behave unexpected on a limited number of inputs.

Continuous integration tools build the system and run the test suites automatically in regular intervals, and make the results accessible to all developers. Thereby, developers get immediate feedback about the current state of the system.

Using unit tests provides a number of benefits:

Find problems earlier Detecting a defect via unit testing saves costs because detecting a defect at later stages is much more expensive.

Facilitate change By changing the software, there is always the risk of introducing a defect. Therefore, developers might refrain from changing specific parts of the system or making design changes although such changes might be necessary. Thorough unit tests can give the developer confidence that after a change, the system still behaves as expected.

Documentation Tests can provide examples of how to use the program under test. These tests can serve as documentation for developers who are not familiar with the system. Furthermore, the tests are kept up-to-date because they are executed regularly on recent versions of the program. This is often not the case for external code examples.

In order to profit from these benefits, it is important to have a good test suite. However, it is an open question what makes a good test suite. In the following sections, we present methods to assess the quality of unit tests.

2.3 Coverage Metrics

When testing a program, one is interested in thoroughly testing all parts of the program. However, testing with all possible inputs is practically impossible for nontrivial programs. For example, for a method that takes a date encoded as a string, all possible strings might be used as test inputs because tests can also use invalid inputs. In order to direct the testing process, *coverage criteria* can be used. Different coverage criteria impose different *test requirements* that the test should satisfy. [1]

Definition 5 (Coverage Criterion) *A coverage criterion is a rule that imposes requirements on a test suite.*

Definition 6 (Test Requirement) *A test requirement is a specific element or property of the system that the test must cover or satisfy.*

By using coverage metrics that impose test requirements, rules can be stated on how many tests should be created and when to stop testing. This might be the case when all test requirements are satisfied or when a defined fraction of the test requirements is satisfied, i.e. a specific level of coverage is reached.

[1] These definitions of coverage criteria in form of test requirements follow the style of Ammann and Offutt [2].

Definition 7 (Coverage Level) *For a test set T and a set of test requirements TR, the coverage level is the ratio of satisfied requirements relative to all requirements.*

The coverage level can also be used to assess the quality of a test suite with respect to a coverage metric. This also allows comparing different test suites regarding their quality.

Sometimes, coverage criteria impose requirements that cannot be met. These infeasible requirements prevent test sets from satisfying all requirements. Detecting all infeasible requirements can be impossible as this problem can be reduced to the halting problem.

Some coverage criteria are related to each other via subsumption. A coverage criterion is said to subsume another coverage criterion when every test set that satisfies the coverage criterion also satisfies the coverage criterion it subsumes.

Definition 8 (Subsumption) *A coverage criterion C_a subsumes another coverage criterion C_b if and only if every test set that satisfies C_a also satisfies C_b.*

Inspired by different types of defects, different coverage criteria have been introduced. The criteria are designed in such a way that the tests satisfying one criterion do also detect defects of a specific type. Different criteria vary in the effort needed to compute the requirements, to write tests that satisfy them, and in their ability to require tests that reveal defects.

2.3.1 Control Flow Criteria

Criteria that can be defined using the *control flow graph* (CFG) are also called *control flow criteria*. The control flow graph is a directed graph that has a node for every statement or every basic block, i.e. a piece of code that starts with a jump target and ends with a jump and contains no jump or jump target in between. Edges between nodes indicate that there is a direct control flow transition between the nodes. In the following, we will present the most prominent control flow criteria. The intention of these criteria is that defects are revealed by simply exercising the program structures or exercising them in a specific order.

Statement coverage is the most basic coverage metric. It requires that all statements in the program are executed, i.e. a single requirement is that a specific statement

gets executed by at least one test, and the set of test requirements for statement coverage consists of one such requirement for every executable statement in the program. An alternative way of defining statement coverage is to require that every node in the control flow graph is executed by at least one test. Therefore, statement coverage is sometimes also referred to as *node coverage*. Statement coverage can be measured with little overhead, and many tools exist for different programming languages that compute statement coverage. Thus, it is the most widely used coverage metric.

Branch coverage requires that every branch in the program is taken, i.e. every conditional statement is evaluated to `true` and `false`. A single test requirement demands that a specific branch is exercised by at least one test. The set of test requirements is made up of one requirement for every branch in the program. Alternatively, branch coverage can also be defined via the control flow graph by requiring that every edge in the control flow graph is exercised. Thus, branch coverage is sometimes also called *edge coverage*. Branch coverage is a stronger criterion than statement coverage because it implicitly requires that every statement is executed, and in contrast to statement coverage it also requires that branches with no statement are executed, e.g. an empty else block of an if statement. Consequently, branch coverage subsumes statement coverage, which means that every test suite that reaches full branch coverage also reaches full statement coverage.

Path Coverage requires that every possible execution path in the program is exercised. A single test requirement implies that a specific path through the program is followed by at least one test, and the set of test requirements consists of one requirement for each possible path. For the control flow graph this means that every possible path through the CFG has to be taken in order to fulfill path coverage. Path coverage subsumes branch coverage and transitively statement coverage as well. However, the number of paths can become unbounded in the presence of loops. Even if we do not consider loops and restrict ourselves to acyclic paths, there are so many paths that it becomes almost infeasible to exercise them all in non-trivial programs. For example, Ball and Larus [5] reported approximately 10^9 to 10^{11} acyclic paths for the subject programs in the SPEC95 benchmark suite out of which only about 10^4 were exercised by the tests. Therefore, several variants of path coverage exist that limit the number of paths that have to be taken in order to make it feasible to fulfill the criterion. Examples include limiting the length of the paths, only considering loop-free paths, or limiting the number of loop traversals.

2.3.2 Data Flow Criteria

Data flow criteria are based on the idea that a defect is revealed when an erroneous value is used later in the execution of a program. Thus, they impose requirements between the definition and the use of variables. A definition (def) is a point in the program where a value is assigned to a variable, i.e. the value is written to memory. A use is a point in the program where a value of a variable is accessed, i.e. it is read from memory. A definition and a use of the same variable form a *definition-use pair* (also called *def-use* pair or *du pair*) if there exists a *definition-clear path* with respect to the variable between them. A definition-clear path for a variable is a path with no redefinition of the variable.

All definitions coverage requires that every definition of a variable is exercised and also used at least once, i.e. there is at least one test that exercises a du pair that includes the definition.

All du pairs coverage requires that every du pair is exercised by at least one test, i.e. every use for a definition should be covered by a test. Therefore, it is sometimes called *all uses coverage*. It subsumes all definitions coverage, due to the fact that the execution of every du pair also implies that every definition is executed.

All du paths coverage requires that every simple path for every du pair is included by the test suite. A simple path is a path that contains no loops, i.e. no node is contained twice. It is considered to be included with respect to a du pair by another path if it exercises every node of the simple path with no redefinition of the variable associated with the du pair. This definition ensures that all paths between definitions and corresponding uses are exercised, but it excludes all paths that arise from a different execution frequency of loops. A du pair can be exercised by one or more simple paths. Hence, du path coverage subsumes all du pairs coverage.

2.3.3 Logic Coverage Criteria

Logic coverage criteria are based on logical *predicates* of a program. A predicate is a logical expression that evaluates to `true` or `false`. It is composed of subpredicates, boolean variables and literals, function calls, and comparisons between non-boolean variables and literals. The elements of a predicate are connected via *logical operators*. A *clause* is an atomic expression that contains no logical operator. This group of coverage criteria follows the intuition that predicates partition the program's state space. For

this reason, inputs that test different assignments of logical predicates systematically test the state space, which increases the chance to detect defects.

Predicate coverage requires each predicate to be evaluated to `true` and `false`. It is important to note that this criterion is not the same as branch coverage, because it includes boolean expressions used in conditional statements and expressions used in variable assignments. Thus, it subsumes branch coverage.

Basic condition coverage requires each basic condition to be `true` and `false`. That is, every clause is evaluated to `true` and `false`. Thus, this criterion is also called *clause coverage*. It typically requires more tests than predicate coverage, but it does not subsume it because a predicate might not be evaluated to both boolean values although its clauses do so.

Compound condition coverage, which is also known as *multiple condition coverage*, addresses this problem by requiring that every possible assignment of a predicate to be exercised. Therefore, it subsumes basic condition coverage and predicate coverage. This requirement, however, leads to an exponential growth of the required assignments in the number of clauses. In order to avoid this multitude of assignments, some restricted versions have been proposed.

The most prominent one is *modified condition/decision coverage* (MC/DC) which is also called *active clause coverage*. For each clause, it requires an assignment which independently affects the outcome of the whole predicate. In other words, the predicate's result should change whenever the result of the clause changes.

2.3.4 Summary Coverage Criteria

The different criteria were inspired by different types of defects, with the aim that tests fulfilling a criterion do also detect a specific type of defects. However, writing tests that fulfill the criteria requires significant effort. Therefore, the effort needed to fulfill a criterion has to be weighted against the strength of the criterion. In areas where failures have graver consequences, stronger coverage criteria are used, e.g. the RTCA/DO-178B standard [82] for avionics requires modified condition decision/coverage for safety critical applications.

The criteria may also impose infeasible requirements, e.g. when a program has unreachable code it is impossible to cover these statements. These infeasible requirements can also influence the subsumption relations. The relation between infeasible

requirements and subsumption is discussed in more detail by Ammann and Offutt [2], and a more detailed discussion on software testing and coverage criteria can be found in dedicated books on software testing [2, 81].

Satisfying coverage metrics, however, does not guarantee to detect defects, and in general testing cannot show the absence of defects. For example, failures caused by missing code might not be detected by a coverage criterion. Furthermore, the coverage metrics only measure how well the inputs exercise the program. They do not assess how well the results of the program are checked by the oracles. This might result in test cases that trigger a defect but do not detect it, because the input causes a failure which is not checked for.

2.4 Program Analysis

The approaches presented later in this work combine software testing and program analysis. This section gives an introduction to program analysis and presents techniques which are used by approaches presented later in this work in more detail.

Program analysis techniques are concerned with investigating and gaining insight into different aspects of program behavior. Different techniques are used to help developers get a better understanding of complex programs, transform programs (compiler optimizations), and to detect bugs. The different analysis techniques fall into two categories: *static* and *dynamic* techniques, which are presented in the following sections.

2.4.1 Static Program Analysis

Static program analysis techniques reason about a program by analyzing its source code or other static representations, e.g. an intermediate representation used by a compiler. The results of a static analysis technique hold for all possible executions and no specific inputs are needed. Static program analysis emerged from compiler construction where it is needed for optimizations, but it also has further application areas such as model checking and bug detection.

An analysis technique should be *safe* and *precise*. Safe means that each possible behavior of a program is detected by an analysis, i.e. no possible behavior is missed. Precise means that no impossible behavior of a program is detected by an analysis, e.g.

an imprecise analysis might infer that a variable can take values that it never takes in practice. While in most cases one is interested in a safe analysis, there is usually a trade-off between precision and scalability.

For example, compiler optimizations transform a program so that the transformed program meets a performance goal, which is mostly faster execution. For these transformations, it is important that they do not change the behavior of the program, i.e. that they are *semantics preserving*. Program analysis techniques are used to prove that specific transformations do not alter the behavior of the program. In this case, the analysis must be safe because the transformation should preserve the semantics for all possible executions. On the other hand, the analysis does not have to be fully precise because it is tolerable that not all possible optimizations are detected.

Many problems can be solved efficiently by static analysis techniques. For some problems, however, static analysis suffers from *combinatorial explosion*. This means that the number of possible states which have to be considered becomes too large, so that a solution cannot be computed efficiently.

2.4.2 Dynamic Program Analysis

Dynamic program analysis aims to overcome the problem of combinatorial explosion by observing concrete program runs. Typical application areas are bug detection and localization, profiling for speed and memory consumption, or the detection of memory leaks.

In contrast to static analysis, the results of dynamic analysis are only guaranteed to hold for the observed runs. Therefore, its results also depend on the quality of the inputs, i.e. whether they thoroughly exercise the program and trigger a specific behavior of the program. Furthermore, the results of dynamic program analysis are always precise as they correspond to observed behavior. The results, however, are not safe because for all non-trivial programs, it is impossible to exercise them exhaustively. Therefore, a possible behavior can be missed. As a consequence the results are likely to hold, or they are partial results. Likely results are results which have been obtained from a set of runs and might also hold for all runs. Partial results reflect the parts of the obtainable result that could be inferred from the given runs, while other aspects are missed.

Technically, dynamic analysis can be applied in two ways: the program can be *instrumented*, or it can be run in a special *interpreter*. The instrumentation approach

inserts additional code into the program that allows observing it, and then the instrumented program is run in its original environment. The interpreter approach runs the program in a different environment that simulates the original environment but also allows observing different parts of the program. The advantages of the instrumentation approach are that it is less intrusive and generally faster than the interpretive approach. Furthermore, for the interpreter approach some effort is needed to make sure that the original environment is simulated correctly. The advantages of the interpretive approach are that more aspects of the execution can be observed, and it provides more control over the execution. For example, it can observe program internals that cannot be accessed with the instrumentation approach.

For the analysis part there are two options as well: the data can be analyzed at runtime, or offline after the execution. The runtime techniques analyze the data right when it is observed and can present the result during or at the end of the execution. The offline techniques are split into a *tracing* and *processing* part, i.e. first, the observed data is streamed to disk and the analysis, then, takes place in a separate step after the program run is finished.

2.4.3 Execution Trace

Most dynamic analyses capture the program behavior in the form of an *execution trace*. This is a sequence of observed events during a program run where a single event represents a step of the execution.

Definition 9 (Execution Trace) *An execution trace T is a series of execution events $T = \langle e_1, \ldots, e_n \rangle$, where an execution event e is a tuple of attributes.*

Depending on the analysis technique, different events and attributes are of interest. The attributes of an execution event may include the time at which an event took place and information about the code and the data that was involved. The time attribute can either be a timestamp of the absolute time or the time passed since the start of the program. The attributes referring to the code might consist of the statements that were executed, and the thread that executed them. This would allow to track the control flow of an execution. The attributes referring to the data can be the results of intermediate calculations, variables and values that are read or written, values and locations that arise from interaction with the main memory or disk, or from communication with the environment, e.g. input devices or network.

The tracing of all possible events and attributes is limited by the sheer amount of data which can be traced, e.g. a few seconds of executions can produce several GB of data. Therefore, several techniques can be used to reduce the traced data. For example, only parts of the program or a subset of events and attributes are traced, a statistical sampling method can be applied, e.g. only every 100th event is traced, or an abstraction over the data is used, e.g. for every variable only the observed minimum and maximum value is stored.

2.4.4 Program Analysis Techniques

In order to address different problems, several program analysis techniques have been proposed. Depending on their application area, they are concerned with different aspects of the data and control flow of a program. In the following sections, we present program analysis techniques. Mainly dynamic techniques are presented in more detail, because they are used in later parts of this work. Further on, we conclude with a sample of related program analysis techniques.

2.4.5 Coverage Metrics

Coverage metrics can also be seen as a form of dynamic analyses that measure the thoroughness of execution by a test suite. To be able to determine whether a set of tests fulfills a coverage criterion or to compute the coverage level, requires to run and trace the program. Depending on the type of coverage metric, different events and attributes have to be traced. For example, one way to determine the statement coverage is to trace the first execution of each statement. For more complex metrics like all du pairs coverage, every read and write of a variable has to be traced.

Static analysis techniques have been successfully used to generate test inputs that satisfy coverage criteria [8, 99]. Recent approaches also combine static and dynamic analysis for this purpose [91, 96].

2.4.6 Program Slicing

Program slicing was introduced by Weiser [102, 103] as a technique that determines the set of statements that potentially influence the variables used in a given location.

Weiser claims that this technique corresponds to the mental abstractions programmers make when they debug a program.

A static backward slice is computed from a *slicing criterion* which consists of a statement and a subset of the variables used in the program. A slice for a given slicing criterion is computed by transitively following *all data and control dependencies* for the variables from the statement, whereas data and control dependencies are defined as follows:

Definition 10 (Data Dependency) *A statement s is data dependent on a statement t if there is a variable v that is defined (written) in t and referred to (read) in s, and there is at least one execution path from t to s without a redefinition of v.*

Definition 11 (Control Dependency) *A statement s is control dependent on a statement t if t is a conditional statement and the execution of s depends on t.*

Definition 12 (Backward Slice) *A backward slice for a slicing criterion (s,V) is the transitive closure over all data and control dependencies for a set of variables V at statement s.*

Korel and Laski [47] refined this concept and introduced *dynamic slicing*. In contrast to the static slice as proposed by Weiser, the dynamic slice only consists of the statements that *actually influenced* the variables used in a specific occurrence of a statement in a specific program run. This means that only those data and control dependencies are considered which actually emerged during a specific run of the program.

A *dynamic slicing criterion* specifies, in addition to the static slicing criterion, the input to the program and distinguishes between different occurrences of a statement. A *dynamic backward slice* is then computed by transitively following all dynamic dependencies for a set of variables, which are used in a specific occurrence of a statement in a program run that was obtained by using the specified input.

Definition 13 (Dynamic Data Dependency) *A statement s is dynamically data dependent on a statement t for a run r, if there is a variable v that is defined (written) in t and referred to (read) in s during the program run without an intermediate redefinition of v.*

Definition 14 (Dynamic Control Dependency) *A statement s is control dependent on a statement t for a run r, if s is executed during the run and t is a conditional statement that controls the execution of s.*

Definition 15 (Dynamic Backward Slice) *A dynamic backward slice for a dynamic slicing criterion (s_o, V, I) is the transitive closure over all dynamic data and control dependencies for the variables V, in a run with input I, at the o-th occurrence of statement s.*

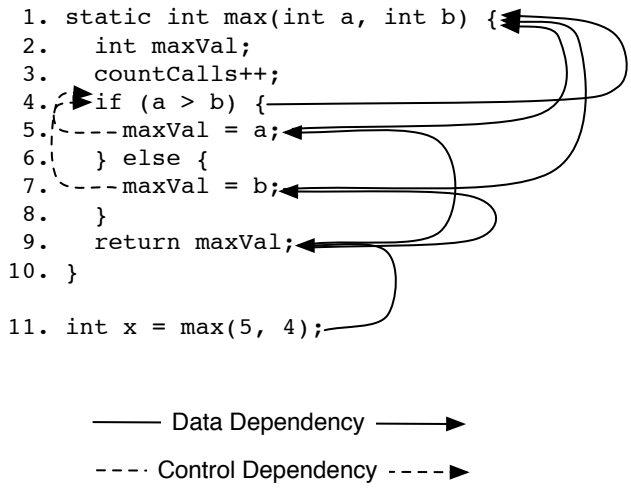

Figure 2.1: Static data and control dependencies for the `max()` method.

In contrast to dynamic slices, static slices take into account all possible dependencies. Therefore, static slices tend to include more statements than dynamic slices.

The code shown in Figure 2.1, for example, displays a method that computes the maximum for two integers, and a statement that calls this method and assigns the result to a variable. Solid arrows show the data dependencies whereas dashed arrows display the control dependencies. The static backward slice from the last statement consists of statements $\{1,4,5,7,9,11\}$. The increment of `countCalls` (statement 3) , for example, is not included in the trace, as there is neither a transitive static control nor

data dependency from the variables used in the last statement to `countCalls`. Note that there are different definitions of which statements should be included in a slice. While we only consider those statements where an actual control or data dependency exists, there are also definitions that require the slice to be executable. Thus, some statements will never be included in a slice because there exist no dependencies. For example, statement 2 in the example, which just declares a variable.

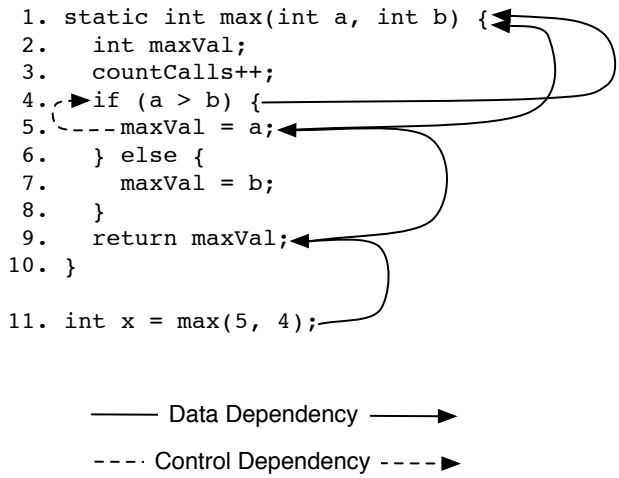

Figure 2.2: Dynamic data and control dependencies for the `max()` method.

Figure 2.2 shows the dynamic data and control dependencies, e.g. the dependencies that actually emerge during the program run. The dynamic slice consists of the statements $\{1,4,5,9,11\}$. In contrast to the static slice, the statement in the else part (statement 7) of the `max()` method is excluded since this code is not exercised by this run, and consequently, these control and data dependencies are not present in the trace.

2.4.7 Invariants

Most program analysis techniques are concerned with the control flow or the data flow. Techniques that are concerned with the actual values used in a program a rare. One abstraction that considers actual values is the concept of *program invariants*. They

characterize a program by specifying properties, mostly about variables, that always hold at specific program points.

Definition 16 (Invariants) *Invariants are properties of the program that are true at specific program points.*

For example, an invariant for the `square()` method given in Figure 2.3 is that the return value is always greater than or equal to zero.

```
public int square(int x) {
  return x * x;
}
```

Figure 2.3: A method that computes the square of an integer.

Invariants help in understanding programs, because they give details that might not be obvious from the source code itself, e.g. that a parameter only takes specific values. Furthermore, invariants can help to avoid introducing bugs if they are explicitly stated. For example, if a parameter is only allowed to take specific values, the use of wrong values can be prevented by an explicitly stated invariant. Although programmers often have invariants in mind when writing programs, they are rarely explicitly stated. Thus, in most cases they have to be inferred from the program.

Static invariants that hold for all possible inputs can be inferred with static analysis techniques. Although static analysis techniques can deduce useful invariants, they are also limited by the state explosion problem. Thus, *dynamic invariants* have been proposed that hold for a specific set of runs, and can be inferred via dynamic analysis by observing one or more program runs. Thereby, the problem of state explosion is avoided. The DAIKON tool by Ernst et al. [23] pioneered the idea of dynamic invariant detection. The dynamic invariants inferred by DAIKON represent *likely invariants*, i.e. some of them correspond to static invariants while others are artifacts of the input data. For example, if the code in Figure 2.3 would only be executed with positive values, two dynamic invariants might be inferred: (1) that the parameter is always positive, which is an artifact of the input, and (2) that the returned value is always greater than or equal to zero, which corresponds to the static invariant.

2.4.8 Related Work

Several static and dynamic analysis techniques have been used in different application areas. Among others, they have been successfully applied to assist in program understanding and debugging, to find and locate defects, to automatically generate test inputs, and to detect memory management problems.

Reps et al. [84] presented path spectra, a dynamic technique that characterizes a run of a program, which aims at helping in testing and debugging. A path spectrum for a program run is the distribution of loop-free paths that were exercised. The Year 2000 problem is presented as an example application for path spectra. Differences in spectra between execution with pre-2000 data and post-2000 data were used to identify computations that are date dependent.

The Delta Debugging algorithm as proposed by Zeller et al. [106] is a dynamic analysis technique that narrows down defect causes. It either creates a minimal test case for a single failing test case (ddmin), or it narrows down the difference between a passing and a failing test case (dd). In contrast to other dynamic analysis techniques, Delta Debugging also generates new executions. To this end, the delta debugging algorithm starts with a failing test case and systematically applies changes (deltas) to it, either by removing parts or copying parts from a passing test case, until a minimal test case is produced, i.e. removing or replacing a single entity from the test case would make the failure disappear.

Another family of dynamic analysis techniques uses *shadow values* to observe the program behavior. A shadow value for every memory location and every register is introduced to keep track of additional information. This analysis is used to detect boundary values, to detect memory leaks, and for taint analysis, which tracks the flow of information from possibly untrusted sources through the program. For example, the Valgrind tool presented by Nethercote and Seward [62] uses shadow values to detect memory management problems for C and C++ programs. Valgrind is a framework for dynamic binary instrumentation, and the Memcheck tool of Valgrind can detect illegal access to memory, uses of uninitialized values, memory leaks, and bad frees of heap blocks. For this purpose, it uses shadow values to keep track of pointers to the memory and whether the memory is allocated and initialized.

Several approaches use static analysis or combined techniques to generate test inputs. The Korat framework for automated testing of JAVA programs introduced by Boyapati et al. [8] automatically generates all test inputs within given bounds. To this

end, Korat translates the preconditions of a method to a JAVA predicate that either returns `true` or `false`. Further on, it systematically generates non-isomorphic inputs that exhaustively explore the bounded input space. Visser et al. [99] presented an approach that uses JAVA Path Finder to generate test inputs to achieve structural coverage by using symbolic execution and model checking. In order to generate test inputs that reach full branch coverage, the program is executed symbolically. *Symbolic execution* uses symbolic values instead of real data, and a path condition is used to keep track of how to reach a program point. To this end, the program is run with symbolic inputs, and during program execution, these symbolic inputs are used instead of concrete values. Thereby, symbolic formulas over the input are obtained. These formulas are used in the path condition in order to get constraints on the input that need to be satisfied to reach a specific program point. The constraints can then be fed to a constraint solver to generate concrete test inputs. Sometimes the constraints to reach a program point get too complex to be solved within reasonable time. Thus, several approaches extend symbolic execution with dynamic analysis [30, 91, 96]. In these approaches, also called concolic testing, the program is executed with real inputs, and constraints are collected that need to be satisfied to take a different branch. Solving these constraints gives a new input that is then used to repeat the process. By using this approach, less complex constraints have to be solved. Thus, even more paths can be reached than through via static analysis alone.

2.5 Summary

In this chapter, we described the different levels at which software can be tested, and we focused on testing at the unit level. Unit tests are carried out via automated test cases, which are small programs that exercise the program and check the result via test oracles. A test can either pass or fail. A failing test indicates that the program behaved in an unexpected way. When developing tests for a program, one is interested in the quality of the tests. Coverage criteria are a method to assess and improve the quality of test suites. They are inspired by different types of defects. The idea is that tests satisfying a specific coverage criterion also detect defects of a specific type. Thus, several coverage criteria have been introduced that impose different requirements on the tests. *Control flow criteria* measure how well the tests exercise control flow structures of the program. *Data flow criteria* are concerned with the relation between definitions and uses of variables, and *logic coverage criteria* with the evaluation of logical predicates. These coverage metrics, however, only measure how well the tests

exercise the program. They do not measure how well the results of the execution are checked.

In the second part of this chapter, program analysis techniques were presented. Program analysis is concerned with investigating and gaining insight about different aspects of the program behavior and comes in the form of static and dynamic techniques. Static program analysis reasons about the program without executing it while dynamic analysis observes concrete program runs. Some application areas of program analysis techniques include defect detection, minimizing defect causes, memory leaks, or generating test inputs. Two techniques, program slicing and dynamic invariants, were presented in more detail because further techniques presented in this work rely on them. Program slicing is a technique that determines the statements that influenced the variables used at a specific point. Dynamic invariants are properties that hold at a specific program point for a set of executions.

Chapter 3

Mutation Testing

Traditional coverage metrics that are used to assess the quality of tests at the unit level only measure how well the test inputs exercise specific structures of the code. They do not gauge the quality of the checks that are used to detect defects.

Mutation testing is a technique to assess and improve the quality of software tests in terms of *coverage and checks*. A program gets mutated by seeding artificial defects (mutations). Such a mutated program is called *mutant*. Then, it is checked whether the test suite detects the mutations. A mutation is considered to be *detected* when at least one test that passes on the original (not mutated) version of a program fails on the mutant. In such cases it is also said that the mutant is *killed*, and an undetected mutant is called a *live mutant*. The defects that are introduced are provided by *mutation operators*. These are well-defined rules that describe how to change elements of a program and aim at mimicking typical errors that programmers make. Usually, a mutation operator can be applied to a program at multiple locations, each leading to a new mutant.

More formally, mutation testing can be defined as modifications to a ground string, i.e. the program that gets mutated.

Definition 17 (Ground String) *A ground string S is a string that belongs to a language L, which is specified by a grammar G.*

A mutation operator then becomes a function that modifies the ground string, and a mutant is the result of applying a mutation operator. [1] A mutant is detected if a test suite contains a test that has a different test result (as introduced in Definition 3) for the mutant than it has for the original version.

Definition 18 (Mutation Operator) *A mutation operator is a function M_{OP} that takes a ground string S from a language L and a location l inside the ground string as input and produces a syntactic variation of the ground string.*

$$S \neq M_{OP}(S,l)$$

Definition 19 (Mutant) *A mutant S_M is the result of a mutation operator's application to a ground string at a specified location l.*

$$S_M = M_{OP}(S,l)$$

Definition 20 (Mutant Detection) *A mutant S_M is detected by a test suite T if it contains a test t that has a different result for the original than for the mutant:*

$$\exists_{t \in T}\ t(S_M) \neq t(S)$$

A mutation operator, however, cannot be applied to every location in the ground string. Some mutation operators, for example, only modify specific syntactic elements, and consequently, they can only be applied if this element is present. Thus, for each mutation operator, there is a fixed set of locations in a program which it can be applied for.

Definition 21 (Mutation Possibilities) *For a mutation operator M_{OP}, and a ground string S that belongs to a language L, the set of mutation possibilities $\mathscr{L}$ is the set of locations for which the mutation operator can be applied so that the result is a string that belongs to the language.*

$$\mathscr{L} = \{l \mid M_{OP}(S,l) \in L\}$$

[1] Although the term mutation refers to the actual change that is made to the program and the term mutant refers to the mutated program, they are sometimes used interchangeably in this work.

The results of the mutation testing process can be summarized in the *mutation score*, which provides a quantitative measure of a test suite's quality. The mutation score is defined as the number of detected mutants divided by the total number of mutants. To measure the mutation score, a set of mutation operators has to be applied exhaustively which is done by applying all mutation operators to all possible locations.

Definition 22 (Set of all Mutants) *For a given ground string S and a set of mutation operators $\mathscr{O} = M^1_{OP}, \ldots M^2_{OP}$, the set of all possible mutants are all valid applications of the mutation operators to the ground string.*

$$\mathscr{M}_{all}(S, \mathscr{O}) = \{M^i_{OP}(S, l) | M^i_{OP} \in \mathscr{O}, l \in \mathscr{L}^i\}$$

Most times only one single mutation is applied to a program at once. However, we can also apply multiple mutations at once. *Higher order mutants* are a combination of multiple mutants. Mutants that are a combination of n ($n \in \mathbb{N}^+$) regular mutants are called n-order mutants.

Definition 23 (Higher Order Mutants) *A n-order mutant S_{M^n} is the result of applying n different mutations to a ground string S.*

$$S_{M^n} = M^n_{OP}(M^{n-1}_{OP}(\ldots M^1_{OP}(S, l^1), \ldots, l^{n-1})l^n)$$

In terms of test requirements, as defined in Section 2.3, *mutation coverage* requires each mutant to be detected.

Definition 24 (Mutation Coverage) *For ground string S and a set of mutation operators $\mathscr{O}$ and the resulting set of all mutants $\mathscr{M}_{all}(S, \mathscr{O})$, mutation coverage requires every mutant to be detected.*

With this definition of mutation coverage, the coverage level as introduced in Definition 7 corresponds to the mutation score.

Furthermore, the undetected mutant can be given as a qualitative feedback to the developer on how to improve a test suite, i.e. by adding tests or modifying existing tests so that previously undetected mutants get detected. This scenario is often assumed by

mutation testing tools. The process of mutation testing and writing tests can be repeated until an *adequacy criterion* is reached, e.g. the mutation score is above a specific value.

It is important to note that mutation testing does not test the software directly; it rather tests the tests of the software and helps to improve them. The assumption is that tests that detect more mutations will also detect more potential defects. Thus, they help to improve the quality of the software. This is expressed in the *competent programmer hypothesis* and the *coupling effect*, which are the two underlying hypotheses of mutation testing and are explained in detail in the following sections.

3.1 Underlying Hypotheses

Mutation testing promises to assist in generating tests that detect real defects. The number of possible defects, however, is unlimited. Mutation testing, on the other hand, only focuses on a subset of errors: those that can be introduced via specific small changes. Therefore, two fundamental assumptions were made when mutation testing was introduced by DeMillo et al. [18]. The first assumption is called the *competent programmer hypothesis* which states that programmers create programs that deviate from the correct one only by small errors. The second assumption is called the *coupling effect* which states that complex errors are coupled to simple errors.

3.1.1 Competent Programmer Hypothesis

The competent programmer hypothesis claims that programmers have a sufficient idea of a correct program. They tend to develop programs that are close to the correct one, which means that their programs only deviate from the correct version by small changes. Mutations, which are in fact small changes, try to simulate those defects. Although it is obvious that not all defects can be fixed via small changes, a huge fraction of the defects can be. In a study on the bug fixes of seven JAVA open-source projects, Pan et al. [78] showed that 45.7% to 63.3% of the fixes can be mapped to simple patterns. The relation between defects that require more complex fixes and mutations is expressed through the coupling effect.

3.1.2 Coupling Effect

The coupling effect puts simple defects in relation to complex ones and was originally stated by DeMillo et al. [18] as follows:

> *Test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors.*

As simple defects are simulated by mutations, the coupling effect implies that tests that detect simple mutations do also detect more complex defects.

Offutt [66, 67] later provided evidence for the coupling effect with a study on higher order mutants. He investigated whether tests that detect all simple mutants do also detect higher order mutants. The results indicated that tests that detect simple mutants also detect 99% of the second and third order mutants.

3.2 Costs of Mutation Testing

Two factors detained the practical application of mutation testing. The first factor is the cost needed to produce the mutants and to execute the tests for them. Depending on the number of mutants and the time needed to run the test suite, mutation testing can easily take several hours, as the test suite has to be executed for every mutant. For example, to execute $10,000$ mutants for a program with a test suite that needs 10 seconds to execute, more than one day of computing time is needed. Furthermore, the number of mutations is influenced by the choice of the mutation operators. The more mutation operators are used the more mutations are produced. This number grows linear in the number of statements and linear in the number of mutation operators. For example, applying 108 mutation operators for the tcas program of the Siemens suite [19], which contains 137 lines of code, produces 4,937 mutations [107]. Therefore, several optimization techniques have been developed to speed up mutation testing (see Section 3.3).

The second factor is the cost introduced by *equivalent mutants*. Equivalent mutants differ in the syntax of a program but do not change its semantics—that is, the mutated program produces the same output as the original. These equivalent mutants cannot be detected by a test because there is no output that would allow a test to distinguish them.

Definition 25 (Equivalent Mutant) *An equivalent mutant S_M^E of a ground string S is a mutant that does not change the observable behavior of the program, so that all possible tests produce the same result on the original and mutated version.*

$$\forall_t \; t(S) = t(S_M^E)$$

Equivalent mutants impose a burden on programmers who try to interpret the results from mutation testing as they first have to decide on the equivalence of a mutant before accomplishing their intended task, which is improving the test suite. Furthermore, it was reported that programmers judge mutant equivalence correctly only in about 80% [71] of the cases. Therefore, equivalent mutations degrade the usefulness of the results of mutation testing.

3.3 Optimizations

As explained earlier, it takes much time to execute all mutations. Thus, several techniques have been proposed to speed up mutation testing. Offutt and Untch [73] divide these techniques into three different categories: *do fewer*, *do smarter*, and *do faster*. The do fewer approaches aim to reduce the number of mutants that have to be executed. The do smarter approaches aim to reduce the computational effort by retaining state information between runs or by avoiding specific executions. The do faster approaches aim to generate and execute the mutants as fast as possible. In the following sections, we will present several optimizations from these categories. *Mutation reduction techniques* such as *mutant sampling* and *selective mutation* are do fewer techniques. *Weak mutation* is an example for a do smarter technique. *Mutant schema generation*, the use of *coverage data*, and *parallelization* fall in the group of the do faster techniques

3.3.1 Mutation Reduction Techniques

As discussed above, one cost factor of mutation testing is the huge number of mutations that can be applied. Thus, several techniques have been developed that try to reduce the number of mutants that need to be executed. The goal of these approaches is to approximate the results (e.g. the mutation score) of applying all possible mutations as precise as possible while trying to use as few mutants as possible. There are two major reduction techniques: *mutant sampling* that randomly selects a fraction of all mutations, and *selective mutation* that uses a subset of the mutation operators.

3.3.2 Mutant Sampling

Mutant Sampling reduces the number of mutants by randomly choosing a subset out of all mutants. The idea was first proposed by Acree and Budd [73, 44], and later Wong and Mathur studied this technique by comparing random selections of different size. Their results suggest that test suites that are created to detect a random selection of 10% of the mutants, also detect around 97% of all mutants.

Zhang et al. [107] proposed two round random selection, a technique that first randomly selects the mutation operator and then a concrete mutant for this operator. With this technique it is equally likely to choose mutants produced by an operator that has only a few than to choose mutants produced by an operator that has many. Their results suggest that this selection technique is comparable to normal random selection, but rarely brings any benefits.

3.3.3 Selective Mutation

Selective mutation tries to reduce the cost of mutation testing by reducing the number of mutations applied to a subject. Thereby, the reduced set of mutants should lead to results similar to applying all mutants. The number of mutations is usually reduced by focusing on a few mutation operators.

Offutt et al. [69] showed in an empirical study on 10 Fortran programs that 5 out of 22 mutation operators are sufficient—test suites that detect all selected mutants also detect 99.5% of all mutations. Barbosa et al. [6] did a similar study for C programs. They proposed 6 guidelines for selecting the mutation operators. In an experiment on 27 programs, they determined 10 out of 39 operators that produce a sufficient set with a precision of 99.6%.

A slightly different approach was taken by Namin et al. [61]. They tried to produce a smaller set of mutants that could be used to approximate the mutation score for all mutants. Using statistical methods, they came up with a linear model that generates less than 8% of all possible mutants, but accurately predicts the effectiveness of the test suite for the full set of mutants.

In a recent study, Zhang et al. [107] compared these 3 different operator-based selection techniques to random selection techniques. They showed that all techniques performed comparably well, and that random selection can outperform the best more sophisticated operator-based selection schemes.

3.3.4 Weak Mutation

Weak mutation testing, as proposed by Howden [40], assesses the effect of a mutation by examining the state after its execution. If the state is different, then the mutant is considered to be detected. Weak mutation testing thus checks whether the tests could *possibly* detect a mutation because a state change might result in a different result while no change indicates that the result of the mutated run will be the same as for the original run. In contrast to regular strong mutation testing, it does not matter whether the tests actually pass or not.

Weak mutation can save execution costs because it is not necessary to continue a run once a mutation is reached and a difference in the state was detected. If no difference is detected, the execution has to be continued because the mutated statement might be executed again. Additional costs, however, are introduced by capturing the state right after the mutation is executed, which can happen many times during one execution of the test suite.

It has to be noticed that weak mutation produces different results than mutation testing, and it only assesses the quality of the test inputs, i.e. whether they are chosen in such a way that the mutation is triggered and causes a state change. Furthermore, the benefits of this technique are questionable. It has not been shown that by limiting the execution the cost savings outweigh the cost for capturing the state changes.

3.3.5 Mutation Schemata

The traditional way of applying several mutations to a program is to mutate the source code of the program and produce a new version for each mutation. Then, each mutated version is compiled, and the tests are executed for each compiled version. In order to speed up this process, Untch et al. [98] proposed to use *mutant schema generation* (MSG). Out of the original program, the MSG approach generates a *metamutant*. For every statement that can be mutated, the metamutant contains a metamutation which can be instantiated at runtime so that it represents a specific mutation. This approach reduces the costs for compiling the program as *only one version* needs to be compiled instead of one version for each mutation. Furthermore, the setup cost for running the tests can be reduced, as mutations can be switched on and off at runtime, and the setup of the test environment only occurs once instead of once per mutant.

3.3.6 Coverage Data

In order to detect a mutation, a test has to exercise the mutated statement and cause the statement to have a direct or indirect influence on the state of the program. This change to the state must propagate to the end of the execution. If one of these conditions cannot be fulfilled, the test cannot detect the mutation. Tests that do not exercise the mutation cannot detect it. Thus, they do not need to be considered. By collecting information on which tests execute which mutations, only a fraction of the whole test suite needs to be executed, which results in a reduction of the execution costs. Although this optimization can be very effective, it is rarely implemented or mentioned in the literature on the topic. CERTITUDE [35] is the only other tool we are aware of that considers coverage. It is a mutation testing tool for integrated circuit designs and checks whether a mutant is activated, i.e. it is executed by at least one test. In the actual testing phase, only the activated mutations are considered. However, the coverage data per test is not used by CERTITUDE.

3.3.7 Parallelization

The execution of different mutants is independent of each other, and mutation testing has little communication needs, as only the results for each run of the test suite on the mutated version have to be reported. Thus, mutation testing lends very well to parallelization approaches. It was shown that by using multiple processors a speed-up that is almost linear in the number of cores can be reached [72].

3.4 Related Work

The idea of mutation testing was first mentioned in a term paper by Richard Lipton in 1971 [73]. However, it took until 1978 before the first major work on mutation testing was published by DeMillo et al. [18]. Their paper also introduced the coupling effect and the competent programmer hypothesis. Since then, many different mutation testing frameworks for different programming languages have been introduced. For example, PIMS [12] for FORTRAN code, MOTHRA [74] for FORTRAN 77, PROTEUM [16] for C, and μJava [55] for JAVA.

The idea of mutation testing was also applied to different areas than source code. Budd and Gopal [11] applied mutation testing to specifications in predicate calculus

form. They checked whether tests given as pairs of an input and an output detect mutated specification predicates. Spafford [94] introduced the idea of environment mutation which uses mutations to simulate errors that can be caused by the environment of a program. For example, errors caused by memory limitations or overflow errors caused by numeric limitations.

Further information about mutation testing frameworks and uses of mutation testing can be found in the survey papers of Offutt and Untch [73] and Jia and Harman [44].

3.5 Summary

In this chapter, we introduced and defined the concept of mutation testing as a method to measure the quality of a test suite and to aid in improving its quality. During mutation testing, artificial defects (mutations) are inserted into the program, and it is checked whether the test suite detects them, i.e. whether a test that passes on the original program fails on the mutated version. Based on the competent programmer hypothesis and the coupling effect, it is claimed that mutations are similar to real defects and that a test suite that detects mutations will also detect real defects. In contrast to traditional coverage metrics, mutation testing also measures the quality of the checks which come as explicit checks of the program and its test suite, and implicit checks of the runtime system.

Two obstacles prevented the widespread use of mutation testing. First, the execution costs, which arise from the fact that the tests have to be executed for every mutant, and that there is a huge number of mutants for a program. Thus, several optimization techniques have been proposed. In the following Chapter 4, we will present a mutation testing framework that implements many of these optimizations in order to make mutation testing feasible for larger programs. The second factor that prevented a widespread use of mutation testing is the occurrence of equivalent mutants. Equivalent mutants are mutants that cannot be detected by tests because they do not change the observable behavior of the program, although they change the program's syntax. In the following chapters, we will quantify the problem of equivalent mutants (Chapter 5) and introduce techniques to distinguish equivalent from non-equivalent mutants (Chapters 6 and 7).

Chapter 4

The Javalanche Framework

For our research, we were interested in a mutation framework that can handle projects of significant size. Therefore, we developed JAVALANCHE, a mutation testing framework for JAVA with a special focus on *automation and efficiency*. To this end, we applied several optimizations:

Selective mutation As described previously, the idea of *selective mutation* is to use a small set of mutation operators that provides a sufficiently accurate approximation of the results obtained by using all possible operators [98]. JAVALANCHE, therefore, uses a slightly modified set of operators as used by Andrews et al. [3], which were also adapted from Offutt [69]. Table 4.1 lists the default operators that JAVALANCHE uses.

Mutant schemata Traditional mutation testing tools produce a new mutated program version for every mutation possibility. For large systems, this can result in several thousand different mutated versions, which are too many to be handled effectively. Furthermore, executing each mutant in a separate process increases the runtime because the environment (JAVA virtual machine, data base connections) has to be set up for every mutant. To reduce the number of generated versions, we use mutant schema generation [98]. Mutant schema generation produces a metaprogram that is derived from the program under study and contains multiple mutations. In contrast to the original approach that introduces metamutations, we insert several simple mutations at once. Each mutation is guarded by a conditional statement so that it can be switched on and off at runtime.

Coverage data Not all tests in the test suite execute every mutant. In order to avoid executing those tests, we collect coverage information for each test before executing the mutants. When executing mutants, JAVALANCHE only executes those tests that are known to cover the mutated statement.

Manipulate bytecode Traditionally, a mutation is inserted into the source code. Then, this mutated source code version gets compiled. JAVALANCHE avoids the costly recompilation step by manipulating the JAVA bytecode directly (using the ASM bytecode manipulation and analysis framework [9]). This is done via the agent mechanism that is provided by the JAVA platform. This mechanism allows to instrument all classes that are loaded by the JAVA virtual machine.

Parallel execution JAVALANCHE can execute several mutants in parallel, thus taking advantage of parallel and distributed computing. To this end, the mutations are split into several subtasks and these subtasks are distributed to several processors.

Automation JAVALANCHE is fully automated. In order to apply mutation testing to a program, only the name of a test suite, the base package name of the project, and the set of classes needed to run the test suite are required.

Handle endless loops In order to handle a huge number of mutations efficiently, a mutation testing tool must be able to detect and abort runs where a mutation causes an endless loop. To this end, JAVALANCHE executes the mutations in separate threads and gives each test a predefined period (the default is 10 seconds) to run. After the timeout is reached, the mutation is disabled, and it is checked whether this causes the program to finish regularly. If the program is still running, the `Thread.interrupt()` and optionally the `Thread.stop()` method is called in order to abort the execution. When JAVALANCHE was not able to stop the run of the tests on a mutation, it shuts down the virtual machine and starts a new one for the next mutation. Although starting the virtual machine again introduces a runtime overhead, this is done because a thread that is stuck in an endless loop consumes computing resources and might influence other runs. If a mutation causes an endless loop, it is considered to be detected because an implicit assumption is that a run of a test case always terminates.

Stop after first fail If one is only interested in whether a mutation is detected or not, there is no need to execute more tests once a mutation is detected by a test. Therefore, JAVALANCHE stops the execution of tests for a mutation once the first test fails. This behavior, however, is configurable because for some applications, it is also beneficial to check the result for all tests that cover a mutation.

Table 4.1: JAVALANCHE mutation operators.

Replace numerical constant Replace a numerical constant X by $X+1$, $X-1$, or 0.
Negate jump condition Replace a conditional jump by its counterpart. This is equivalent to negating a conditional statement or subexpression in the source code. (Since composite conditions compile into multiple jump instructions, this also negates individual subconditions.)
Replace arithmetic operator Replace an arithmetic operator by another one, e.g. $+$ by $-$.
Omit method call Suppress a call to a method. If the method has a return value, a default value is used instead, e.g. $x = Math.random()$ is replaced by $x = 0.0$.

4.1 Applying Javalanche

When JAVALANCHE is applied to a project, the mutation testing process is conducted in two steps. First, the project is scanned to extract information that JAVALANCHE later needs to execute the mutations, and then, in a second step, the actual mutation testing is carried out.

However, before applying JAVALANCHE to a project, it must be ensured that the project's test suite is suitable for mutation testing. This means that all tests in the suite pass and that they are independent of each other, i.e. running the tests multiple times in a different order gives consistent results. To this end, JAVALANCHE provides several commands that run the test suite in different settings and check for these preconditions.

In the scanning step, JAVALANCHE executes the test suite in order to collect the following information about the project:

Mutation possibilities The most important information are all mutation possibilities for a project. A mutation possibility is a statement in the program where a mutation operator can be applied. It is uniquely identified by its location that consists of the containing class, the operator that produced the mutation, the line number of the mutated statement, the number of the mutation in this line (if the operator can be applied more than once in this line), and an optional additional parameter for the mutation operator. For example, the operator that replaces constants stores the numeric value that is inserted as an additional parameter. During the scan step all mutation possibilities are stored in a database.

Coverage data For each mutation possibility, the coverage data is collected so that JAVALANCHE knows which mutation is executed by which tests. To this end, additional statements are inserted into the bytecode that log the execution for each mutation possibility.

In JAVA, there is also code that is only executed when a class is loaded by the JAVA Virtual Machine (JVM). The JVM loads a class exactly once when it is first accessed by another class. Thus, the execution of this code depends on which code was executed earlier. This, however, is not under the control of a test. Therefore, JAVALANCHE does not consider these static initializations for the coverage data. To this end, it is made sure that all classes are loaded by executing all tests once before the coverage information is collected.

Including static initialization code would lead to more mutations to be covered and more mutations might be detected. However, considering the initialization code for coverage and mutant detection would involve setting up a new environment for each test case that is executed, which would be too costly when executing several thousand mutations. Furthermore, it is not regarded as good practice to rely on the order of class loading for a test.

Detect test classes JAVALANCHE applies several heuristics to detect classes that belong to the test suite. By default, these classes are not mutated. The results, however, are stored in text files and can be modified when JAVALANCHE erroneously classifies a class to be part of the tests which is actually a class of the program that should be mutated.

Impact information JAVALANCHE supports computing the impact of a mutation. The impact is computed by comparing properties between a run of the original version and a run of the mutated version of the program (see Chapter 6 and Chapter 7). To this end, information about an unmodified original run must be collected, which is also done in the scanning step.

Optional information JAVALANCHE supports to collect additional information in this step. Examples include information about inheritance relations between classes that are used for specific analyses of the mutation test results, and information about jump targets for experimental mutation operators.

In the second step, actual mutations are inserted for the previously collected mutation possibilities. As the mutations are inserted via mutant schema generation, additional checks are introduced for every mutation. The goal of mutant schema generation is to save the set-up time. The checks, though, also introduce some runtime overhead.

Table 4.2: Description of subject programs.

Project name	Description	Version	Program size (LOC)
ASPECTJ	AOP extension to JAVA	cvs: 2010-09-15	28,476
BARBECUE	Bar code creator	svn: 2007-11-26	4,837
COMMONS-LANG	Helper utilities	svn: 2010-08-30	18,452
JAXEN	XPath engine	svn: 2010-06-07	12,438
JODA-TIME	Date and time library	svn: 2010-08-25	26,582
JTOPAS	Parser tools	1.0(SIR)	2,031
XSTREAM	XML object serialization	svn: 2010-04-17	15,266

Lines of Code (LOC) are non-comment, non-blank lines as reported by `sloccount`.

For ASPECTJ, we only considered the *org.aspectj.ajdt* package.

Therefore, a trade-off between the set-up time and the overhead for checking the mutation flags has to be found. To this end, the set of all mutation possibilities is divided into several subtasks (by default, JAVALANCHE creates tasks that contain 400 mutations). These tasks can be processed sequentially, or they can be distributed to different processors.

The results of executing the mutations are stored in a database. A result for a mutation consists of all tests that were executed for this mutation and their outcome and whether they passed or failed. Additionally, the output of the tests is stored. This feature can be turned off in order to save disk space.

4.2 Subject Programs

The goal of JAVALANCHE is to carry out mutation testing on real-life programs. In order to be suitable for mutation testing with JAVALANCHE, a program has to be written in JAVA (or another language that can be compiled into JAVA bytecode), and it has to come with a JUNIT test suite. To this end, we selected seven open-source projects that fulfill these criteria. These projects are listed in Table 4.2. The subjects come from different application areas (column 2), and for each project, we took a recent version from the revision control system (column 3)—except for JTOPAS, which was taken from the software-artifact infrastructure repository (SIR) [19]. The size of each project (column 4) is measured in lines of code (LOC), which refers to non-comment,

Table 4.3: Description of the subject programs' test suites.

Project name	Test code size (LOC)	Number of tests	Test suite runtime (s)	Statement coverage (%)
ASPECTJ	6,830	339	11	38
BARBECUE	3,293	153	3	32
COMMONS-LANG	29,699	1,787	33	88
JAXEN	8,418	689	11	78
JODA-TIME	50,738	2,734	37	71
JTOPAS	3,185	128	2	83
XSTREAM	16,375	1113	7	77

For ASPECTJ, we only considered the tests of the *org.aspectj.ajdt* package.

Table 4.4: Mutation statistics for the subject programs.

Project Name	Number of mutants	Covered mutants (%)	Mutation score (%)	Mutation score (covered) (%)
ASPECTJ	19,236	48.25	33.11	68.62
BARBECUE	17,629	9.56	5.68	59.41
COMMONS-LANG	15,715	92.89	80.13	86.26
JAXEN	9,279	68.73	48.30	70.28
JODA-TIME	21,647	64.94	54.04	83.21
JTOPAS	1,678	83.43	66.98	80.29
XSTREAM	8,215	79.76	71.26	89.35

non-blank lines as reported by the `sloccount` tool [105], and does not include test code. The size varies from 2,031 lines for JTOPAS up to 28,476 lines of code for *org.aspectj.ajdt* package of ASPECTJ.

Table 4.3 summarizes the test suites of the projects. We removed all tests that failed on the original version from the test suites, as well as tests whose outcome depended on the order of test execution. The size of the test suites ranges from 3,185 lines of code for JTOPAS to 50,738 for JODA-TIME, which in this case is more lines of code than the program that is tested. The total number of test cases for each project is given in column 3. For example, the test suite of JODA-TIME, which has the most lines of code, consists of 2,734 test cases. Running the test suites of the projects (column 4)

Table 4.5: JAVALANCHE runtime for the individual steps.

Project name	Scanning step	Mutation testing	Time per mutant
ASPECTJ	2m 01s	2h 29m	0.96s
BARBECUE	1m 04s	7m	0.25s
COMMONS-LANG	2m 02s	1h 18m	0.32s
JAXEN	1m 39s	1h 39m	0.93s
JODA-TIME	2m 09s	50m	0.21s
JTOPAS	3m 21s	1h 01m	2.61s
XSTREAM	2m 03s	1h 41m	0.92s
All	14m 19s	9h 05m	0.61s

Time reported is the time used to execute the steps ("user time") measured using `time`;

takes between 2 and 37 seconds. The statement coverage is given in the last column, which is the percentage of lines that is executed by the tests relative to all executable lines, is lowest for BARBECUE with 32% and highest for COMMONS-LANG with 88%.

4.3 Mutation Testing Results

In order to check whether JAVALANCHE is capable of mutating real-life programs, we applied it to the seven subject programs.

Table 4.4 gives the mutation testing results. The second column shows the *total number of mutations,* ranging from 1,678 for JTOPAS to 21,647 for JODA-TIME. The *coverage rate* of mutations, given in the third column, indicates how many of the mutations were actually executed. For most projects, it varies between 60 to 80%. For ASPECTJ, this rate is lower since we only executed the tests for the *org.aspectj.ajdt* package. The low coverage rate for BARBECUE is due to the fact that most mutations appear in classes that mainly consist of the static initializations of maps for bar code values. As explained in Section 4.1, mutations only executed during class loading are not considered to be covered.

As discussed earlier, the mutation score is the percentage of detected mutations relative to all mutations, which is given in column 4. A low score for covered mutations implies a low mutation score. A mutation that is not executed by the test suite, however,

cannot be detected by a test suite. Thus, we also present the *mutation score for the covered mutations* (column 5). ASPECTJ, for example, has a mutation score of 33%. However, when we only consider the covered mutations, it has a score of 69%.

The time needed for mutation testing the projects is given in Table 4.5. The times were measured on a server with 16 Intel Xeon 2.93GHz cores and 24 GB main memory. For the experiments, however, JAVALANCHE was limited to one processor and 2GB memory in order to achieve comparable results and not interfere with other processes.

Furthermore, all optimizations of JAVALANCHE were enabled. The scanning step (column 2) takes between 1 and 2 minutes for all projects. The actual mutation testing step that checks all covered mutations takes between 7 minutes for BARBECUE and 2 hours and 29 minutes for ASPECTJ. As the number of mutations varies between the different projects, we also calculated the average time that is needed for one mutation (last column). The time per mutation ranges from 0.21 seconds for JODA-TIME to 2.61 seconds for JTOPAS. The high number for JTOPAS compared to the other projects is due to several mutations that cause an infinite loop. With the default settings, JAVALANCHE stops every test after 10 seconds. If a project, like JTOPAS, has many mutations that cause an endless loop, this means that each of these mutations takes at least 10 seconds to check, which is far above the average execution time of a mutant.

4.4 Related Work

Since mutation testing was first proposed in the late 1970's [18], several tools for different programming languages have been built. The first tool was presented by Budd et al. [12]. Their pilot mutation system (PIMS) can mutate FORTRAN code. It implements 25 different mutation operators and was built as an interactive tool that asks the user to provide test data for not yet detected mutations.

Later the MOTHRA framework was introduced [74], which was the basis for a lot of research in mutation testing [73, 44]. The MOTHRA framework consists of a set of tools that perform the different tasks in mutation testing. It uses an interpreter approach to apply mutation testing to FORTRAN 77 programs. The advantage of this approach is that the mutations can be inserted by the interpreter. Thus, the overhead for maintaining several mutated versions is avoided. Furthermore, this approach provides access to program internals. This, for example, is used to detect endless loops by counting the number of statements that are executed while executing a mutation and comparing it

to an expected number. However, executing a program in an interpreter usually takes significantly more time than running a program in its native environment.

Another widely used mutation testing framework is PROTEUM [16, 6, 17]. It mutates programs written in C, and in contrast to MOTHRA, it produces different mutated versions and runs them in their native environment. PROTEUM supports 108 mutation operators that also include interface mutation operators which aim to assess the quality of integration tests by mutating code that is related to the communication between two modules.

μJava, a mutation testing tool for JAVA, was presented by Ma et al. [55]. To optimize the execution of mutations it implements mutant schema generation and manipulates the bytecode directly. It supports a huge number of traditional mutation operators and introduces object-oriented mutation operators, which mimic errors that arise due to object-oriented features. For example, *hiding variable deletion* is an operator that deletes each declaration of an overriding or hiding variable. However, in contrast to JAVALANCHE, the μJava tool provides little support for automatization as it requires user input to apply the mutations and run the tests.

4.4.1 Further Uses of Javalanche

Besides our uses of JAVALANCHE that will be presented later in this work, it has also been used by other researchers.

Gligoric et al. [29] extended JAVALANCHE for efficient mutation testing of multithreaded code. Multithreaded code adds a cost factor to mutation testing because different thread schedules have to be considered. Their MUTMUT tool (MUtation Testing of MUltiThreaded code) allows to explore different thread schedules for a mutation. By recording the first state on an execution path that reaches a mutation, MUTMUT can prune schedules that do not reach a mutation. Hence, it saves execution time. Compared to a basic technique that only prunes tests that can never reach a mutant, MUTMUT achieves a performance improvement up to 77%.

Fraser and Zeller [28] used JAVALANCHE for their approach to automatically generate test cases with oracles. Their μTEST tool applies a genetic algorithm to create tests that detect mutations. To this end, μTEST uses a fitness function that assesses the input quality based on whether it reaches a mutation and whether it impacts the execution of a mutation (see the following Chapters 6 and 7 for impact measures). When an input was generated that executes the mutation and causes it to have an impact on

the execution, μTEST tries to generate assertions that detect the mutation. To this end, the original and mutated versions are run with the generated inputs, and the runs are analyzed for differences in primitive values and objects. If such a difference is found, an assertion that checks for this property is introduced. In a last step, a heuristic is used to find a minimal set of assertions that still detects all detectable mutations. The evaluation on two open-source libraries showed that μTEST produces test suites that detect significantly more mutations than manually written test suites.

Aaltonen et al. [1] used JAVALANCHE to assess the quality of test suites generated by students. In the course of an introductory programming class, students had to create a test suite for their own implementation. The students were rewarded when their test suites reached a high statement coverage level. However, it was observed that many of them fooled the system. They created weak test suites that only reached a high coverage level, i.e. their test suites contained weak or nonsensical checks. Thus, there was an interest in a technique that assesses the test quality more thoroughly. Experiments using *mutation testing* showed that it *is more appropriate than coverage metrics to assess the quality of test suites* generated by students. However, it is mentioned that mutation testing can also be fooled by creating meaningless code that contains a lot of easily detectable mutations.

Sharma et al. [92] used JAVALANCHE to compare the effectiveness of different test generation strategies. In their work, they compared random test generation with a test generation strategy based on shape abstraction which has been found to perform best in terms of satisfying coverage metrics in a previous study. The random testing technique generates a fixed number of tests that consist of randomly generated method call sequences of a fixed length. The shape abstraction strategy explores all method sequences up to a fixed length. However, during the generation of method sequences, a list of abstract states is maintained which is used to prune sequences. The abstract state of an object after the execution of a sequence is compared with the already encountered abstract states. If the abstract state has not been encountered before, it is added to the list, and its producing sequence is used as basis for further sequences. Otherwise, if the abstract state has been encountered before, the producing sequence is pruned. The output of the algorithm are all sequences that exercise a predicate combination not yet exercised. In an experiment, the two strategies are compared on 13 container classes, and the results showed that random testing achieves a better mutation score for 4 classes, shape abstraction for 5 classes, and that there is no significant difference for the remaining 4 classes. According to these results, it is concluded that random testing is comparable to shape abstraction while random testing is less expensive in terms of computation resources.

4.5 Summary

We presented JAVALANCHE, a mutation testing framework for JAVA. JAVALANCHE was developed with a focus on efficiency and automation and applies several optimization techniques in order to carry out mutation testing on real-world programs. As optimization techniques, JAVALANCHE uses mutant selection to reduce the number of mutants that have to be checked, mutant schemata to save set-up costs, coverage data to reduce the number of tests that have to be executed, it manipulates bytecode to prevent recompilation steps, and it allows for parallel execution of mutants. JAVALANCHE carries out mutation testing in two steps. First, in a scan step information about the project is collected, e.g. which mutations can be applied and which tests cover which mutations. In a second step, the actual mutation testing is carried out, i.e. the mutations are inserted into the program and it is checked whether the tests detect them.

The results from an experiment on seven open-source projects showed that mutation testing can be applied to programs with up to 28,000 lines of source code. In total, 93,399 mutations were applied to the seven projects, out of which 58% were covered by at least one test. Checking one mutation took 0.61 seconds on average. Among all projects, the test suites detected about 46% of all mutants and 80% of the covered mutants. Because of its efficiency in carrying out mutation testing, JAVALANCHE was used by other researchers. For example, it was extended for mutation testing of multi-threaded code [29], used in an approach that generates tests with oracles [28], used to assess the quality of test suites generated by students [1], and to compare the effectiveness of different test generation strategies [92].

Chapter 5

Equivalent Mutants

One usage scenario of mutation testing is to improve a test suite by providing tests for undetected mutants. To this end, mutations are applied to a program, and it is checked whether the test suite detects them or not. This step is carried out automatically and results in a set of undetected mutants. A programmer then tries to add or modify existing tests so that previously undetected mutants are detected. There are several reasons why a test suite might fail to detect a mutation, which determine the usefulness to the programmer:

1. The mutation may not change the program's semantics and *cannot be detected*. These equivalent mutants cannot help improving the test suite and place an additional burden on the programmer because the equivalence of a mutation has to be assessed manually.

2. The mutated statement may *not be executed*. In order to find non-executed statements, standard coverage criteria can be used.

3. The mutation may not be detected because of an *inadequate test suite.* These are the *most valuable* mutations since they provide indicators to improve the test suite that other coverage metrics might not provide. If a mutation is covered but not detected, this either means that the tests do not check the results well enough, or that the input data is not chosen carefully enough to trigger the erroneous behavior.

```
...
for (final Iterator iter = methods.iterator();
      iter.hasNext();) {
  final Method method = (Method)iter.next();
  method.setAccessible(true);
  if (Factory.class.isAssignableFrom(
       method.getDeclaringClass())
     || ⇒ &&
     (method.getModifiers() & (Modifier.FINAL |
      Modifier.STATIC)) > 0) {
    iter.remove();
    continue;
...
```

Figure 5.1: A non-equivalent mutation from the XSTREAM project.

5.1 Types of Mutations

In the following sections, we will present examples for the different types of mutations.

5.1.1 A Regular Mutation

A mutation in the *createCallbackIndexMap* method of the *CGLIBEnhancedConverter* class is shown in Figure 5.1. This mutation changes an `||` operator to an `&&` operator which can cause the expression to evaluate to `false` when it should evaluate to `true`, and then the method is not removed from the underlying map. Eventually, this results in spurious entries in the XML representation of an object. An existing test case of the XSTREAM test suite triggers this behavior (*testSupportProxiesUsingFactoryWithMultipleCallbacks* in class *CglibCompatibilityTest*). However, this test fails to check the results thoroughly. By modifying this test, the mutation can be detected.

5.1.2 An Equivalent Mutation

Another mutation of the XSTREAM project is shown in Figure 5.2. It is applied to the `com.thoughtworks.xstream.io.json.JsonWriter` class. Here, the mutation changes an `&` operator to an `|` operator which might cause the expression to evaluate to `true` when it should not. The result of this expression gets connected to

```
void addValue(String value, Type type) {
  if (newLineProposed && ((format.mode()
        & ⇒ |
      Format.COMPACT_EMPTY_ELEMENT) != 0)) {
    writeNewLine();
  }
  if (type == Type.STRING) {
    writer.write('"');
  }
  writeText(value);
  if (type == Type.STRING) {
    writer.write('"');
  }
}
```

Figure 5.2: An equivalent mutation from the XSTREAM project.

the `newLineProposed` variable via a logical conjunction. As JAVA applies short-circuit evaluation, the mutated code only gets executed when the variable evaluates to `true`. Further investigation shows that the `newLineProposed` variable is only set to `true` in one place of the program. This only happens if the same condition as in the mutated statement is `true`. Thus, the mutated condition is always `true` when it is evaluated, and the mutant is equivalent.

5.1.3 A Not Executed Mutation

```
public boolean aliasIsAttribute(String name) {
  return
     nameToType.containsKey(name) ⇒ false;
}
```

Figure 5.3: A mutation of XSTREAM project that is not executed by tests.

The method *aliasIsAttribute* of *ClassAliasingMapper* shown in Figure 5.3 returns `true` if the given name is an alias for another type. What happens if we mutate this method so that it always returns `false`? The existing test suite does not detect this mutation because the statement is not executed. Thus, a test should be added that checks this functionality. However, to detect uncovered code, we do not need to apply

full-fledged mutation testing. Simple statement coverage does this much more efficiently. For the remainder of this work, we thus assume that mutations are only applied to statements that are executed by the test suite.

5.2 Manual Classification

We saw that determining the equivalence of a mutant requires manual investigation. But how widespread is this problem in real programs? Offutt and Pan [71] reported 9.10% of equivalent mutants (relative to all mutants) for the 28-line `triangle` program. As we were interested in the extent of the problem on modern and larger programs, we applied mutation testing (Section 4.2) to our seven subject programs and investigated the results.

For each of the seven projects, we randomly took 20 mutations from different classes that were not detected by the test suite for manual inspection. Then, we classified each mutation either

- as *non-equivalent*, as proven by writing a test case that detects the mutation; or
- as *equivalent* when manual inspection showed that the mutation does not affect the result of the computation.

5.2.1 Percentage of Equivalent Mutants

The results for classifying the 140 mutations, which stem from 20 mutants for each of the seven subject projects, are summarized in Table 5.1. Out of all classified mutants, 77 (55%) were non-equivalent and 63 (45%) were equivalent. The project with the highest ratio of non-equivalent mutants is ASPECTJ with 75%, while COMMONS-LANG had the lowest percentage with 30%. Such differences might also indicate differences in the quality of the test suites, as better test suites have a higher rate of equivalent mutations among their undetected mutations. Notice that the ratio of 45% of equivalent mutants relates to the undetected ones. Relative to all mutants, we obtain a ratio of 7.39% of equivalent mutants.

*On our sample of real-life programs, **45%** of the undetected mutations were equivalent.*

Table 5.1: Classifying mutations manually.

Project name	Non-equivalent mutants	Equivalent mutants	Average classification time
ASPECTJ	15 (75%)	5 (25%)	29m
BARBECUE	14 (70%)	6 (30%)	10m
COMMONS-LANG	6 (30%)	14 (70%)	8m
JAXEN	10 (50%)	10 (50%)	15m
JODA-TIME	14 (70%)	6 (30%)	20m
JTOPAS	10 (50%)	10 (50%)	7m
XSTREAM	8 (40%)	12 (60%)	11m
All	**77 (55%)**	**63 (45%)**	**14m28s**

5.2.2 Classification Time

The time that was required to classify the mutations as equivalent or non-equivalent varied heavily. While some mutations could be easily classified by just looking at the mutated statement, others involved examining large parts of the program to determine a potential effect of the mutated statement. This led to a maximum classification time of 130 minutes. The average classification time for each project is given in the last column of Table 5.1 and ranges from 7 minutes for JTOPAS up to 29 minutes for ASPECTJ. The average time for all projects is 14 minutes and 28 seconds.

*On average, it took us 14 minutes and 28 seconds to classify **one single** mutation for equivalence.*

5.2.3 Mutation Operators

JAVALANCHE generates mutations by using different mutation operators as introduced in Section 4. In order to check the relation between the equivalence of a mutant and its underlying mutation operators, we grouped the 140 manually classified mutations according to their operator. The results are summarized in Table 5.2. Some operators produce far more mutants than others (column 2). For example, the operator *replace*

Table 5.2: Classification results per mutation operator.

Mutation operator	Number of mutants	Non-equivalent mutants	Equivalent mutants
Replace numerical constant	78	34 (44%)	44 (56%)
Negate jump condition	12	10 (83%)	2 (17%)
Replace arithmetic operator	7	3 (43%)	4 (57%)
Omit method call	43	30 (70%)	13 (30%)

numerical constant produces over half of the mutants in our sample. However, we can also check the ratios of non-equivalent (column 3) and equivalent (column 4) mutants for the operators. Here, we see that for the operators *replace numerical constant* and *replace arithmetic operator*, which manipulate data, around 57% of all produced mutants are equivalent, while for the operators *negate jump condition* and *omit method call*, which manipulate the control flow, only around 30% are equivalent.

Mutation operators that change the control flow produce less equivalent mutants than those that change the data.

5.2.4 Types of Equivalent Mutants

A mutation is defined to be equivalent when no test can be written that distinguishes the mutated version from the original, e.g. the mutated version produces the same results as the original. When classifying mutations, we found several reasons why this can be the case. In the following, we list some of the reasons why a mutation can be equivalent. However, this list does not include all possible reasons.

Mutations in unneeded code Some parts of the code just duplicate default behavior or set a value that is later reset without using it in-between. When mutating these parts, the program is not affected.

As an example for such a mutation, consider the code shown in Figure 5.4 from the `org.jaxen.pattern.PatternParser` class, which calls the method `setXPathFactory()` after creating a new `JaxenHandler` object. This call is unnecessary as the standard constructor of `JaxenHandler` sets the appropriate field to `DefaultXPathFactory()` anyway. Thus, the mutation that suppresses this call has no effect on the program execution.

```
JaxenHandler handler = new JaxenHandler();
handler.setXPathFactory(new DefaultXPathFactory());
   ⇒ call to setXPathFactory(...) gets omitted
```

Figure 5.4: An equivalent mutation in unneeded code.

Mutations that suppress speed improvements Some code is only considered with speed improvements, e.g. a check that suppresses the sorting of a list if it has less than two elements. Mutations to these checks can cancel the performance improvement, but the results of the computation remain the same.

```
String myPrefix = n.getPrefix();
if ( !nsMap.containsKey(myPrefix) ⇒ true )
{
  NamespaceNode ns = new NamespaceNode((Node)contextNode,
                                       myPrefix, myNamespace);
  nsMap.put(myPrefix, ns);
}
```

Figure 5.5: An equivalent mutation that does not affect the program semantics.

For example, the code shown in Figure 5.5 checks whether a string `myPrefix` is already contained in a map `nsMap`. If not, a new node is created and added to `nsMap` together with a new `NamespaceNode` object. A possible mutation to this code is to change the condition to a constant such as `true`. This updates the map entries even if the prefix string already is in the map, and should result in very different map contents. However, it is an equivalent mutant since the values of the `contextNode` and `myNamespace` variables have the same values for every `myPrefix` instance. Thus, the `NamespaceNode` objects are the same, too. The map update as forced by the mutation does not alter the map contents—it just replaces an existing node with a new, equivalent one.

Changes that do not propagate These mutations can alter the state of the program, but the changes do not propagate to the end of the execution. This can either happen locally, inside a method or via the private state of the class.

Let us consider a mutation to the code shown in Figure 5.6 taken from the class `org.jaxen.expr.NodeComparator`. The value to start the depth computation from is initialized with 1 instead of 0, which causes different depth values to be returned by the method. This method is private, and only used by a method that compares different nodes. Since the depth for all nodes that are compared is

```
private int getDepth(Object o) throws
            UnsupportedAxisException {
  int depth =   0 ⇒ 1
  Object parent = o;
  while ((parent = navigator.getParentNode(parent)) != null){
    depth++;
 }
return depth;
}
```

Figure 5.6: An equivalent mutation that alters state.

increased by 1, the comparison of nodes remains correct.

Effects that cannot be triggered These are mutations that would cause the program to fail under specific conditions. However, meeting these conditions causes other failures upstream.

Figure 5.7 displays a mutation to the `org.jaxen.dom.NamespaceNode` class. The mutation replaces the contents of the first if condition with `false` so that the body is never executed. However, supplying an XML Document that would evaluate the non-mutated condition to `true` causes an exception by the XML parser.

Equivalence because of local context Some mutations change the syntax of the program, but from the local source code context it is obvious that the mutation does not change the semantics of the program.

Consider the code shown in Figure 5.8. The mutation changes the comparison operator from $<$ to $!=$. This mutant is equivalent because the loop still runs from 0 to 5 as the variable *i* is not manipulated in the body of the loop.

5.2.5 Discussion

The reported number of 45% of equivalent mutants is much higher than the 6.24% that are reported by Offutt and Craft [68] or the 9.10% reported by Offutt and Pan [71]. Their numbers are relative to all mutations, including the ones that are detected by the test suite. These mutations, however, are not of interest when improving the test suite as they do not indicate a weakness of the test suite. If we also take the detected

```
NamespaceNode (Node parent, Node attribute)
  String attributeName = attribute.getNodeName();
  if ( attributeName.equals("xmlns") ⇒ false ) {

    this.name = "";
  }
  else if (attributeName.startsWith("xmlns:")) {
    this.name = attributeName.substring(6);
    // the part after "xmlns:"
  }
  else { // workaround for Crimson bug;
      // Crimson incorrectly reports the
      // prefix as the node name
    this.name = attributeName;
  }
  this.parent = parent;
  this.value = attribute.getNodeValue();
}
```

Figure 5.7: An equivalent mutation that could not be triggered.

```
for(int i=0; i <⇒ != 5; i++){
 ...// Code that does not manipulate i
}
```

Figure 5.8: An equivalent mutation because of the context.

mutations into account, we found 7.39% of all mutations to be equivalent, which is in line with the other reported numbers.

In practice, though, it is the percentage of equivalent mutants across the *undetected* mutations that matters—since these are the mutants that will be assessed by the developer. Here, 45% of equivalent mutants simply means 45% of wasted time. Even worse: While the percentage of equivalent mutants across *all* mutants stays fixed, the percentage of equivalent mutants across the *undetected* mutants increases as the test suite improves. This is due to the fact that an improved test suite detects more (non-equivalent) mutants. A perfect test suite would detect *all* non-equivalent mutants; hence, 100% of undetected mutants would be equivalent. In other words, as one improves the test suite,

one has more and more trouble finding non-equivalent mutants among the undetected ones—with growing effort as the test suite approaches perfection.

The percentage of equivalent mutants among the undetected mutants increases as the test suite improves.

5.3 Related Work

The problem of assessing mutation equivalence has been noted before. We are only aware of the following quantitative measures of equivalent mutants. Offutt and Pan [71] report 9.10% of equivalent mutants (relative to all mutants) for the 28-line `triangle` program, and in another paper, Offutt and Craft [68] report 6.24% of equivalent mutants for a set of small programs ranging from 5 to 52 lines of source code. Furthermore, it is also mentioned that programmers judge mutant equivalence correctly only in about 80% [71]. Hu et al. [41] did a study on object-oriented mutation operators which are mutation operators that simulate errors caused by object-oriented features of a programming language, e.g. hiding variables. In their study on 38 JAVA classes, they report 12.3% of equivalent mutants. In a comparative study between mutation testing and the all-uses adequacy criterion, Frankl et al. [27] do not directly report on the percentage of equivalent mutants, but from the numbers given in the paper we can conclude that they classified 14.2% out of 18,125 mutants as equivalent. The reason for the higher number of equivalent mutants might be traced back to the procedure they used to classify mutants. They used a random test generator to produce test inputs and considered a mutant as equivalent if no test input was generated that detected a mutant. However, they also state that they possibly misclassified some mutants as equivalent when using this procedure. Furthermore, they comment on the problem of equivalent mutants as follows:

> *Although our experiments were designed to measure effectiveness, we also observed that using these criteria, particularly mutation testing, was costly. Even for these small subject programs, the human effort needed to check a large number of mutants for equivalence was almost prohibitive.*

The problem of equivalent mutations was also investigated in several other papers [10, 39, 22], but they do not comment on the quantitative aspect of the problem.

5.4 Summary

A developer who uses mutation testing to improve a project's test suite focuses on the mutants that are not yet detected by the test suite. However, there are different reasons why a mutant is undetected. The simplest reason is that the tests might not execute the mutation. For these cases, however, we do not need to apply mutation testing because statement coverage would also report unexecuted code. Among the covered mutations, there are the ones that can be detected by a test, and the equivalent mutations that cannot be detected by a test. A developer is only interested in detectable mutants because only they can help in improving the test suite. Equivalent mutants put a burden on the programmer because the equivalence of a mutant has to be determined, but this does not help to improve the test suite.

In a manual study, we studied the extend of the problem of equivalent mutants for seven JAVA projects. To this end, we classified 140 undetected mutants from these projects and saw that equivalent mutants are a widespread problem. Classifying a mutant as equivalent or non-equivalent took on average about 15 minutes. About 45% of the undetected mutants in this sample are equivalent, and some mutation operators produce 57% of equivalent mutants. Therefore, it would be helpful to detect equivalent mutants automatically in order to make mutation testing applicable in practice. In the next two chapters, we thus present approaches to separate non-equivalent from equivalent mutants.

Chapter 6

Invariant Impact of Mutations

In the previous chapter, we have seen that equivalent mutants cause additional effort when interpreting the results from mutation testing. In the past, a number of efforts have been made to detect equivalent mutants. Approaches based on heuristics [68], static program analysis (in particular path constraints) [70], and program slicing [39] have been proposed. However, none of these techniques has shown to scale to large programs.

Therefore, we suggest an alternate, novel way to eliminate equivalent mutants. Our approach is based on the assumption that a non-equivalent mutant must impact not only the program code but also the program behavior—just like a defect must impact program execution to produce a failure. To characterize the "normal" behavior, we use *dynamic invariants*—specific properties that hold under a given set of inputs. These dynamic invariants form the pre- and postconditions for every function as observed during its executions. Our hypothesis is that a mutation which violates such invariants causes different behavior—and therefore, is much more likely to also violate the program's semantics than a mutation which satisfies all learned invariants. We further propose that testers focus on those mutants that violate *most* invariants (i.e. have the greatest impact on program behavior) and are still not caught by the test suite. Our approach is summarized in Figure 6.1. After learning dynamic invariants from test suite execution, JAVALANCHE instruments the program with invariant checkers (Step 1), generates mutations (Step 2), runs the test suite on each mutation (Step 3), and ranks mutations by the number of violated invariants. Finally, the tester (Step 4) improves the test suite to detect the top-ranked mutations.

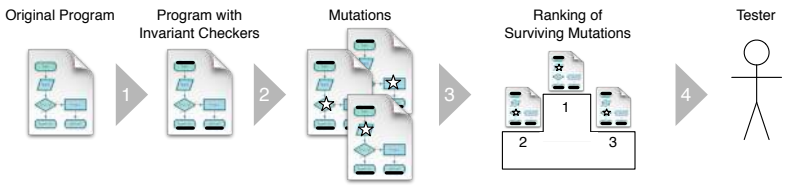

Figure 6.1: The process of ranking mutations by invariant impact.

6.1 Learning Invariants

To deduce invariants, JAVALANCHE relies on the tracer of ADABU [14] and the invariant detection engine of DAIKON. Invariants are learned in three steps: First, we run an instrumented version of the program and collect a trace of all parameter and field accesses. Second, we analyze the trace and generate input files for DAIKON. Finally, we feed those files into DAIKON to deduce invariants.

Tracing JAVALANCHE uses ADABU to instrument JAVA classes. ADABU injects code that writes interesting events such as field accesses, beginning and end of methods into a compact trace file. On a set of sample programs, the current ADABU implementation produces ∼30 MB/s of trace data on average.

DAIKON analyzes JAVA programs through CHICORY, a front-end for tracing JAVA programs. CHICORY uses *all* variables in the scope of a method, regardless of whether they were accessed or not. Consequently, the invariant detection engine has to deal with a lot more data which led to out-of-memory errors for some of the larger subjects in our case study. In contrast, ADABU uses only those variables that were *actually accessed by a method,* which greatly reduces the amount of data to be analyzed and allows for learning invariants even from very large programs.

Generating Daikon Files In the second phase, for every method that is invoked at least once, ADABU generates program point declarations for the beginning (*ENTER*) and the end (*EXIT*) of the method. For every method invocation, ADABU generates DAIKON trace entries for the corresponding *ENTER* and *EXIT* program points.

Running Daikon The generated files are then fed into DAIKON which supports over 85 different types of invariants. For performance reasons, we decided to limit

Table 6.1: Invariants used by JAVALANCHE.

Unary invariants. Compare a one-word integral variable (any JAVA type other than `long`, `float`, or `double`) X against $X \neq 0$, $X \neq$ `null`, $X \leq c$, $c \leq X$, $c_1 \leq X \leq c_2$, where c, c_1, c_2 are constants.
Binary invariants. Compare two one-word integral variables X_1 and X_2 against $X_1 = X_2$, $X_1 > X_2$, $X_1 < X_2$.
Strings and other objects are only checked for being `null`.

> DAIKON to those types of invariants that occur most frequently in practice. In a small experiment using a total of 94 different runs of our subject programs, we detected those DAIKON invariants that occurred most frequently. Our current configuration uses 28 different types of invariants, summarized in Table 6.1, accounting for over 95% of all invariants found in our sample.

As a consequence, large programs like ASPECTJ are handled with reasonable efficiency.

6.2 Checking Invariants

JAVALANCHE learns invariants from the *unmutated* program and checks for violations in the *mutated* program. To check for invariant violations of the mutations, we use a runtime checking approach that is similar to the approach pioneered by DIDUCE, a tool to learn invariants and detect violations at runtime [36].

For each learned invariant, we insert statements into the bytecode that check for invariant violations before and after the execution of a method. If an invariant is violated, this is reported and the run resumes. This allows for efficient and scalable checking of invariants.

Figure 6.2 shows an invariant checker for a method that computes the square root of a floating-point number. An invariant that might be discovered is that only numbers greater than or equal to zero are passed as parameters. The code displayed in bold face shows a check for this invariant. Note that the checkers are actually inserted into the bytecode, but for sake of brevity, the corresponding code is shown in JAVA. At the

```
public double sqrt(double d){
  if (d < 0) {
    InvariantObserver.invariantViolated(513);
  }
...
}
```

Figure 6.2: An invariant checker for a method that computes the square root.

beginning of the method the parameter is checked inline for the inferred invariant, and in the case of a violation, it is reported to a central instance called *InvariantObserver* with an id that identifies the invariant. The *InvariantObserver* also keeps track of the currently activated mutation and can associate the violated invariants with a mutation and keeps track about all invariants violated by a mutation.

6.3 Classifying Mutations

The results of applying JAVALANCHE with invariant impact detection contain the following information:

Detectability A flag indicates if the mutant was detected ("killed") by the test suite. A mutant is considered detected if at least one test fails, runs into a timeout, or throws an exception.

Impact Each invariant represents a different property of "normal" program runs. The more properties violated, the higher the impact of the mutation on the program execution. We, therefore, use *the number of invariants violated* by a mutation to measure impact; the greater the impact of a mutation, the higher the ranking.

Definition 26 (Invariant Impact) *For a mutant S_M of a program S and a set of tests T, the invariant impact I is the number of distinct invariants violated by running the tests T on S_M, and learning the invariants from running the tests T on S.*

$$I(S_M, S, T) = \#\ \textit{of violated invariants}$$

According to their impact, we can split the mutants into two sets: *non-violating mutants* (NVM) and *violating mutants* (VM). Non-violating mutants (NVM) are mutants that do not violate any invariant. They do not impact the program with respect to its dynamic invariants. On the contrary, violating mutants are mutants that violate at least one invariant.

Definition 27 (Non-Violating Mutants (NVM)) *The set of non-violating mutants $\mathscr{M}_{NVM}$ for a test set T are those mutants that do not violate any invariant.*

$$\mathscr{M}_{NVM} = \{S^i_M \mid I(S^i_M, S, T) = 0\}$$

Definition 28 (Violating Mutants (VM)) *The set of violating mutants $\mathscr{M}_{VM}$ for a test set T are those mutants that violate at least one invariant.*

$$\mathscr{M}_{VM} = \{S^i_M \mid I(S^i_M, S, T) \geq 1\}$$

The idea behind our approach is that violating mutants are less likely to be equivalent since they violate the typical behavior of the program as captured by invariants. This assumption will be investigated in the following section.

6.4 Evaluation

In order to evaluate our approach, we have conducted four different experiments. In Section 6.4.2, we manually assess a sample result for equivalent mutants. In Section 6.4.3, we compare the detection rates of mature test suites for invariant-violating mutants (VM) as well as for non-violating mutants (NVM). Section 6.4.4 investigates whether the mutants with the highest impact have the highest detection rate which implies a low number of equivalent mutants. In Section 6.4.5, we check whether invariant impact can be used as a predictor of equivalence or non-equivalence of undetected mutants.

6.4.1 Evaluation Subjects

For the evaluation of the invariant impact, we used the seven open-source projects presented in the previous chapter (see Table 4.2). However, in the evaluation of this

Table 6.2: Description of subject programs.

Project name	Version	Program size (LOC)	Test code size (LOC)	Number of tests
ASPECTJ	1.6.1	25,913	6,828	321
BARBECUE	1.5b1	4,837	3,136	137
COMMONS-LANG	2.5-S	18,782	31,940	1,590
JAXEN	1.1.1	12,449	8,371	680
JODA-TIME	1.5.2	25,861	47,227	3,447
JTOPAS	1.0(SIR)	2,031	3,185	128
XSTREAM	1.3.1	14,480	13,505	838

Table 6.3: Runtime (in CPU time) for obtaining the dynamic invariants.

Project name	Trace test suite	Create Daikon files	Learn invariants (Daikon)	Total time invariants
ASPECTJ	3m 17s	1h 02m	22h 37m	*23h 42m*
BARBECUE	17s	1m	16m	*17m*
COMMONS-LANG	3m 03s	13m	3h 23m	*3h 39m*
JAXEN	4m 48s	2h 01m	5h 31m	*7h 37m*
JODA-TIME	4m 36s	13m	13h 38m	*13h 56m*
JTOPAS	2m 44s	52m	1h 41m	*2h 36m*
XSTREAM	2m 52s	26m	5h 03m	*5h 32m*

approach, we used different (older) program versions (column 2) that are given in Table 6.2. The table also lists the size of the source code that is tested (column 3) and the size of test suite (column 4). The number of tests we used for the experiments is given in the last column. All times were measured on a 16-core 2.0GHz AMD Opteron 870 machine with 32 GB RAM; we used up to 7 cores and 2 GB RAM per core. It is important to note that CPU time does not depend on the number of cores.

Table 6.3 and Table 6.4 show the time required to perform the steps discussed in the previous sections. Columns 2 to 4 of Table 6.3 list the steps required to learn invariants, and the last column gives the total time that is needed to learn the invariants. The dominating step in terms of runtime is usually mining invariants with DAIKON, and for ASPECTJ, this process takes the longest: 22 hours and 37 minutes to run DAIKON,

Table 6.4: JAVALANCHE runtime (in CPU time) for the individual steps.

Project name	Create and test checkers	Prepare mutation testing	Check mutated versions	Total time mutants	Total time invariants + mutants
ASPECTJ	9h 35m	5m	14h 22m	*24 h 02 m*	**47h 44m**
BARBECUE	1m	2m	42m	*45 m*	**1h 02m**
COMMONS-LANG	6m	3m	2h 26m	*2 h 35 m*	**6h 14m**
JAXEN	13m	50m	2h 05m	*3 h 08 m*	**10h 45m**
JODA-TIME	19m	46m	3h 03m	*4 h 08 m*	**18h 04m**
JTOPAS	1m	16m	44m	*1 h 01 m*	**3h 37m**
XSTREAM	24m	1h 47m	2h 44m	*4 h 55 m*	**10h 27m**

and 23 hours and 42 minutes in total to learn the invariants. Columns 2 to 4 of Table 6.4 list the steps required to instrument and check mutated versions of the program. The rightmost column gives the total time needed to evaluate each subject, which is the sum of the time required to learn the invariants and to carry out mutation testing. Our record holder is again ASPECTJ, with a one-time effort of 24 CPU-hours to learn invariants and create checkers, and another 24 CPU-hours to run the mutation test.

6.4.2 Manual Classification of Impact Mutants

In order to check whether the invariant impact helps detecting non-equivalent mutants, we started our experiments with a manual assessment of invariant-violating mutants and non-violating mutants.

Hypothesis

Our hypothesis was:

H1 *Mutants that violate invariants are less likely to be equivalent than mutants that do not violate invariants.*

Experimental Setup

As discussed in Section 5.2.2, manually checking if a mutant is equivalent requires a lot of effort. We, therefore, restricted our evaluation to one project (JAXEN) and twelve samples for each group. The non-violating mutants were chosen randomly; the twelve violating mutants were those that displayed the highest number of violations.

For each mutant, we first examined the source code around the mutant; we then tried our best to come up with a test case to trigger the mutant. In those cases where it was not possible to come up with such a test, we considered the mutant to be equivalent. Assessing these 24 mutants took us 12 person-hours (i.e. 30 minutes per mutant on average), plus the additional time that was needed to write test cases for non-equivalent mutants.

The classification time of 30 minutes per mutant in this experiment is twice as much as the average reported for JAXEN in the classification for the 140 mutants (see Table 5.1). The reasons for this are the type of the mutants and the different setup. The sample from Section 5.2 contains more easy to classify equivalent mutants, i.e. mutants that can be classified as equivalent by looking at the surrounding code. Mutants that violate invariants, which make one half of the sample used in this experiment, cannot be classified by just looking at the surrounding code. Therefore, it takes more time to classify them. Furthermore, we set up a better infrastructure for the experiment on the 140 mutants, e.g. we set up all projects in an integrated development environment (IDE) that made navigating the code and writing tests more efficient.

Results

The results of this manual inspection indicate that mutants which violate invariants are more than *twice as likely to be non-equivalent:*

Violating mutants For 10 out of the 12 inspected mutants (10/12 = 83%), we could write a test that detects the mutant. Most of these mutations impacted tracking input line numbers, a feature that is not covered by the JAXEN test suite. We failed to write tests for the remaining two mutants. Therefore, we considered them to be equivalent.

```
public Context(ContextSupport contextSupport)
{
     this.contextSupport = contextSupport;
     this.nodeSet = Collections.EMPTY_LIST;
     this.size =   0 ⇒ -1 ;
     this.position = 0;
}
```

Figure 6.3: A non-detected JAXEN mutation that violates most invariants.

Non-violating mutants We were able to write tests for only 4 out of the 12 mutants (4/12 = 33%), but failed to do so for 7 of the remaining mutants. We did not categorize one mutant since the behavior of the mutant depends on the system locale.

A Fisher Exact Value test confirms the statistical significance of the difference at $p = 0.036$. We, therefore, accept our hypothesis **H1**—taken with a grain of salt as the size of our sample set was small.

In our sample, ***mutants that violate several invariants are less likely to be equivalent.***

Qualitative Analysis

In order to see whether invariant violating mutations help improving a project's test suite, we did a qualitative analysis for some of the mutations that violate most invariants.

The mutation that impacts most invariants for JAXEN is shown in Figure 6.3. It originates from line 102 of the `org.jaxen.Context` class. The mutation sets the initial size of the context to -1 instead of 0. This violates several preconditions for methods that use this information about the size during their computation, e.g. that the size is greater than zero or in a certain range. However, while the test suite checks the size of the underlying node set (whose size seems to be tracked by `Context`'s size field) after the creation of a `Context` object, it does not check the size of the `Context` itself (in Test `ContextTest`). Thus, this is either an insufficiency in the

```
private Token plus()
{
  Token token = new Token(TokenTypes.PLUS,
      getXPath(),
      currentPosition(),
      currentPosition() + 1 ⇒ 0);
  consume();
  return token;
}
```

Figure 6.4: A non-detected JAXEN mutation that violates the second most invariants.

test suite, or a *code smell* since the size may always be computed from the underlying node set.

The mutation with the second largest invariant impact can be found in line 615 of class `XPathLexer` (Figure 6.4). It sets the end index of a plus token to the same value as the begin index, making it a token of size 0. This leads to several invariant violations that involve the `tokenEnd` field of the `Token` class. The mutation, for example, has an effect on the program whenever a string representation of this token is requested, e.g. in error messages for wrong XPATH expressions. While the test suite checks for the *errors,* it does not check *the expressions enclosed in error messages.* Such checks take place for many other JAXEN messages, though. The mutation presented above indicates another opportunity to improve the test suite.

Inspired by these examples, we also did a qualitative analysis for some of the ASPECTJ mutations. In Figure 6.5, we see the method `getLazyMethodGen()` of the class `LazyClassGen` in ASPECTJ. This method takes a method name and signature and returns a `LazyMethodGen` object with the same name and signature or `null`. The mutation negates a condition and changes the behavior so that a `LazyMethodGen` is only returned when the names match but the signatures do not.

This change impacts not only the behavior of the method itself but also violates 39 of the inferred pre- and postconditions of 18 other methods that are scattered all throughout the program. Yet, this mutation is not detected by any test which implies that the ASPECTJ test suite is not adequate with respect to such defects. Although a test triggers this behavior, it fails to check for the error that was caused. As a consequence, a test manager should extend the test suite so that this mutation could be detected, too.

```
public LazyMethodGen getLazyMethodGen(String name,
    String signature, boolean allowMissing) {
  for (Iterator i = methodGens.iterator(); i.hasNext();){
    LazyMethodGen gen = (LazyMethodGen) i.next();
    if (gen.getName().equals(name) &&
          ⇒ !  (gen.getSignature().equals(signature)))
      return gen;
  }
  if (!allowMissing)
    throw new BCException("Class " + this.getName() +
        " does not have a method " + name +
        " with signature " + signature);
  return null;
}
```

Figure 6.5: The undetected mutation that violates most invariants.

6.4.3 Invariant Impact and Tests

The manual effort that is required for assessing mutations is not only a problem in mutation testing itself; it is also a problem when *evaluating* mutation testing approaches. In particular, the precise rate of equivalent mutants can only be measured by assessing all mutations manually, which is precisely the problem we want to overcome. Therefore, for our second experiment, we wanted to have an objective classification that could be more easily automated.

An Indirect Evaluation Scheme

How do we detect that a mutation is non-equivalent? In practice of mutation testing, this is done all the time: By having the *test suite* detect the mutation. Any mutation detected by the test suite, by definition, alters the program semantics. Thus, it is non-equivalent. This fact that "detected" implies "non-equivalent" leads to our key idea:

> **A mutant generation scheme whose mutants are *more frequently detected* by the test suite also produces *fewer equivalent mutants.***

For mutation testing, we are not interested in detected mutants, though; what we want are non-detected mutants, as these help us to improve the test suite. But if a

scheme *generally* produces fewer equivalent mutants, this property should hold regardless of the detection by the test suite. Hence, the ratio of equivalent mutants can be expected to be lower in the set of undetected mutants as well.

In our case, the mutant generation scheme favors those mutations with impact on invariants. If we can show that impact on invariants correlates with detection by the test suite, this means that impact on invariants also correlates with non-equivalence as non-equivalence is implied by test suite detection. In formal terms, we have an implication

$$\textit{detected by test} \implies \textit{non-equivalent}$$

and if we can show (statistically) that

$$\textit{violates invariants} \stackrel{?}{\implies} \textit{detected by test}$$

then we would conclude that

$$\textit{violates invariants} \implies \textit{non-equivalent}$$

This indirect evaluation approach relies on the assumption that test suites, as they stand, are already good detectors of changed behavior. This assumption was also the base of previous studies [3, 61]; there is no reason to believe that it would not hold for our experiment subjects. (To actually *apply* JAVALANCHE, rather than evaluating it, such a mature test suite is not required; instead, it is our aim to achieve this level of maturity.)

Hypothesis

Given our limited set of mutations, the constrained range of invariants checked and the complexity and richness of the test suites involved, it is not obvious at all that invariant violations would correlate with test outcome. The hypothesis to be checked in our experiment was thus:

H2 *Mutants that violate invariants are more likely to be detected by actual tests.*

Experimental Setup

Our setup for **H2** is straightforward: We execute the test suite on the original program to learn dynamic invariants. Then, we identify mutations that have an invariant impact and check whether they are detected by the test suite.

Table 6.5: Results for **H2**. Invariant-violating mutants (VM) have higher detection rates than non-violating mutants (NVM).

Project name	Number of NVMs	Number of VMs	NVMs detected (%)	VMs detected (%)	p-value χ^2 test
ASPECTJ	1159	13133	61.69	52.18	< 0.0001
BARBECUE	613	200	60.03	89.50	< 0.0001
COMMONS-LANG	10215	967	82.48	86.35	0.0021
JAXEN	2860	1250	44.48	97.84	< 0.0001
JODA-TIME	7523	2122	75.50	90.29	< 0.0001
JTOPAS	566	609	64.13	78.82	< 0.0001
XSTREAM	1835	1765	86.32	97.85	< 0.0001

VM = Invariant-Violating Mutant, NVM = Non-Violating Mutant.

Results

Our results are summarized in Table 6.5. Let us compare the detection rate of invariant-violating mutants (VMs) versus non-violating mutants (NVM) (columns 4 and 5). With the exception of ASPECTJ, all projects show a higher detection rate for VMs. The difference is statistically significant according to the χ^2 test.

The difference can be dramatic: In JAXEN, for instance, 98% of invariant-violating mutants are detected versus 44% of the non-violating mutants. This also means that in JAXEN, the rate of equivalent mutants across all generated invariant violators is not higher than 2%.

Mutants that violate invariants are more likely to be detected by actual tests.

6.4.4 Ranking

For our third experiment, we wanted to explore what was so special about ASPECTJ that the invariant-violating mutations were less likely to be detected than the non-violating ones. In Table 6.5, we see that ASPECTJ has far more violating mutants than all other projects combined. We wanted to *rank* these invariants focusing on those with the

highest impact: If a mutant violates many invariants, it should have a strong impact on the behavior of the program and is, therefore, less likely to be equivalent than mutants that violate fewer invariants.

Hypothesis

In this experiment, we classify not only mutations by whether they violate invariants or not, but we actually rank them according to their invariant impact, which is the number of *different invariants violated.* This is our hypothesis:

H3 *The more invariants a mutant violates, the more likely it is to be detected by actual tests.*

Experimental Setup

Our experimental setup for **H3** is the same as for **H2** discussed in Section 6.4.3, with the exception that we now focus on the detection rate of the top $n\%$ of the invariant-violating mutants, where n ranges from 5 to 100.

Results

Our results can be summarized easily. For ASPECTJ, there is a clear trend: The higher the ranking of a mutation (= the more invariants it violates), the higher its likelihood to be detected by the full test suite. This is shown in Figure 6.6. The x axis shows the subset considered, ranging from 5% (the top 5% of mutations which violated most invariants) to 100% (all mutations that violated at least one invariant). The y axis shows the respective detection rate.

This trend holds for all projects (see Figure 6.6), except for BARBECUE, the project with the lowest number of violating mutants: Just as ASPECTJ, the project with the highest number of violating mutants, benefits from ranking, it is reasonable to assume that the low number of violating invariants in BARBECUE prevents a meaningful ranking. Still, even focusing on the top 5% still yields a higher detection rate than average.

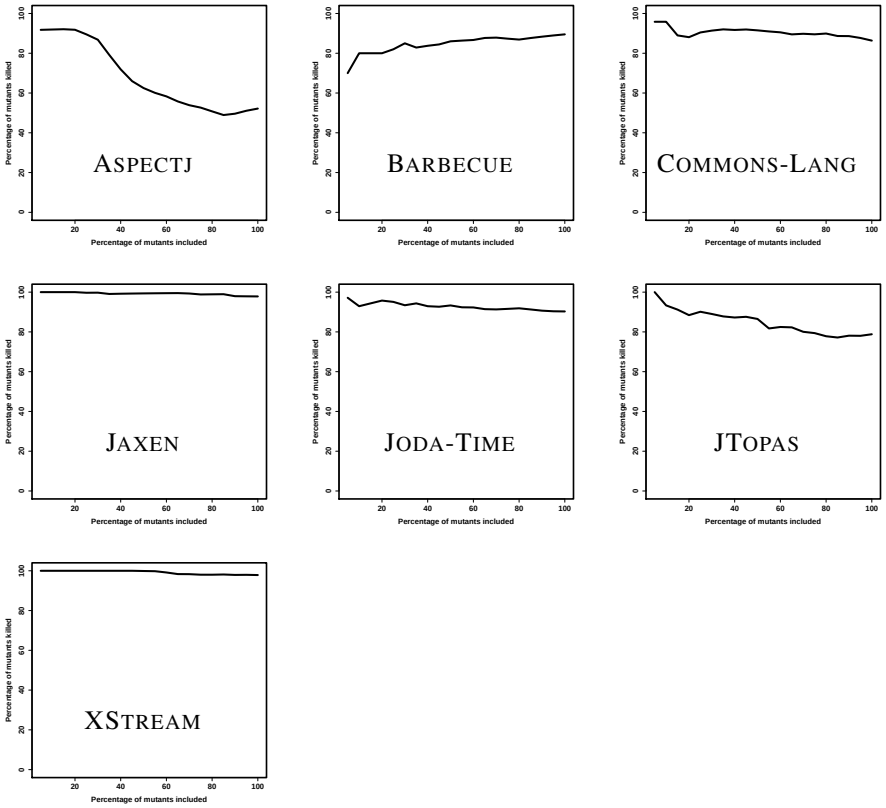

Figure 6.6: Detection rates (y) for the top x% mutations with the highest impact.

In all seven projects, higher-ranked mutations are more likely to be detected than all mutations (violating or non-violating). For all projects except BARBECUE, higher-ranked mutations are always more likely to be detected than the average across all violating mutations.

In Table 6.6, we have shown the detection rate of the 5%, 10%, and 25% violating mutations with the highest impact. Except for BARBECUE, all detection rates are higher than those of 100% non-violating mutations (column five of Table 6.5), thus confirming **H3**.

Table 6.6: Results for **H3**. Best results are obtained by ranking VMs by the number of invariants violated.

Project name	Top 5% VMs detected	Top 10% VMs detected	Top 25% VMs detected
ASPECTJ	91.77	91.93	89.49
BARBECUE	70.00	80.00	82.00
COMMONS-LANG	95.83	95.83	90.46
JAXEN	100.00	100.00	99.68
JODA-TIME	97.17	92.92	95.09
JTOPAS	100.00	93.33	90.13
XSTREAM	100.00	100.00	100.00

VM = Invariant-Violating Mutant, NVM = Non-Violating Mutant.

The more invariants a mutant violates, the more likely it is to be detected by actual tests.

Again, this implies that the undetected high-impact mutants will also have a low rate of equivalent mutants.

6.4.5 Invariant Impact of the Manually Classified Mutants

For our fourth experiment, we were interested if the invariant impact can be used as a predictor for the equivalence or non-equivalence of undetected mutants. We have seen that mutations with a high impact are very likely to be non-equivalent. Another question is how many of the undetected mutants are correctly classified using the invariant impact.

Hypothesis

In order to use the invariant impact as a predictor, it has to be *precise* which means that the impact on invariants should indicate non-equivalence, and it has to be *sensitive* which means that non-equivalent mutations should have an impact on invariants. This leads us to the following hypothesis:

Table 6.7: Results for **H4**. Precision and recall of the invariant impact for the 140 manually classified mutants.

Project name	Precision %	Recall %
ASPECTJ	100	7
BARBECUE	75	43
COMMONS-LANG	50	17
JAXEN	50	10
JODA-TIME	100	21
JTOPAS	100	10
XSTREAM	40	25
Total	**68**	**19**

H4 *Most non-equivalent and undetected mutants have an impact on invariants, while equivalent mutants do not have an impact.*

Experiment

As discussed earlier, classifying all undetected mutants would be an enormous effort. Therefore, we limited this experiment to the 140 previously classified mutants (see Section 5.2). For the classified mutants, we computed their invariant impact. Then, we checked how many of the mutants with impact were actually non-equivalent (precision), and we computed how many of the non-equivalent mutants actually had an invariant impact (recall).

Results

Table 6.7 shows the results for the 140 manually classified mutants. The first column gives the project name, the second column (precision) displays the percentage of mutations that are non-equivalent among all mutations with impact, and the last column (recall) gives the percentage of mutations that have impact among all non-equivalent mutations. On average, 68% of the mutations with impact are correctly classified as non-equivalent. This number ranges from 40% for XSTREAM up to 100% for ASPECTJ, JODA-TIME, and JTOPAS. However, the average recall is only about 19%

which indicates a weakness of our technique. Although it is very likely that a mutation is non-equivalent when it violates multiple invariants and still likely when it violates at least one invariant, we cannot use the invariant impact as a predictor for non-equivalence since many non-equivalent mutants do not have an impact on invariants.

Many non-equivalent mutants do not have an impact on invariants.

6.4.6 Discussion

The results of our case study suggest that developers should not only focus on those mutations that violate invariants, but that they should actually focus on those mutations that violate most invariants.

Whenever mutation testing results in a large number of undetected mutants, it thus seems a good idea to *prioritize* the mutants by their impact on invariants:

1. By focusing on those mutants with the highest impact on invariants, one creates a *bias* towards non-equivalence. This is good, as this minimizes the number of equivalent mutants to deal with.

2. As these invariants are originally learned from test suite executions, this implies a bias towards mutations whose induced behavior *differs most* from the "normal" behavior as characterized by the test suite.

3. Focusing on high-impact mutants also implies focusing on those areas where a defect can create *most damage* across the program execution. Again, we consider such a focus a very valuable property.

Our results also indicate that it is generally useful to focus on those mutations that violate most invariants. Even in a program with few violating mutants like BARBECUE, which did not benefit much from ranking (see the discussion in Section 6.4.4), the top-ranked mutations consistently yielded better detection results than the average mutation.

Finally, our results also place an upper bound on the number of equivalent mutants. Omitting BARBECUE due to its low number of violating mutants, the detection rate for the top 5% of violating mutations (Figure 6.6) is 92 to 100%, with an average of 97%.

Thus, only 3% of these high-impact mutations were undetected placing an upper bound on the number of equivalent (undetectable) mutants. (Note that in Section 6.4.2, only 17% of this small high-impact set were actually found to be equivalent, suggesting an even lower overall rate.) This very low rate is what makes our approach to mutation testing efficient.

On average, focusing on the top 5% of invariant-violating mutants yields ***less than 3% of equivalent mutants.***

6.5 Threats to Validity

Like any empirical study, this study has limitations that must be considered when interpreting its results.

External validity The results of **H1** should be considered as promising but not generalizable as the sample is small. Similarly, we cannot claim that the results of **H4** for the 140 manually classified mutants are generalizable. The manual effort generally stands in the way of larger studies. Regarding **H2** and **H3**, we evaluated our approach on seven programs with different application domains and sizes; some of them were larger by several orders of magnitude than programs previously used for evaluation of mutation testing [70, 27, 39, 3]. Generally, our results were consistent across a wide range of programs. Still, there is a wide range of factors with regard to both programs and test suites that may impact the results, and we, therefore, cannot claim that the results would be generalizable to other projects. Prospective users are advised to conduct a retrospective study like ours.

Internal validity Regarding **H1** and **H4**, our own assessment may be subject to errors, incompetence, or bias; to counter these threats, all our assessments are publicly available[1]. For **H2** and **H3**, our implementation could contain errors that affect the outcome. To control for these threats, we ensured that earlier stages (Figure 6.1) had no access to data used in later stages. Our statistical evaluation was conducted using textbook techniques implemented in widely used frameworks.

[1] see http://www.st.cs.uni-saarland.de/mutation

Construct validity Regarding **H1** and **H4**, being able to write a test can be seen as the ultimate measure whether a mutant is non-equivalent. In **H2** and **H3**, our assumption that the test suite measures real defects is an instance of the "competent programmer hypothesis" also underlying mutation testing [18]. This hypothesis may be wrong; however, the maturity and widespread usage of the subject programs suggest anything but incompetence. Further studies will help completing our knowledge on what makes a test suite adequate.

6.6 Related Work

6.6.1 Mutation Testing

Assessing the state to check the impact of mutations is also related to the concept of *weak mutation testing,* as proposed by Howden [40] and explained in Section 3.3.4. Weak mutation testing assesses the effect of a mutation by determining the state after its execution: If the state is different, then the mutation is detectable. Weak mutation testing thus checks whether the test suite could *possibly* detect a mutation; it does not matter whether the tests actually pass or not. Strong mutation testing, which is what we assume, assesses the test suite by determining whether it *actually* detects a mutation. Our approach also measures state changes. However, the knowledge of state changes, in the form of invariant violations, is used to compute the impact of a mutant instead of deciding whether the mutant is detected or not.

6.6.2 Equivalent Mutants

The issue of equivalent mutants has frustrated generations of mutation testers. In Section 5.3, we have quoted Frankl et al. [27] on the enormous amount of work needed to eliminate equivalent mutants. A number of researchers have tackled the problem of detecting equivalent mutants. Baldwin and Sayward were the first ones to suggest *heuristics* for detecting equivalent mutants. Their approach is based on detecting idioms from semantics-preserving compiler optimizations. Later, it was shown by Offutt and Craft [68] that this approach detects approximately 10% of all equivalent mutants.

In 1996, Offutt and Pan [70] realized that detecting equivalent mutants is an instance of the *infeasible path* problem which also occurs in other testing techniques.

They presented an approach based on solving *path conditions* that originate from a mutant. If the constraint solver can show that all subsequent states are equivalent, the mutant is deemed equivalent. The technique was reported to detect 48% of equivalent mutants. A similar approach, based on *program slicing,* was presented by Hierons and Harman [39]; this approach additionally provides guidance in detecting the locations that are potentially affected by a mutant. Modern change impact analysis [86] can do this in the presence of subtyping and dynamic dispatch. The recent concept of *differential symbolic execution* [80] brings the promise of easily detecting potential impact of changes.

All of these techniques are orthogonal to ours; indeed, if we can prove statically that a mutation will have no impact, we can effectively omit the runtime tests. The question is how well these static approaches scale up when it comes to detecting mutant equivalence in real programs. Offutt and Pan's [70] technique, for instance, was evaluated on eleven Fortran 77 programs which "range in size from about 11 to 30 executable statements". In contrast, the programs we have been looking at are larger by several orders of magnitude.

6.6.3 Invariants and Contracts

The idea of *checking* the program state at runtime is as old as programming itself. *Design by contract* [59] mandates specifying invariants for every public method in a program; the resulting runtime checkers effectively catch errors at the moment they originate.

If the programmer does not provide invariants, one can *infer* them from program runs. This is the idea of dynamic invariants as realized in the DAIKON tool by Ernst et al. [23]. Most related to our work is the ECLAT tool by Pacheco and Ernst [77] which selects, from a set of test inputs, a subset that is most likely to reveal defects by assessing the impact of the inputs on dynamic invariants. McCamant and Ernst [58] use dynamic invariants to check whether invariants had changed after a code change—indicating a potential problem in the future. (Our approach, of course, explicitly looks for such invariant changes.)

Sosič and Abramson [93] suggest another technique to detect the impact of changes to programs. The idea of *relative debugging* is to compare the execution of two programs (in our setting, the original vs. the mutant) and automatically report any differences in variable values. While the differences do not translate into invariants, they could nonetheless serve as impact indicators.

Our concept of efficient dynamic invariant checking was inspired by the DIDUCE tool by Hangal and Lam [36], flagging invariant violations as they occur during the run. Neither approach discussed in this section was applied to mutation testing so far.

6.7 Summary

In this chapter, we explained how to assess the impact of mutations using dynamic invariants. To this end, we presented a scalable and efficient method to learn the most common invariants (Section 6.1) and to check them (Section 6.2). In order to learn the invariants, a run of the test suite on the original program is traced with ADABU. From the resulting trace, dynamic invariants are computed using DAIKON. During mutation testing, JAVALANCHE then inserts code that checks for invariant violations at runtime. In the evaluation of the approach, we have seen that we can apply this technique to real-world programs with reasonable efficiency, and that mutations with impact on invariants are less likely to be equivalent. Furthermore, we have seen that mutants violating most invariants are even less likely to be equivalent. However, our results also showed that the invariant impact cannot be used as a predictor for the non-equivalence of a mutant because there are also many non-equivalent mutants that do not have an impact on the invariants. When improving test suites, we therefore, suggest that test managers should focus on those surviving mutations that have the greatest impact on invariants.

Chapter 7

Coverage and Data Impact of Mutations

The results of the previous chapter showed that a mutation that has an impact on invariants is very likely to be non-equivalent. However, we have also seen that mutations violating dynamic invariants are rare (Section 6.4.5). This finding motivated us to develop a measure that provides a more *fine-grained* view on the impact. A mutant can have an impact on the program run by changing the *control flow* of the run or by changing the *data* that is processed. The control flow refers to the order in which the individual statements in a program are executed. By manipulating a control flow statement, a mutation can change the order or cause different statements to be executed. The data refers to the program state which includes all values that can be accessed at a specific point of the program run. By manipulating statements involving the computation or storage of data, a mutation can manipulate the program state. Both changes can propagate throughout a program run and result in a different result. Furthermore, a change in the control flow can also result in changes to the data, and vice versa, changed data can cause a differing control flow.

In this chapter, we present an approach that measures the impact on the control flow by checking for differences in the execution frequency of individual statements and the impact on data by comparing the return values of public methods.

7.1 Assessing Mutation Impact

Equivalent mutants are defined to have no observable impact on the program's output. This impact of a mutation can be assessed by checking the program state at the end of a computation, just like tests do. However, we can also assess the impact of a mutation *while the computation is being performed.* In particular, we can measure *changes in program behavior* between the mutant and the original version. The idea is that if a mutant impacts internal program behavior, it is more likely to change external program behavior. Thus, it is also more likely to impact the semantics of the program. If we focus on *mutations with impact,* we would thus expect to find fewer equivalent mutants.

How does one measure impact? *Weak mutation* [40] assesses whether a mutation changes the *local state* of a function or a component; if it does, it is considered detectable (and, therefore, non-equivalent). In this work, we are taking a more *global* stance and examine how the impact of a mutation propagates all across the system. To assess this impact degree, we consider two aspects:

- One aspect of impact is *control flow:* If a mutation alters the control flow of the execution, different statements are executed in a different order. This is an impact that can be detected by using standard coverage measurement techniques.
- Another aspect of the behavior concerns the *data* that is passed between methods during the computation: If a mutation alters the data, different values are passed between methods. This is an impact that can be detected by tracing the data that is passed between methods.

In both cases, we measure the impact as *the number of changes detected all across the system;* as the number of impacted methods grows, so does the likelihood of the mutation to be generally detectable—and non-equivalent.

7.1.1 Impact on Coverage

In order to measure the impact of mutations on the control flow, we developed a tool that computes the code coverage of a program and integrated it into the JAVALANCHE framework. The program records the execution frequency for each statement that is executed for each test case and each mutation. Note that the data collected by our tool

is very similar to *statement coverage* which computes whether a statement is executed or not. In addition to statement coverage, our tool also stores the execution frequency of a statement.

Running the complete test suite of a program and tracing its coverage provides us with a set of lines that were covered together with frequency counts for every test case of the test suite. By comparing the coverage of a run of the original version with the coverage of the mutated version, we can determine the *coverage difference.*

7.1.2 Impact on Return Values

Mutations with impact on the control flow manifest themselves in coverage differences, but it is also possible that a mutation has only impact on the data, which is not used in control flow affecting computations. In a manual investigation of random undetected mutations, we found two categories of non-equivalent mutations that had no impact on the code coverage:

- The first category are mutations that *changed return values* that were subsequently just passed around.
- The second category are mutations causing state changes that only manifest in a *change of the string representation* of an object.

Therefore, we decided to additionally trace the return values of public methods. We chose the public methods as they represent an object's communication to the environment.

Storing all return values of a program run would require a huge amount of disk space. For example, objects can cover huge parts of the program state through references. The storage of all this data for each return value might be justifiable for one run of a test suite. As we plan to use this data for assessing each mutation, which involves several thousand executions of the test suite, we decided to abstract each return value into an integer value.

For each public method that has a return value, we store these integers and count how often they occur. In this way, we end up with a set of integers for each method together with frequency counts. Similar to coverage data, we can compare the sets of traced return values of the original execution with the mutated execution and obtain the *data difference*.

Abstracting Return Values

To obtain an integer value for returned JAVA objects, we compute its string representation by invoking `toString()`. Then, we remove substrings that represent memory locations, as returned by the standard implementation of the `toString()` method in `java.lang.Object`, because these locations change between different runs of the program even though the computed data stays the same. From the resulting string, we then compute the hash code. Thereby, we obtain an integer value that characterizes the object.

For each primitive value (`int`, `char`, `float`, `short`, `boolean`, `byte`), we store its natural integer representation; for 64-bit values (`long`,`double`), we compute the exclusive or of the upper and lower 32 bits.

7.1.3 Impact Metrics

The techniques defined above produce a set of *differences* between a run of the test suite on the original and mutated program. Using these differences, we define *impact metrics* that quantify the difference between the original and mutated run:

Coverage impact —the *number of methods* that have at least one statement that is executed at a different frequency in the mutated run than in the normal run—while leaving out the method that contains the mutation.

Data impact —the *number of methods* that have at least one different return value or frequency in the mutated run than in the normal run—while leaving out the method that contains the mutation.

Combined coverage and data impact —the *number of methods* that either have a coverage or data impact.

Definition 29 (Coverage Impacted Methods) *For a mutant S_M of a program S and a set of tests T, the set of methods with coverage impact $\mathscr{S}_{CI}$ is the set of methods that have at least one statement with a different execution frequency between a run of T on S and a run of T on S_M, excluding the method m where the mutation was applied.*

Definition 30 (Coverage Impact) *The coverage impact CI of a mutant S_M of a program S for a set of tests T, is the cardinality of the set of methods with coverage impact $\mathscr{S}_{CI}$.*

$$CI(S_M, S, T) = |\mathscr{S}_{CI}|$$

Definition 31 (Data Impacted Methods) *For a mutant S_M of a program S and a set of tests T, the set of methods with data impact $\mathscr{S}_{DI}$ is the set of methods that have at least one different return value between a run of T on S and a run of T on S_M, excluding the method m where the mutation was applied.*

Definition 32 (Data Impact) *The data impact DI of a mutant S_M of a program S for a set of tests T, is the cardinality of the set of methods with data impact $\mathscr{S}_{DI}$.*

$$DI(S_M, S, T) = |\mathscr{S}_{DI}|$$

Definition 33 (Combined Impact) *The combined impact CombI of a mutant S_M of a program S for a set of tests T, is the cardinality of the union of the set of methods with coverage impact $\mathscr{S}_{CI}$ and the set of methods with data impact $\mathscr{S}_{DI}$.*

$$CombI(S_M, S, T) = |\mathscr{S}_{CI} \cup \mathscr{S}_{DI}|$$

These metrics are motivated by the hypothesis that a mutation that has *non-local impact* on the program is more likely to change the observable behavior of the program. Furthermore, we would assume that mutations which are undetected despite having an impact across several methods can be considered as particularly valuable for the improvement of a test suite, as they indicate inadequate testing of multiple methods at once.

7.1.4 Distance Metrics

To further emphasize non-local impact, we use *distance metrics* that are based on the distance between the method that contains the mutation and the method that has a coverage or data difference.

The distance between two methods is the length of the shortest path between them in the *undirected call graph*. The undirected call graph is a variant of the traditional

call graph that contains a node V_M for each method M in the program. There is an edge between two nodes V_M and V_N if there exists a call from method M to N or vice versa.

By using this distance, we can define three distance metrics analogous to the impact metrics defined above:

- **Coverage distance**—For each method that has a coverage difference, we compute the shortest path to the method that contains the mutation. The coverage distance is then the length of the longest path.
- **Data distance**—For each method that has a data difference, we compute the shortest path to the method that contains the mutation. The data distance is then the length of the longest path.
- **Combined coverage and data distance**—the maximum of the data and coverage distance.

Definition 34 (Distance) *For the mutant S_M that was applied to method m of the program S, a set of impacted methods $\mathscr{S}$, and a distance function sp that computes the shortest path between two methods in the undirected call graph, the coverage distance is the maximum distance between the method that contains the mutation and one of the impacted methods.*

$$Dist(\mathscr{S}) = max(sp(m,x)),\ x \in \mathscr{S}$$

With this distance definition, the distance metrics are defined by using different sets of impacted methods. The set of methods with coverage impact $\mathscr{S}_{CI}$, as introduced in Definition 29, defines the coverage distance. The set of methods with data impact $\mathscr{S}_{DI}$, as introduced in Definition 31, defines the data distance. The union of both sets defines the combined distance.

7.1.5 Equivalence Thresholds

Each of the metrics defined above (Sections 7.1.4 and 7.1.3) produces a natural number that describes the impact of a mutation. As we want to automatically classify mutants that are less likely to be equivalent, we introduce a threshold t. A mutant is considered to have an impact if and only if its impact metric is greater than or equal to t.

This allows us to split the set of mutants into the set of *mutants with impact* (MI) and the set of *mutants with no impact* (MNI).

Definition 35 (Mutants with impact (MI)) *For an impact metric I_x, a test set T, and a threshold t, the set of mutations with impact $\mathscr{M}_I$ is the set of mutations where the impact is greater than or equals t.*

$$\mathscr{M}_I = \{S_M^i \mid I_x(S_M^i, S, T) \geq t\}$$

Definition 36 (Mutants with no impact (MNI)) *For an impact metric I_x, a test set T, and a threshold t, the set of mutations with no impact $\mathscr{M}_{NI}$ is the set of mutations where the impact is less than t.*

$$\mathscr{M}_{NI} = \{S_M^i \mid I_x(S_M^i, S, T) < t\}$$

7.2 Evaluation

We evaluated our approach in three experiments. First, in Section 7.2.1, we applied our techniques to automatically classify mutants to the 140 manually classified mutants from Section 5.2. For our second experiment, we devised an evaluation scheme based on mature test suites. This automated evaluation scheme is presented in Section 7.2.2 and compares the detection rate of mutants with impact and mutants with no impact. Finally, we were interested if the mutants with the highest impact are less likely to be equivalent. We, therefore, ranked the mutants according to their impact and looked at the highest ranked mutants (Section 7.2.3), both for the manually classified and the ones detected by the test suites.

7.2.1 Impact of the Manually Classified Mutations

In the first experiment, we wanted to evaluate our hypothesis that mutants with impact on coverage or return values are less likely to be equivalent. We, therefore, determined the coverage and data differences and computed the impact (Section 7.1.3) and the distance metrics (Section 7.1.4) for the 140 manually classified mutants, and automatically classified them by using a threshold of 1 for all metrics. Then, we compared these results to the actual results of the manual classification.

To quantify the effectiveness of the classification, we computed its *precision* and *recall*:

Table 7.1: Effectiveness of classifying mutants by impact: precision and recall.

Project name	Coverage impact	Data impact	Combined impact	Invariant impact
ASPECTJ	72 / 87	72 / 87	72 / 87	100 / 7
BARBECUE	100 / 43	100 / 29	100 / 43	75 / 43
COMMONS-LANG	0 / 0	0 / 0	0 / 0	50 / 17
JAXEN	67 / 60	78 / 70	73 / 80	50 / 10
JODA-TIME	90 / 64	89 / 57	91 / 71	100 / 21
JTOPAS	100 / 70	43 / 30	64 / 70	100 / 10
XSTREAM	50 / 25	67 / 25	60 / 38	40 / 25
Total	**75 / 56**	**67 / 48**	**70 / 61**	**68 / 19**

First value in a cell gives the precision, the second the recall.

Table 7.2: Effectiveness of classifying mutants by distance based impact.

Project name	Coverage distance	Data distance	Combined distance
ASPECTJ	77 / 67	67 / 67	67 / 67
BARBECUE	100 / 43	100 / 29	100 / 43
COMMONS-LANG	0 / 0	0 / 0	0 / 0
JAXEN	67 / 60	78 / 70	73 / 80
JODA-TIME	90 / 64	89 / 57	91 / 71
JTOPAS	100 / 60	50 / 30	67 / 60
XSTREAM	50 / 13	67 / 25	67 / 25
Total	**79 / 49**	**68 / 44**	**71 / 55**

First value in a cell gives the precision, the second the recall.

- The *precision* is the percentage of mutants that are correctly classified as non-equivalent, i.e. the mutant has an impact and is non-equivalent. A high precision implies that the results of a classification scheme contain *few false positives*—that is, most mutants classified as non-equivalent are indeed non-equivalent.

- The *recall* is the percentage of non-equivalent mutants that are correctly classified as such. A high recall means that there are *few false negatives*—that is, a high ratio of the non-equivalent mutants was retrieved by the classification

scheme.

While it is easy to achieve a 100% recall (just classify all mutants as non-equivalent), the challenge is to achieve both a high precision and a high recall.

The results of the evaluation of the different metrics on the classified mutants are summarized in Table 7.1 and Table 7.2 . Each entry gives first the precision of the metric, and then its recall. Table 7.1 displays the results for coverage impact, data impact, and their combined impact as defined above. It also contains the results for the impact on dynamic invariants as defined in Chapter 6. Table 7.2 gives the results for the distance metrics.

When considering the average results (last row), we can see that all techniques have a high *precision*, ranging from 68% for the data distance (column 3 of Table 7.2) and invariant metric (last column of Table 7.1) up to 79% for coverage distance (column 2 of Table 7.2) . This means that 68% to 79% of all mutants classified as non-equivalent actually are non-equivalent. In comparison, a simple classifier that classifies all mutations as non-equivalent, would have a precision of 54%. Thus, the metrics improve over the simple approach by 14 to 25 percentage points.

Mutations with impact on coverage and data have a likelihood of 68 to 79% to be non-equivalent, compared to 54% across all mutations.

When we look at the results of each project, COMMONS-LANG is a clear outlier, with a precision and recall of zero for almost all metrics. This is due to several mutations that alter the *caching behavior* of some methods. Although they are manually classified as equivalent because the methods still return a correct object, they have a huge impact because new objects are created at every call instead of taking them from the cache. When we look at the result of the manual classification for COMMONS-LANG (Table 5.1), we also see that it is the project with the highest number of equivalent mutants—which might indicate that most mutations that are not detected by the test suite are equivalent.

The recall values for the coverage and data metrics range from 44% for data distance to 61% for the combined impact metric. Both the combined impact and the combined distance metric have a higher recall than the two metrics they are based on. This, however, comes at the cost of lower precision. Furthermore, all coverage and data metrics also have a far better recall than the earlier invariant-based technique which has a recall of only 19%.

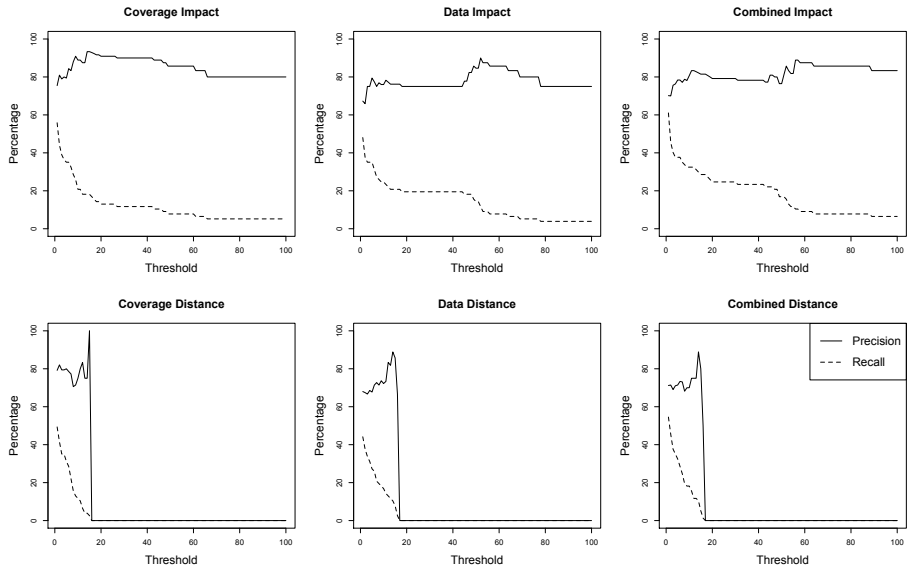

Figure 7.1: Precision and Recall of the impact metrics for different thresholds.

Coverage and data impact have better recall values than invariant impact.

There is always a trade-off between precision and recall. Increasing one of both values decreases the other one. The simple classifier, for example, has a recall of 100% by definition while it only has a precision of 55%. On the other hand, we can also increase the precision of our metrics by increasing the threshold, e.g. when we use a threshold of 2 for the coverage impact, we get a precision of 81% and a recall of 44%.

All distance metrics have a lower recall than their corresponding impact metrics. A reason for this is that some mutations impact methods that are not connected via method calls. In these cases, the impact propagates through state changes.

Sensitivity Analysis

The previous results were all computed with a threshold of 1. Thus, it is not clear how the metrics perform when a higher threshold is used. In order to analyze its influence,

Table 7.3: Assessing whether mutants with impact on coverage are detected by tests.

Project name	Number of MIs	Number of MNIs	MIs detected	MNIs detected
ASPECTJ	5,531	1,661	76%	20%
BARBECUE	1,045	528	83%	32%
COMMONS-LANG	10,061	4,559	97%	58%
JAXEN	5,997	548	97%	26%
JODA-TIME	15,883	2,037	95%	18%
JTOPAS	1,362	150	93%	5%
XSTREAM	5,940	788	97%	39%

MI = Mutation with Impact, MNI = Mutation with No Impact.

we repeated the previous experiment with varying thresholds. Figure 7.1 shows a graph for each distance metric. The x-axis displays the threshold and the y-axis shows the percentage for precision and recall. A continuous line stands for the precision whereas a dashed line stands for the recall. For a higher threshold, we would expect higher precision as only the mutations with a higher impact are considered. This comes at the cost of a lower recall as fewer mutations are considered in total. By looking at the graphs, we can see that all metrics follow this trend. For the distance based metrics, however, this trend only holds up to a threshold of 15. For thresholds greater than 15, both recall and precision are 0 since our sample contains no non-equivalent mutation with a distance impact greater than 15. In contrast to the distance based metrics, the trend holds for the coverage, data, and combined impact up to a threshold of 100.

Coverage and data metrics are more stable for higher thresholds than distance based metrics.

7.2.2 Impact and Tests

Besides our evaluation on the manually classified mutations, we also wanted a broader objective evaluation scheme that can be automated. However, in order to automatically determine the equivalence of a mutation, we either need a test suite that detects all non-equivalent mutations, or an oracle that tells the equivalence of a mutation. Unfortunately, obtaining such a test suite or an oracle is infeasible. Thus, we decided to base our automated evaluation scheme on the existing mature test suites of the projects.

Table 7.4: Results for ranking the mutations according to their impact on coverage.

Project name	Top 5% MIs detected	Top 10% MIs detected	Top 25% MIs detected
ASPECTJ	100%	100%	99%
BARBECUE	100%	97%	99%
COMMONS-LANG	98%	99%	99%
JAXEN	100%	100%	100%
JODA-TIME	100%	100%	99%
JTOPAS	100%	100%	100%
XSTREAM	100%	100%	100%

Table 7.5: Assessing whether mutants with impact on data are detected by tests.

Project name	Number of MIs	Number of MNIs	MIs detected	MNIs detected
ASPECTJ	5,186	2,006	80%	19%
BARBECUE	956	617	92%	25%
COMMONS-LANG	7,861	6,759	98%	70%
JAXEN	6,005	540	95%	46%
JODA-TIME	15,173	2,747	91%	55%
JTOPAS	1,286	226	94%	31%
XSTREAM	5,543	1,185	95%	64%

The rationale for our evaluation is as follows: A mutation classification scheme helps the programmer when it detects many non-equivalent and fewer equivalent mutants. For every mutant that is detected by the test suite, we know for sure that it is non-equivalent. If we can prove that a classification scheme has a high precision on the mutations that are detected by the test suite, this might also hold for the mutations that are not detected by the test suite.

Thus, we applied the impact metrics to all mutations in each project and evaluated them on the mutations detected by the test suite. The results are given in Tables 7.3 to 7.8.

For each project and impact metric, we determined the number of mutations that had an impact (MIs in column 2), and the number that had no impact (MNIs in column

Table 7.6: Results for ranking the mutations by their impact on data.

Project name	Top 5% MIs detected	Top 10% MIs detected	Top 25% MIs detected
ASPECTJ	100%	99%	99%
BARBECUE	100%	97%	99%
COMMONS-LANG	97%	98%	98%
JAXEN	100%	100%	100%
JODA-TIME	100%	100%	99%
JTOPAS	100%	100%	100%
XSTREAM	100%	100%	100%

Table 7.7: Assessing whether mutants with combined coverage and data impact are detected by tests.

Project name	Number of MIs	Number of MNIs	MIs detected	MNIs detected
ASPECTJ	5,200	1,992	81%	17%
BARBECUE	1,142	431	81%	25%
COMMONS-LANG	10,467	4,153	95%	59%
JAXEN	6,063	482	95%	41%
JODA-TIME	15,841	2,079	91%	43%
JTOPAS	1,388	124	92%	6%
XSTREAM	6,059	669	94%	52%

3). For the MIs and MNIs, we then computed the ratio that was detected by the test suite (column 4 and 5).

In Section 7.1.5, we saw that we need a *threshold t* when to consider a mutation to have an impact according to the underlying metric. As our manual classification showed 45% of the undetected mutations to be equivalent, we automatically set t so that at most 45% of the undetected mutations are classified as having no impact.

The ratio of mutations with impact ranges from 54% for COMMONS-LANG and data impact (Table 7.5) up to 93% for JAXEN and the combined impact metric (Table 7.7). The number of mutations with impact that are detected is around 90% on average (i.e. at most 10% are equivalent) while the average ratio of mutations with

Table 7.8: Results for ranking the mutations by their impact on coverage and data.

Project name	Top 5% MIs detected	Top 10% MIs detected	Top 25% MIs detected
ASPECTJ	100%	100%	99%
BARBECUE	100%	97%	99%
COMMONS-LANG	98%	98%	99%
JAXEN	100%	100%	100%
JODA-TIME	100%	100%	99%
JTOPAS	100%	100%	100%
XSTREAM	100%	100%	100%

no impact ranges from 28% for coverage impact to 45% for data and combined impact. These results indicate that the impact metrics classify the mutations with a high precision while the coverage impact metric has the highest precision.

Of the mutations that have impact on coverage or data, at most 10% are equivalent.

Sensitivity Analysis

We investigated the sensitivity of our approach in relation to the threshold by repeating the previous experiment for the coverage impact measure and varying values for the threshold. The results are shown in Figure 7.2. For each of the 7 projects, there is one graph where the x-axis gives the different threshold values and the y-axis the percentage for 3 different measurements: (1) a solid line for the mutations with impact, (2) a dashed line for the ratio between detected and undetected mutations with impact, and (3) a dotted line for the ratio between detected and not detected mutations with no impact.

The percentage of mutations with impact declines for all projects, and for BARBECUE, COMMONS-LANG, and JTOPAS there are no mutations with an impact greater than 200 while for JAXEN, JODA-TIME, and XSTREAM, there are some mutations with an impact greater than 800. The ratio of detected mutations with impact rises up to 100% for all mutations when higher thresholds are used. This is because only mutations with a big impact (mostly mutations that cause exceptions) are considered.

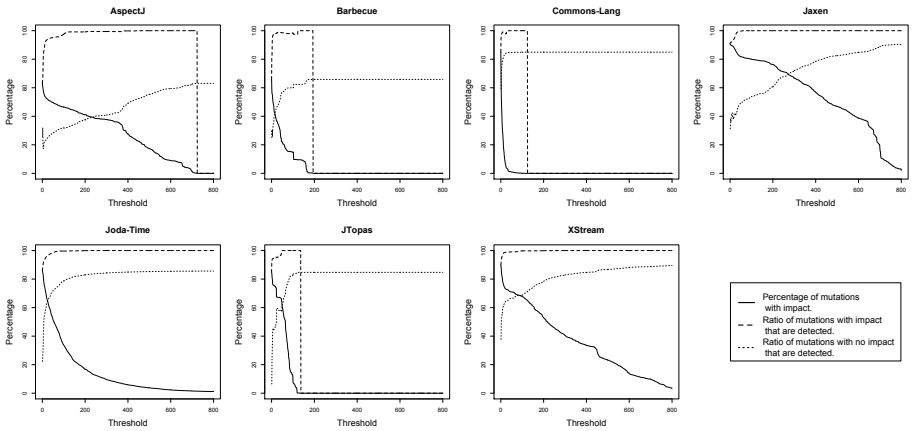

Figure 7.2: Percentage of mutations with impact and detection ratios of mutations with and without impact for varying threshold.

The ratio of detected mutations with no impact also rises for higher thresholds, because more mutations are considered to have no impact.

At higher thresholds 100% of the mutations with impact are detected.

Mutation Operators

As the results from the manual classification indicated that there is a difference in ratio between equivalent and non-equivalent mutations for different mutation operators (Section 5.2.3), we were interested in the detection ratios for mutation operators. To this end, we grouped the results for combined coverage and data impact by mutation operator and combined the results for all projects. Table 7.9 summarizes the results. The number of mutations (column 2) that are produced for an operator ranges from 2,359 for *replace arithmetic operator* to 22,457 for *replace numerical constant*. Similar to the results concerning equivalence (Section 5.2.3), there is a difference between operators that manipulate the control flow (*negate jump condition* and *omit method call*) and operators that manipulate data (*replace numerical constant* and *replace arithmetic operator*).

Table 7.9: Detection ratios for different operators

Mutation operator	Number of mutants	Detected mutants	Mutant with impact (MI)	MI detected
Replace numerical constant	22,457	76.96	85.08	83.59
Negate jump condition	9,790	90.41	91.65	90.69
Replace arithmetic operator	2,359	80.63	86.90	87.95
Omit method call	21,484	86.55	94.42	88.11
Total	**56,090**	**83.14**	**89.88**	**86.85**

Compared to the data manipulating operators, the control flow manipulating operators have higher detection rates (87 to 90% vs. 77 to 81% in column 3), more mutations with impact (92 to 94% vs. 85 to 87% in column 4), and higher detection rates for mutations with impact (88 to 91% vs. 84 to 88% in column 5).

Mutations that manipulate the control flow have higher impact and higher detection rates than mutations that manipulate data.

7.2.3 Mutations with High Impact

In the previous experiments, we saw that mutations with impact are more likely to be non-equivalent. Besides that, we were interested in whether mutations with a *high impact* are more likely to be non-equivalent.

To evaluate this hypothesis, we did two experiments. First, we ranked the mutations that were detected (as described in Section 7.2.2) by their impact, picked the top 5, 10, and 25%, and checked how many of them were non-equivalent. In a second experiment, we ranked the mutations from the manual classification according to their impact for the different impact metrics. Then, we picked the 15, 20, and 25% of the highest ranked mutations out of all mutations that were classified as non-equivalent by the metric, and checked if they were correctly classified.

The results for the first experiment (for mutations detected by the test suite) can be found in the last 3 columns of Tables 7.3 to 7.7. For many projects and impact metrics the 25% of mutations with the highest impact are *all* detected. If not all of them are detected, at least 98% of them are. For the impact on invariants (see Chapter 6), we

Table 7.10: Focusing on mutations with the highest impact: precision of the classification

Impact metric	Top 15%	Top 20%	Top 25%
Coverage impact	88%	91%	93%
Data impact	88%	91%	86%
Combined impact	90%	85%	76%
Coverage distance	86%	80%	75%
Data distance	88%	80%	85%
Combined distance	89%	75%	80%

observed a similar trend but not as distinctive as for the data and coverage impact metrics.

Table 7.10 shows the results for the manual classification. For all impact metrics, 75% or more out of the top 25% are non-equivalent. Compared to the precision results in Table 7.1, picking the 25% of mutations with the highest impact attains a higher ratio of non-equivalent mutations than choosing mutations with impact in no specific order. In this setting, the *coverage impact* metric performs best again. When we choose the top 25% ranked mutations, 93% of them are non-equivalent.

More than 90% of the mutations with the highest coverage impact are non-equivalent.

The results for the detected mutants indicate that a high impact strongly correlates with non-equivalence, and the results for the manually classified mutations confirm this finding also for the undetected mutants.

In practice, this means that focusing on the mutations with the highest impact yields the fewest amount of equivalent mutants. The question is whether mutations with a high impact are also the most *valuable* mutations—that is, whether they uncover most errors, or at least the most important errors. Our intuition tells us that if we can make a change to a component that impacts several other components, while the test suite still does not detect it, such a change has a higher chance to be valuable than a change whose impact is hardly measurable. The relationship between impact and value of mutations remains to be assessed and quantified, though.

7.3 Threats to Validity

The interpretation of our results is subject to the following threats to validity.

External validity In our studies, we have examined 20 sample mutations from seven non-trivial JAVA programs with different application domains and sizes; some of them were larger by several orders of magnitude than programs previously used for evaluation of mutation testing [70, 27, 39, 3]. Generally, our results were consistent across a wide range of programs. Still, there is a wide range of factors of both programs and test suites that may impact the results, and we, therefore, cannot claim that the results would be generalizable to other projects.

Internal validity Regarding the manual classification (Section 5.2), our own assessment may be subject to errors, incompetence, or bias. At the time we conducted the assessment, we did not know how the mutations would score in terms of impact. For assessing mutations based on coverage (Sections 7.2.1 and 7.2.2), our implementation could contain errors that affect the outcome. To control for these threats, we ensured that earlier stages had no access to data used in later stages.

Construct validity Regarding the manual classification of mutations (Section 5.2), being able to write a test is the ultimate measure whether a mutant is non-equivalent. When classifying mutations based on impact (Section 7.2.1), we directly provide the information as required by the programmer. Finally, in Section 7.2.2, our assumption that the test suite measures real defects is an instance of the "competent programmer hypothesis" also underlying mutation testing [18]. This hypothesis may be wrong; however, the maturity and widespread usage of the subject programs should suggest sufficient competence. Further studies will help completing our knowledge on what makes a test suite adequate.

7.4 Related Work

As stated in Section 6.6.2, the problem of equivalent mutants was also diagnosed and tackled by other researchers [71, 68]. Similar to invariant impact, coverage and data impact are also dynamic analysis techniques. Therefore, the static approaches can be used in combination with the dynamic approaches. If it can be statically proven that a mutation is equivalent, we do not need to compute its impact and can focus on those mutations that cannot be handled with static approaches.

The traditional use of statement coverage is to measure how well tests exercise the code under test and to detect areas of the code that are not covered by the tests. In contrast to our approach which also takes into account the execution frequency of a statement, traditional statement coverage just measures whether a statement is executed or not. Besides its traditional use, statement coverage is also used in different scenarios.

Jones and Harrold [45] presented the TARANTULA tool that uses coverage information for *bug localization*. By comparing the statement coverage of passing and failing test cases, the suspiciousness of a statement is computed. The intuition behind this approach is that statements that are *primarily executed by failing test cases* are more suspicious than statements primarily executed by passing tests. The statements can then be ranked according to their suspiciousness. The results of their evaluation show that the defective statement is ranked in the top 10% for 56% of the cases. Both our approach and TARANTULA follow the idea that changed coverage expresses anomalous behavior. TARANTULA uses coverage differences between test cases to find a defective statement while our approach uses coverage difference between runs of the test suite to estimate the impact of a mutation.

Elbaum et al. [21] investigated how statement coverage data changes when the source code is changed. The results suggest that even small changes can have a huge impact on the code coverage. However, the authors also state that the changes in code coverage are hard to predict. These findings support our decision to use statement coverage as an impact metric because mutations are also small changes. Furthermore, not all changes result in coverage changes which indicates that the coverage changes are sensitive to semantic changes.

Gordia [31] proposed the concept of *dynamic impact analysis*. To this end, the *dynamic impact graph* is built which is directed and acyclic. The nodes of the graph represent different executions of the program elements. Edges are between nodes that can potentially impact each other. The edges carry a probability that tells how likely an element impacts its direct successor. By traversing the graph, it can be computed how likely it is that a program element impacts an output element. The proposed applications of this approach are to estimate the risk of changes and to aid in test case selection for mutation testing. However, this approach might also be used in a similar way as our approach. For a not detected mutant its probability of manipulating the output can be computed, which corresponds to its chance of being non-equivalent.

Test case prioritization [85] is concerned with meeting a specific performance goal faster by reordering the execution of test cases, e.g. to detect faults earlier in the testing process, or to reach specific coverage goals faster. To this end, coverage data is used to prioritize the test cases according to different strategies.

Software change impact analysis aims to predict the results of code changes, that is, which parts of the code are affected by a change. Orso et al. [75, 76] proposed an approach for software impact analysis which is also called COVERAGEIMPACT. They use their *Gamma* approach to collect actual field data from deployed programs to compute the potential impact of a program change. To this end, traces are taken from a sample of deployed programs. A trace, in this context, consists of all methods that were called during an execution of the program. Every trace that traverses a changed method is identified, and all methods that are covered by this trace are added to a set of *covered methods*. Then, a static forward slice for every method that was changed is computed, and the methods that are covered by the slice are collected in a set of *slice methods*. The intersection of both sets is the set of methods that are potentially impacted by this change. Besides using the same name, our approach and the approach by Orso et al. are different. Orso et al.'s approach samples many executions on different machines to approximate the impact of potentially larger changes on deployed programs and typical usage scenarios, while our approach just aims to assess the impact of one small change (mutation) for a specified usage scenario (test cases) for one deployed version. However, using static forward slices can also help assessing the equivalence of a mutant. A similar idea was proposed by Hierons et al. [39].

7.5 Summary

In this chapter, we introduced the concepts of *coverage impact* and *data impact* of a mutation. The coverage impact is *the number of methods* that have *lines with different execution frequencies* between a run of the test suite on the mutated and the original version, and the data impact is the number of methods that have *different return values* between the two runs. To further emphasize non-local impact, we introduced *distance metrics* that measure the distance between the method containing the mutation and its furthest impacted method. The distance between two methods is the length of the shortest path between them in the *undirected call graph*.

In a study on seven open-source programs, it was shown that both *coverage and data impact* can be used to *assess the equivalence of mutations with a good precision and recall*, i.e. most mutations with impact are non-equivalent and most of non-equivalent mutations are detected by impact metrics. The distance metrics reach a higher precision at the cost of a lower recall, i.e. a higher percentage of mutations are correctly classified as non-equivalent but fewer are detected. When we compare the different approaches, we see that the coverage impact performs slightly better than the

data impact, and all metrics based on coverage and data impact perform better than the invariant metrics in terms of recall, i.e. they detect more non-equivalent mutations. Combining coverage and data metrics increases the recall, and by focusing on high impact mutants, the results can be improved further in terms of precision, e.g. if we rank the undetected mutations by impact and choose the top 25% ranked mutations, 93% of them are non-equivalent.

Chapter 8

Calibrated Mutation Testing

During mutation testing, defects are generated by using predefined mutation operators which are inspired by faults that programmers tend to make. But how representative are mutations for real faults? Andrews et al. [3] showed in their study that mutations are better representatives for real faults than faults generated by hand. However, different types of faults are made in different projects [52]. Therefore, mutation testing might be improved by *learning from the defect history*. As future defects are often similar to past ones, mutations that are similar to past defects can help to develop tests that also detect future bugs. This motivated us to develop *calibrated mutation testing* which is a technique that produces mutations according to characteristics of defects that were detected in the past. To this end, we mine the repository of a project for past fixes. These fixes represent attempts to cure a defect. Thus, they provide information about past defects. By producing mutation sets that share properties with these past fixes, we *calibrate* mutation testing to the *defect history* of a project. Figure 8.1 summarizes our approach. First, fixes are mined from different sources. Then, the mutations are (Step 2) calibrated to characteristics of the fixes and (Step 3) presented to the tester.

8.1 Classifying Past Fixes

Calibrated mutation testing aims to produce mutations similar to past defects, as made during the development of a project. To this end, we need to obtain information about

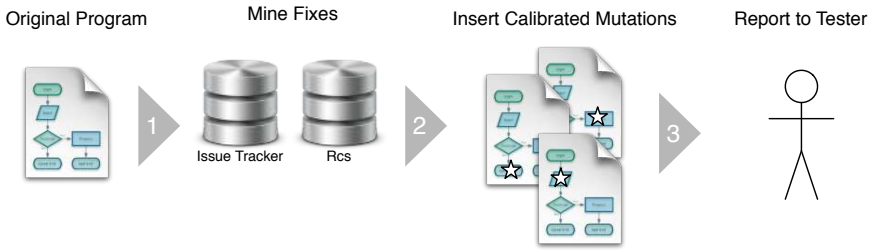

Figure 8.1: The process of calibrated mutation testing.

these defects. We do this by learning from fixes as they represent attempts to cure a defect, and describe a defect. In the following sections, we describe how to collect and classify past fixes for a software project.

8.1.1 Mining Fix Histories

Defects are constantly fixed during the development of a project. These fixes, however, are documented in different ways. Thus, we mine fixes for a project from different sources:

Revision Control System (RCS) Revision Control Systems such as cvs, svn, and git, provide commit message. These messages may contain the word 'fix' which indicates that a defect was fixed with this revision. We use the Kenyon framework [7] for collecting such commits automatically. Then, we manually validate if they are really fix revisions.

Issue tracker connected with RCS logs Several commit logs may contain the issue index numbers of an issue tracking system. We can automatically collect the revisions that contain these numbers with Kenyon, and then check manually whether they are really fix revisions corresponding to a certain issue.

Testing We can compare the test results of two adjacent revisions to obtain fixes. If a test case fails on the former revision and passes on the latter one, this is a fix revision.

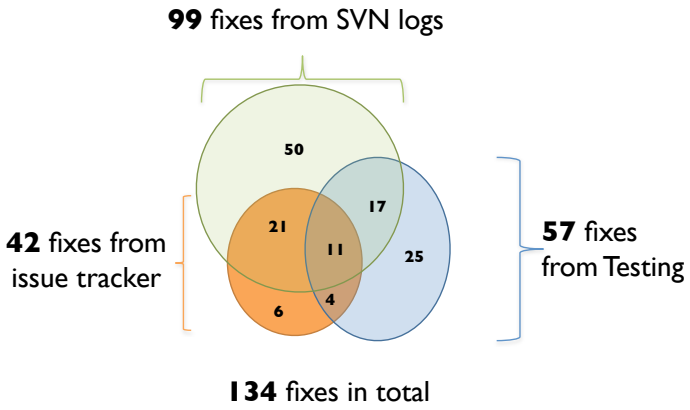

Figure 8.2: Collected fixes for the JAXEN project.

8.1.2 Subject Project

A project has to fulfill several criteria to be suitable for calibrated mutation testing. It has to have a version history that lasts long enough, so that we can learn from past fixes. There has to be an issue tracker from which we can extract bugs that were fixed. Furthermore, it has to come with a JUNIT test suite, so that we can test several revisions automatically and apply mutation testing to it.

The JAXEN XPath engine fulfills these criteria. It has a version history of 1319 revisions that span a duration of about 7 years. It is of medium size (12,438 lines of code) and comes with a JUNIT test suite that consists of 690 test cases.

Figure 8.2 shows the fixes that we collected for JAXEN, with the techniques described in the previous section. In total, we got 134 fixes. Out of these, 99 fixes were extracted from the version archive (svn), 42 from the issue tracker, and 57 from comparing the test results of subsequent revisions. Some of the fixes extracted from different sources are overlapping, which is shown by a Venn diagram. For example, 21 of the fixes that were extracted from the version archive were also extracted from the issue tracker.

Table 8.1: Fix pattern properties and their values.

Properties	Values
Flow	control(C), data(D)
Fix change	add(A), remove(R), modify(M)
Syntactic	if(IF), method call(MC), operation(OP), switch(SW), exception handling(EX), loop(LP), variable declaration(DC), casting(CT)

8.1.3 Fix Categorization

The collected fixes were categorized according to the fix pattern taxonomy proposed by Pan et al. [78]. The taxonomy categorizes fixes based on syntactical factors and their source code context. For example, the 'IF-APC' pattern describes fixes that add a precondition check via an if statement. In total, Pan et al. propose 27 fix patterns. However, some fixes could not be categorized by the taxonomy, so we added 12 new fix patterns.

For each fix pattern, we also determined several properties that describe its effects. The *flow* property describes the impact of the fix on control and data flow. The *change* property indicates if new code was added, or existing code was changed or deleted. The *syntactic* property describes which type of statements were involved in the fix (e.g. if statements or method calls). Table 8.1 shows all values for the different properties. Note that a fix pattern can also be associated with multiple values from each category.

Furthermore, we also determined the location for each fix—that is, the package, class, and method that were affected by the fix. Thereby, we could identify areas of the program that were more vulnerable than others.

By using these patterns and properties, each fix can be categorized. Although the fixes were categorized manually, each of the categorization steps could be automated.

8.2 Calibrated Mutation Testing

The idea of calibrated mutation testing is to adapt mutations to the defects that occurred during the development of the project. We obtain information about the defects by

Table 8.2: 10 most frequent fix patterns for Jaxen.

Pattern name	Abbreviation
Change of Assignment Expression	AS-CE
Addition of an Else Branch	IF-ABR
Change of if condition expression	IF-CC
Addition of Precondition Check	IF-APC
Method call with different actual parameter values	MC-DAP
Addition of a Method Declaration	MD-ADD
Addition of Precondition Check with jump	IF-APCJ
Removal of an Else Branch	IF-RBR
Modify exception message	EX-MOD
Addition of Operations in an Operation Sequence	SQ-AMO

Table 8.3: Classification of the 10 most frequent fix patterns.

	Properties				
Pattern	**Flow**	**Change**	**Syntactic**	**Frequency**	**Mutation operator**
AS-CE	D	M	OP	18	replace assignments
IF-ABR	CD	A	IF	13	skip else
IF-CC	C	M	IF	13	negate jump in if
IF-APC	C	A	IF	9	remove check
MC-DAP	D	M	MC	9	replace method arguments
MD-ADD	D	A	DC	9	-
IF-APCJ	C	A	IF	8	skip if
IF-RBR	CD	R	IF	7	always else
EX-MOD	N	M	EX	6	-
SQ-AMO	D	A	MC	6	remove method calls

mining past fixes. Using this data, we define mutation operators that mimic defects that were fixed (Section 8.2.1), and devise mutation selection schemes based on the properties of previous fixes (Section 8.2.2).

8.2.1 Mutation Operators

To generate mutations similar to actual defects, we use the fix patterns that are based on actual fixes. For a fix pattern, we derive a mutation operator that reverses the changes described by the pattern. Thereby, we introduce defects similar to fixes that are represented by this pattern.

For the JAXEN project, we examined the 10 most frequent fix patterns and checked which mutation operators reverse this pattern. Table 8.2 gives the names of the 10 most frequent fix patterns together with their abbreviations, and the results are summarized in Table 8.3. The first column gives the fix pattern ordered by their frequency (column 5). Columns 2 to 4 show the properties associated with each fix pattern. The mutation operator that corresponds to a fix pattern is given in the last column. For 8 out of these 10 patterns, we found a corresponding mutation operator. For our experiments, we used JAVALANCHE, the mutation testing framework presented in Chapter 4. One of the mutation operators that corresponds to a pattern was already implemented in JAVALANCHE, and for the seven others, we had to implement new mutation operators.

For example, the corresponding mutation operator for the 'IF-APC' pattern (see Section 8.1.3) removes checks. Checks are if conditions without an else part. Thereby, code that was previously guarded by the check gets executed regardless of the check result.

8.2.2 Mutation Selection Schemes

As we do not want to apply all possible mutations exhaustively, we propose different *selection schemes* that aim to represent past defects. To this end, a selection scheme takes properties of past fixes into account and selects mutations according to these properties. Therefore, we also mapped the characteristics, described in Section 8.1.3, to mutations and derived different selection schemes:

Pattern-based scheme Past fixes can be described by fix patterns. Many of these fix patterns can be related to mutation operators as described in Section 8.2.1. By using this relation, we can select a set of mutations that reflects the distribution of fix patterns among the past fixes.

Property-based schemes A fix pattern is characterized by different properties (flow, change, and syntactic properties). We calculate the distribution of these properties among all fixes. Then, we select a set of mutations that manipulate these properties so that the distribution of properties among the fixes is reflected—e.g. if more fixes are associated with a property, more mutations are selected to manipulate this property. In this way, we get three different mutation selection schemes as there are three different properties.

Location-based schemes Source code locations where defects were fixed in the past may be more vulnerable than other locations. Thus, we collect the locations of the fixes and count their occurrences. Then, we select mutations according to the distribution of fix locations. By considering three different granularity levels (package, class, and method level), we obtain three different location-based schemes.

Random scheme A scheme that randomly selects mutations from all possible mutations serves as a benchmark.

From the different selection schemes, we obtain different sets of mutations, which are calibrated to different aspects of the defect history of the project. Note that the mutation selection schemes are not deterministic. When a selection scheme is applied multiple times, different mutations may be chosen, and only the distribution of the underlying characteristic stays the same.

8.3 Evaluation

For the evaluation of our approach, we were interested if mutation schemes that are based on properties of previous fixes help to develop better tests than schemes that are based on a random selection of mutations. But how do we define better or good tests in this context? Although there are many different opinions on what makes a good test, for our evaluation, we consider a test to be good when it *detects bugs*.

Therefore, in an ideal setting, we would first apply each mutation selection scheme to a project and develop tests that cover the selected mutations. Then, we would check how many bugs these tests detect. Unfortunately, the first step is very hard as it would involve tremendous human effort to write tests for all mutations, and the second step is impossible as we do not know all bugs that are in a project. Thus, we propose an alternative evaluation setting that is based on the version history of a project.

8.3.1 Evaluation Setting

For a project, we check out every revision from the revision control system. Then, we compile each revision using the build scripts of it. If the revision can successfully be compiled, we also run its unit tests and record the results of the individual tests. With this approach, we obtain a matrix that depicts which test passes or fails on which revision. Using this data, we consider a test to detect a bug if the test fails on a revision and passes on a later one.

For evaluating our approach, we pick a specific revision. Then, we compute all fixes, as described in Section 8.1, up to this revision. By learning from these fixes, we create mutation sets according to different selection schemes. For these mutation sets, we check by which tests each mutation is detected. Then, we prioritize the tests in a way that they are sorted by the number of additional mutations that they detect. In this way, we obtain a test prioritization for each scheme. To assess the quality of a prioritization, we check how many future bugs are detected by the tests in the top x percent of the prioritization. Again, the number of future bugs that a test detects is derived from the previously produced matrix. As we create multiple sets for each mutation generation scheme, the result is the average number of bugs detected by tests from the different prioritizations. For each scheme, we compare this number to the results obtained for randomly selected mutations, and test whether differences are statistically significant. We use the Mann–Whitney U test to determine whether a difference is statistically significant because we could not ensure that the underlying data is normally distributed, i.e. the hypothesis that the data is normally distributed was rejected by the Shapiro–Wilk test.

8.3.2 Evaluation Results

We applied our approach to two revisions of JAXEN (revisions 1229 and 931). We randomly chose them among the revisions that had enough (≥ 5) tests that detect defects in future revisions. For each revision, we applied the different selection schemes and produced 100 different mutation sets for each scheme. Then, we checked the average failure detection ratios of the prioritizations that were produced for these sets.

Table 8.4 and Table 8.5 show the results for applying the different selection schemes on revision 1229 of JAXEN. The columns give the selection scheme that the prioritization is based on, and the rows give the percentage of tests considered. A value in the table depicts how many future faults are found on average by tests in the top x percent

Table 8.4: Defects detected for pattern and location based schemes (for revision 1229).

Top x percent	Fix pattern	Location Package	Location Class	Location Method	Random
10	0.39	0.75	0.53	0.68	0.58
20	0.85	**1.45**	**0.67**	0.87	1.00
30	**1.39**	**2.61**	**1.30**	**2.54**	1.95
40	**2.28**	3.68	**2.07**	**3.93**	3.24
50	**3.02**	4.47	**2.80**	**4.78**	4.14
60	**3.82**	5.13	**3.84**	**5.59**	4.59
70	**4.64**	5.63	**5.84**	**6.58**	5.32
80	**5.58**	6.29	6.27	6.62	6.40
90	8.95	8.63	8.66	9.31	8.85
100	10.00	10.00	10.00	10.00	10.00

of a prioritization. The values that are better than the values for the random scheme are underlined, and statistically significant values are shown in bold face.

For the detected defects, especially for the random prioritization, one would expect a linear distribution, i.e. for the top x percent, $x/10$ bugs are detected. However, all schemes perform worse. This is due to the setup of the evaluation. First, the mutations are chosen, and then, the tests are prioritized according to these mutations. For the random scheme, also the mutations that are used for the prioritization are randomly chosen and *not the tests*. A test that detects a bug but only detects few or no mutations is ranked low in all prioritization schemes. As there are such tests in this revision, the prioritization schemes perform worse than a linear distribution, which would be achieved by randomly choosing tests.

The results for the prioritization based on the fix pattern scheme (column 2 of Table 8.4) give constantly worse values than the ones for the random scheme (last column of Table 8.4) for the top 10 to 80%, and 6 out of them are significantly worse. Two of the location based schemes, the package and method based scheme (column 3 and 5 of Table 8.4), produce mostly better results than the random scheme. For the scheme that is based on the package location (column 3 of Table 8.4), though, this is only statistically significant for the top 20 and 30% of the test cases. The scheme that is based on the method location of previous fixes (column 5 of Table 8.4) has statistically significant better values for the top 30 to 70%. For the scheme that is based on class location (column 4), this trend does not seem to hold as it produces values that are worse than

Table 8.5: Defects detected for property based schemes (for revision 1229).

Top x percent	Property Flow	Change	Syntactic
10	0.49	0.72	0.56
20	0.94	1.31	1.13
30	1.78	2.19	1.67
40	2.79	3.11	**2.62**
50	**3.47**	3.9	**3.47**
60	4.21	4.44	4.19
70	4.85	4.94	4.86
80	**5.65**	6.06	**5.76**
90	8.46	8.63	8.81
100	10.00	10.00	10.00

the ones for the random scheme. This effect seems counterintuitive and might be due to the nature of the tests. Tests that perform well on mutations in a package seem to be more successful in defect detection than tests that perform well on mutations in defect prone classes. More specific tests that perform well on mutations in defect prone methods are again better in defect detection. The test prioritizations generated for the property based techniques (column 2 to 4 of Table 8.5), on the other hand, perform worse than the prioritization based on a random selection of mutations, except for the change property based scheme which has better values for the top 10 to 30%. However, these values are not statistically significant.

In order to check whether these results hold throughout the history of the project, we also looked at an earlier revision. Table 8.6 and Table 8.7 show the corresponding results for revision 931. Note that for almost all prioritization schemes, all the defects are detected at 90% . This is due to tests that detect very only few mutations but no defect. Therefore, they are always ranked very low by the different prioritizations. The prioritizations based on the fix pattern type performs better for the top 10 and 20% but worse for the rest. The location based schemes, which perform best for revision 1229, do not perform well on this revision. Only three values are better than the ones for the random scheme, but none is statistically significant. Among the property based prioritization schemes, the scheme based on the flow property of fixes performs best. The top 10 to 30% perform better than the random scheme, but again, this is not statistically significant. From these results, we can conclude that:

Table 8.6: Defects detected for pattern and location based schemes (for revision 931).

Top x percent	Fix pattern	Location			
		Package	Class	Method	Random
10	1.72	**1.12**	1.47	1.33	1.44
20	2.73	2.35	2.64	2.61	2.71
30	3.28	2.97	3.09	3.2	3.41
40	3.47	**3.35**	3.25	3.69	3.82
50	3.77	**3.65**	3.53	3.99	4.05
60	4.17	4.13	4.08	4.45	4.32
70	4.44	4.52	4.66	4.68	4.59
80	4.80	4.83	4.84	4.84	4.86
90	5	5	5	5	5
100	5	5	5	5	5

Random mutation selection schemes cannot be outperformed by selection schemes based on defect history.

8.4 Threats to Validity

Like any empirical study, this study is limited by threats to validity.

External validity As the evaluation was only carried out on one project, we cannot claim that the results carry over to other projects. However, the results may serve as a starting point for further investigation and discussion.

Internal validity Due to the randomness that goes into the selection schemes, they are affected by chance. To counter this threat, we produced 100 different mutation sets for each scheme, averaged the results and compared the results to the average of 100 randomly chosen sets. Furthermore, we followed rigorous statistical procedures to evaluate the results.

Construct validity We learned about past defects from fixes. Although we used 3 different methods to mine fixes, some of the fixes are missed—and thereby, data about past defects. For our evaluation, we had to make several approximations

Table 8.7: Defects detected for property based schemes (for revision 931).

Top x percent	Property Flow	Change	Syntactic
10	1.76	1.35	1.41
20	2.95	2.44	2.65
30	3.55	3.13	3.02
40	3.68	3.44	3.39
50	3.92	3.66	3.76
60	4.15	4.17	3.93
70	4.49	4.6	4.43
80	4.79	4.90	4.73
90	5	5	4.98
100	5	5	5

that may have influenced the results. The test cases to detect (kill) mutations were chosen from existing test cases rather than writing new ones. This created a bias towards the type of tests as some mutations are not detected, and tests that covered large parts of the program were preferred. In addition, we only evaluate against defects that were detected by tests that existed in an earlier revision. This provided only a few defects and created a bias towards the type of defect.

8.5 Related Work

8.5.1 Mining Software Repositories

Software repositories have been mined for various purposes, among others to suggest and predict further changes [108], defect prediction [46] and to infer API Usage patterns [53].

The DynaMine tool presented by Livshits and Zimmermann [53] combines software repository mining and dynamic analysis techniques. In their approach, they mine the commits for usage patterns, and then they use dynamic analysis to validate the patterns and to find violations. Similar to our approach, they also use a combination of repository mining and dynamic analysis techniques, but for a different purpose. They

try to detect defects in the form of pattern violations while we aim to improve test suites through calibrated mutation testing.

Pan et al. [78] studied the bug fix patterns in seven open-source projects and came up with a categorization that covers up to 63.3% of all fixes made in a project. Their fix classification forms the basis for one of our calibration schemes.

To our knowledge, calibrated mutation testing is the first trial to combine mutation testing and software repository mining.

8.5.2 Mutation Testing

In one of the first publications on mutation testing, DeMillo et al. [18] introduced the *coupling effect* that relates mutations to errors and is stated as follows:

> *Test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors.*

Offutt [66, 67] later provided evidence for the coupling effect with a study on higher order mutants. The results of our work can also be seen as a support of the coupling effect. They indicate that tests detecting randomly chosen mutations are as effective as tests detecting calibrated mutations.

Mutation selection was first proposed by Mathur [57] as a way to reduce the costs for mutation testing by omitting the mutation operators that produce most mutations. The goal of mutation selection is to produce a smaller set of mutations so that tests that are sufficient (they detect all mutations) for the smaller set are also (almost) sufficient for all mutations.

In an empirical study on 10 Fortran programs, Offutt et al. [69] showed that 5 out of 22 mutation operators are sufficient—test suites that detect all selected mutations also detect 99.5% of all mutations.

Barbosa et al. [6] did a similar study on C programs. They proposed 6 guidelines for the selection of mutation operators. In an experiment on 27 programs, they determined 10 out of 39 operators to produce a sufficient set with a precision of 99.6%.

A slightly different approach was taken by Namin et al. [61]. They tried to produce a smaller set of mutations that can be used to approximate the mutation score for all

mutations. Using statistical methods, they came up with a linear model that generates less than 8% of all possible mutations, but accurately predicts the effectiveness of the test suite for the full set of mutations.

In a recent study, Zhang et al. [107] compared these 3 different operator-based selection techniques to random selection techniques. They showed that all techniques performed comparably well, and that random selection techniques can outperform the best operator-based selection techniques. This is similar to our results, which show no significant difference between more sophisticated calibrated selection techniques and simple random techniques in terms of fault detection ratios for the corresponding test suites.

The selective approaches aim at minimizing the number of mutations and approximating the results for all mutations while our approach selects mutations calibrated to past defects in order to approximate future defects.

8.6 Summary

One application of mutation testing is to improve a test suite by analyzing not detected mutations and developing new tests that detect them. In this work, we investigated whether calibrating mutations to the defect history improves this process as future defects are often similar to past defects. To this end, the inserted mutations were calibrated to different aspects of the defect history expressed through fixes made in the past. We proposed different calibration schemes that are based on the pattern type, the location, and on different properties of the fix. Then, we prioritized tests in such a way that the ones detecting most mutations for a selection scheme were ranked first. For the obtained prioritization we checked how many future defects they would detect.

Although the results on one revision showed better values for some calibrated schemes than for random selection, we found no scheme in which this trend manifested for more revisions. Our results can, therefore, be seen in line with previous research by Zhang et al. [107]. They compared random selection schemes with different operator selection schemes, and in their evaluation random selection schemes were also not outperformed by more sophisticated schemes. Furthermore, our results might also be seen as support for the coupling effect—which states that tests that are sensitive enough to detect simple errors also detect complex errors. From our results, we can conclude that tests that detect randomly chosen mutations are as effective as tests that detect calibrated mutations.

Chapter 9

Comparison of Test Quality Metrics

During software testing, a developer is interested in writing good tests. In most cases a test is considered to be good when it has the potential to reveal defects in the code. Test coverage metrics can be used to assist in writing new tests. A developer can create new tests in such a way that the coverage level is increased, i.e. previously unsatisfied test requirements get satisfied by new tests. This process might help to produce better test suites than those obtained by unsystematically adding new tests. In this chapter, we investigate whether coverage metrics help to develop better test suites, i.e. *if there is a correlation between the coverage level of a test suite and its defect detection capability*. If such a correlation exists, this would mean that test suites reaching a higher coverage are more likely to detect defects, and fewer defects are undetected by the test suite. Thus, developers can gain confidence that their program contains fewer defects if their program exhibits no defect for a test suite with a high coverage.

Another issue when testing software is the cost-effectiveness of a test generation strategy. Developing more tests also means higher cost in terms of development effort and computing resources. For example, it might be the case that test suites satisfying criterion C_1 are more likely to detect defects than tests that satisfy criterion C_2. However, producing a test suite that satisfies criterion C_1 requires more effort than producing a test suite that satisfies C_2, e.g. more tests are needed to satisfy C_1 than for C_2. In cases like these, the better defect detection capabilities can also be caused by

the larger size of test suites satisfying C_1 compared to suites satisfying C_2. Therefore, one is interested in a metric where the coverage level correlates with defect detection *independently of the test suite size*.

Even though such a correlation might exist, there is no guarantee that test suites reaching a high coverage level will detect defects. One reason for this is that there are many tests that satisfy a test requirement or detect a defect. From the universe of all possible tests a specific fraction of tests does both: it satisfies a requirement and also detects a specific defect. The defect detection capability of a concrete test suite depends on the type of test chosen to satisfy a criterion, i.e. whether the test satisfies the requirement and detects the defect or only satisfies the requirement. Therefore, a correlation between defect detection and coverage can only give indication for the likelihood of a test suite detecting potential defects. In the same way subsumption relations do not carry over to the defect detection capability of a coverage metric. For example, it cannot be concluded that C_a correlates with defect detection when it subsumes a criterion C_b that does correlate with defect detection. This is because tests that satisfy requirements in C_a might have a different fraction of tests that detect a defect than tests that satisfy requirements C_b, although some of them also satisfy requirement imposed by both criteria.

9.1 Test Quality Metrics

In this work, we were interested whether a high coverage level correlates with defect detection, i.e. whether there is a correlation for test suites between their coverage level with regard to a metric and their defect detection capability. Especially, we were interested how mutation testing compares to traditional coverage metrics. Therefore, we generated several test suites for a set of subject programs and computed the mutation score and the coverage level for different metrics as well as the number of defects the test suites detect. The mutation score is defined as the number of detected mutants divided by the number of all mutants (see Chapter 3). We used the following coverage metrics to compare against mutation testing:

Statement Coverage that requires every statement to be executed.

Branch Coverage that requires every branch to be executed.

Modified Condition/Decision Coverage that requires assignments to all predicates so that each clause independently affects the outcome of its predicate.

Table 9.1: Description of the Siemens suite.

Project name	Lines of code	Number of test cases	Defects
print_tokens	536	4130	7
print_tokens2	387	4115	10
schedule	425	2650	9
schedule2	766	2710	10
tot_info	494	1052	23
replace	554	3155	32
tcas	136	1608	41

Loop Coverage that requires every loop construct to be executed zero times, once, or several times.

Section 2.3.1 contains a more detailed description of these metrics. For each coverage metric, we computed the coverage level of a test suite, which is the number of requirements satisfied by the test suite divided by the total number of requirements imposed by the coverage criterion.

9.2 Experiment Setup

In order to carry out our experiment, we needed subject programs that contain known defects and several tests that detect these defects. The Siemens test suite [19] is a set of programs that fulfills these requirements. It consists of seven different subjects that have in total 132 faulty versions and 19,420 different tests. For our experiments, we used a version adapted for JAVA by Masri et al. [56]. A description of the subjects can be found in Table 9.1

For each of the seven subjects, we generated several test suites with varying sizes. To this end, we followed the following method. We first fixed the number of test cases a test suite should contain, and then randomly picked that many test cases from the pool of all test cases. This method was applied to generate 250 test suites for each test suite size from 1 to 500 test cases. Following this procedure, we obtained 125,000 test suites per subjects. For each test suite, we computed the coverage metrics and how

many defects it detects. We computed the mutation score with JAVALANCHE, which was presented in Section 4. To compute the coverage level for the different metrics, we used the CODECOVER tool developed at the University of Stuttgart.

By using the test suites' coverage levels and the number of detected defects they detected, we tested for a correlation between them at different sizes. To test for a correlation, we used the Spearman's rank correlation coefficient, because the data is not normally distributed.

9.3 Results

The results of our experiments are shown in Figures 9.1, 9.2, 9.3, and 9.4. The correlation between the coverage level of a test suite and its defect detection capability for test suite sizes from 1 to 15 test cases is shown in Figure 9.1. For each subject, there is an own plot. The x-axis represents the test suite size and the y-axis the correlation coefficient. Different lines depict the development of the correlation coefficient of the different metrics. For almost all projects there is a weak correlation, mostly around 0.4, between the coverage level and the defect detection capability of a test suite. As the test suite sizes increase the correlations decrease. These correlations are statistically significant for 5 out of 7 projects, i.e. the P-Value, obtained by using a t-distribution, is within the confidence interval of -0.05 and +0.05. For `schedule` and `schedule2` the correlation is only statistically significant for very small test suites. However, mutation testing performs better than the other metrics for these two projects. The correlation for mutation testing is statistically significant until the size of 10 for `schedule`, and for `schedule2` it becomes significant at the size of 10. The only project at which the coverage metrics do not decrease is `print_tokens2`. Here the correlation for all metrics except for loop coverage increases with growing test suite size.

For 5 out of 7 subjects mutation testing performs better than or is on par with the best coverage metric. For `schedule2` the correlation is stronger for the coverage metrics at small test suite sizes, but at a test suite size of 5 the correlation is again stronger for mutation testing. For `tot_info` the correlation is the strongest for modified condition/decision coverage.

The mutation score shows the best correlation to defect detection for most projects at small test suite sizes.

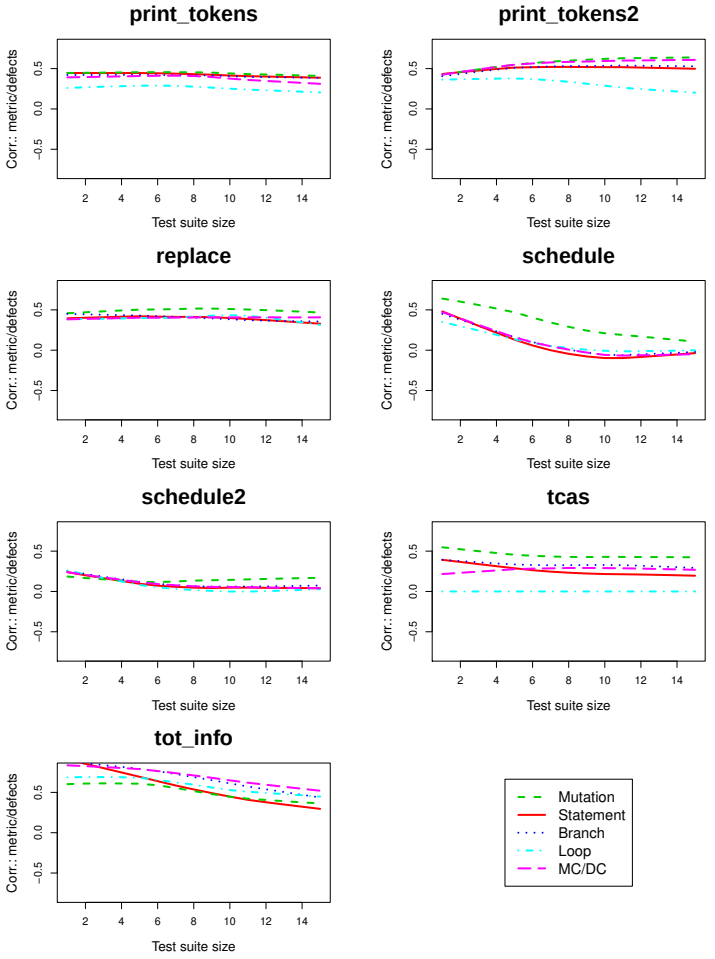

Figure 9.1: Correlation between coverage level and defect detection for test suite sizes 1 to 15.

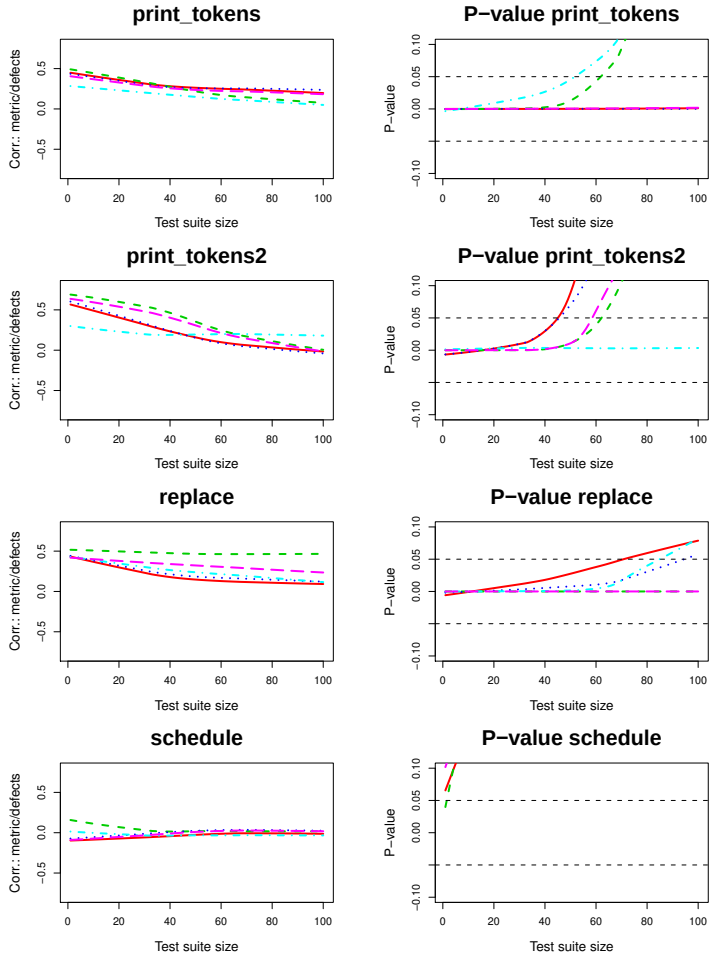

Figure 9.2: Correlation between coverage level and defect detection and corresponding P-values for test suite sizes 1 to 100.

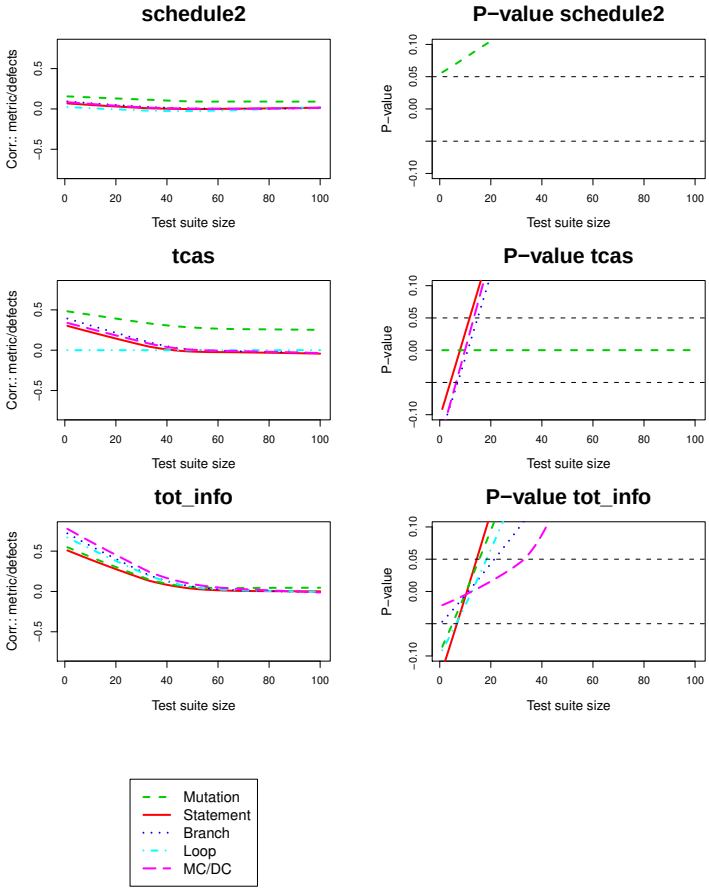

Figure 9.3: Correlation between coverage level and defect detection and corresponding P-values for test suite sizes 1 to 100.

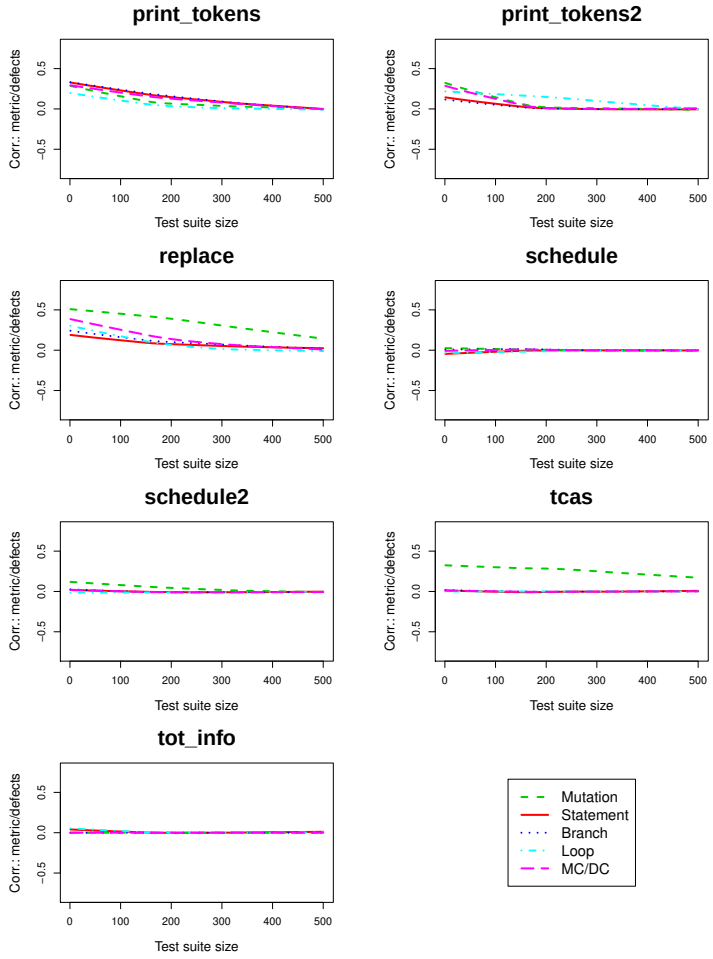

Figure 9.4: Correlation between coverage level and defect detection for test suite sizes 1 to 500.

Figure 9.2 and Figure 9.3 show the same type of graphs as Figure 9.1 but for an interval from 1 to 100. In addition, for each project they also show a graph for the P-value, which determines the significance of the correlation. These graphs show the development of the P-Value for the different metrics as the test suite size increases. The general trend we can observe is that the correlations decrease as the test suite size increases until there is no correlation (Spearman's rank correlation coefficient is zero). However, there are a few exceptions and some metrics perform better than others before the correlation becomes insignificant. Mutation testing performs best for three projects (`replace`, `schedule2`, `tcas`) during the whole interval, i.e. it has the strongest correlation. For two other (`print_tokens2`, `schedule`) projects it performs best at the beginning, and then at larger sizes another metric performs better. For example, mutation testing performs best until a test suite size of 60 for the `print_tokens2` subject, and then loop coverage performs better. For `schedule` mutation testing performs best until a test suite size of 35 and then all criteria have a correlation of 0. When we look at the development of the P-values, we see that for most projects there is a significant correlation for all metrics at smaller test suite sizes. Most correlations, with a few exceptions, become insignificant as the test suites get bigger. For example, almost all correlations for the `print_tokens2` subject become insignificant for test suite sizes between 40 and 60, and only for loop coverage a significant correlation remains. We suspect a direct relation between a test requirement and a defect, in cases where the coverage level during the whole interval stays significant, i.e. all tests that fulfill the test requirement do also detect a defect.

The development of the correlations for test suite sizes from 1 to 500 test cases is shown in Figure 9.4. The trend that we observed for the interval from 1 to 100 also holds for the interval 1 to 500. The correlation for all metrics decreases as the test suite size increases. However, there are two exceptions for the projects `replace` and `tcas`: a correlation for mutation testing remains until a test suite size of 500, again this effect might be caused by a direct relation between a test requirement and a defect.

For certain projects the correlation between mutation testing and detected defects holds longer than for other coverage metrics.

9.4 Threats to Validity

For our experiments, we used the seven subjects from the Siemens suite. They contain hand-seeded defects, which might be different from defects made in other programs.

As the subjects are relatively small, other effects might be observed for small test suites on bigger programs. Thus, it is possible that the results do not generalize to other programs with a bigger size and different types of defects.

Moreover, our construction method for the test suites might not reflect how developers proceed. We choose tests from a pool of existing tests, while developers choose from the universe of all possible tests. This might introduce a bias regarding the type of tests, e.g. the tests might have a different fraction of defect revealing tests.

Furthermore, our implementation and the programs we use might contain defects. To control for this threat, we relied on public, well-established open-source tools wherever possible, and made our own JAVALANCHE framework publicly available as open-source package to facilitate replication and extension of our experiments.

9.5 Related Work

Frankl and Weiss [25] compared the effectiveness of branch and data flow testing in terms of defect detection. In three experiments they compared all-uses coverage, branch coverage (all-edges coverage), and random selection of test cases. In a first experiment, they compared the defect detection rates of test suites that satisfy all-uses coverage, or branch coverage and randomly selected test suites. To this end, they built different test suites for nine small subjects, reaching from 22 to 78 lines of code, each containing several defects. For these test suites it was checked whether they detect an error in the program and whether they satisfy one of the coverage criteria. Then, the ratio of test suites that detect at least one defect was compared for randomly selected test suites and test suites that satisfy a criterion. The results showed that all-uses adequate test sets were better than or equal to branch adequate test suites in all cases. This difference was statistically significant in 5 out of 9 cases. Both criteria were better than randomly selected test suites in all cases. For the all-uses adequate sets this was statistically significant in 6 cases and for branch adequate test suites in 5 cases. This experiment, however, did not consider the size of the test suites. Test suites satisfying the all-uses criterion are typically larger than test suites satisfying branch coverage, and for both criteria the test suites were larger than an average randomly selected test suite. Therefore, in a second experiment the size of the test suites was fixed, so that test suites of the same size were compared. The results showed that significantly more all-uses coverage adequate test suites detect defects than randomly selected test suites in 4 out of 9 cases. Branch adequate test suites, however, did not perform significantly

better than randomly selected test suites. In a last experiment, which is most similar to our approach, the dependency between the coverage level and the defect detection capability of a test suite was investigated. To this end, a regression model in terms of size and coverage level was built for each subject. The coverage level was considered to correlate with the defect detection capability if the coefficient of the term containing highest power of the coverage level was positive and has a large magnitude. The investigation of the regression results showed that there is such a dependency for 4 out of 9 cases, for both all-uses coverage and branch coverage.

Frankl et al. [27] also compared the all-uses coverage criterion and mutation testing. For this comparison the same setup as in the previously described study was used. In their experiments the size of the test suite for each subject was fixed and the defect detection rates for different coverage levels were compared. The results indicated that mutation coverage is more effective than all-uses coverage for 7 out of 9 subjects, and all-uses coverage is more effective for 2 subjects.

In a later paper, Frankl and Iakounenko [26] compared the effectiveness of branch and data flow testing again, but on a different subject program. In this experiment they used the space program, a medium sized C-program with 5,905 lines of code that comes in 33 different versions, each containing one defect. Frankl and Iakounenko restricted their experiments to eight versions, in which defects were harder to detect, i.e. only a limited fraction of 5000 randomly selected test cases detect the defect. The results showed that test sets reaching high all-uses coverage or high branch coverage are more likely to detect a defect than a randomly chosen test suite of the same size, for all of the eight investigated versions. However, the defect detection rates of test suites reaching a high coverage were still relatively low for most of the subjects.

In comparison to our approach, Frankl et al. [25, 27, 26] did not consider the number of defects in their experiments, they just checked whether a test suite detects any defects or none. Furthermore, they did not control for the size of the test suites in many of their experiments.

Hutchins et al. [43] carried out an experiment similar to the first experiment of Frankl and Weis. In their work, they also compared data flow and control flow based coverage criteria in terms of their defect detection capability. As subject programs they used 7 C-programs with hand seeded defects, which also form the basis for the programs used in our study. For each program they generated a test pool and built several test suites of varying sizes. Then, they compared the defect detection capabilities for the test suites that lie in specific coverage intervals, for branch coverage and definition use coverage. The results showed that both coverage criteria performed better than

randomly selected test suites for high coverage levels, i.e. when the coverage level was above 93%. Between the two criteria there was no significant difference. In contrast to our approach, Hutchins et al. used a different approach to generate test suites. Their approach is focused on the coverage level while we randomly generate test suites of a specific size. For their experiments, they built test suites by incrementally adding new test from the pool, but they only added tests that increase the coverage and pruned tests that do not increase it. This process is repeated until a specific size of tests or maximum coverage is reached.

Andrews et al. [4] investigated whether mutation testing can be used to predict the effectiveness of a test suite in detecting real-world defects. Among other questions, they thus investigate whether mutation scores are a good predictor of the defect detection ratio. In an experiment, they used the space program, which was also used by Frankl and Iakounenko. For a first experiment, they compared the average ratio of detected defects with the average ratio of detected mutants of test suites generated to fulfill several coverage criteria. The results indicate the difference between these ratios is small. Therefore, it is concluded that the mutation score can be used as a predictor for the defect detection capability of a program. This finding is then used as a basis to investigate further questions regarding the cost-effectiveness of test suites generated according to specific coverage criteria. Compared to our approach, Andrews et al. used the similarity of defect and mutation detection ratios of test suites as a basis for further experiments that involve other coverage metrics, while we compare different coverage metrics directly to mutation testing.

9.6 Summary

In this chapter, we investigated whether the defect detection capability of a test suite correlates with the coverage level with regard to a specific criterion. Our results showed that a weak correlation exists at small test suite sizes for almost all metrics. In most cases mutation testing performed best. At larger test suite sizes most of these correlation become statistically insignificant, or there is no measurable correlation anymore. This effect is due to the coverage level these test suites reach. Because our subject programs are small, a high coverage for most metrics can be reached with few tests. This coverage level then rarely increases when more tests are used. Thus, we expect a slightly different effect for larger programs. There might also be the trend that the correlations decrease with increasing test suite size but this effect might start later at bigger test suite sizes, because more tests are needed to reach a high coverage level.

For a few projects and few metrics, however, we observed correlations that lasted during the whole observed interval. In these cases, we suspect a direct relation between the defect and the test requirement, i.e. all tests that satisfy a requirement do also detect a defect.

Chapter 10

Checked Coverage

The previous chapters were focused on mutation testing, a technique to assess and improve the quality of the inputs and checks of a test suite. In this chapter, we propose a new alternative coverage metric that also assesses the quality of a test suite's checks.

Coverage criteria are the most widespread metrics to assess test quality. Test coverage criteria measure the percentage of code features such as statements or branches that are executed during a test. The rationale is that the higher the coverage, the higher the chances of catching a code feature that causes a failure—a rationale that is easy to explain, but it relies on an important assumption. This assumption is that we are actually able to *detect* the failure. It does not suffice to cover the error, we also need a means to detect it.

As it comes to detecting arbitrary errors, it does not make a difference whether an error is detected by the test (e.g. a mismatch between actual and expected result), by the program itself (e.g. a failing assertion), or by the runtime system (e.g. a dereferenced null pointer). To validate computation results, though, we need checks in the test code and in the program code—checks henceforth summarized as *oracles*. A high coverage does not tell anything about oracle quality. It is perfectly possible to achieve a 100% coverage and still not have any result checked by even a single oracle. The fact that the runtime system did not discover any errors indicates robustness but does not tell anything about functional properties.

As an example of such a mismatch between coverage and oracle quality, consider the test for the `PatternParser` class from the JAXEN XPATH library, shown in

Figure 10.1. This test invokes the parser for a number of paths and achieves a statement coverage of 83% in the parser. However, none of the parsed results are actually checked for any property. The parser may return complete nonsense, and this test would never notice it.

```
public void testValidPaths() throws
  JaxenException, SAXPathException {
  for (int i = 0; i < paths.length; i++) {
    String path = paths[i];
    Pattern p = PatternParser.parse(path);
  }
}
```

Figure 10.1: A test without outcome checks.

To assess oracle quality, a frequently proposed approach is *mutation testing*. Mutation testing seeds artificial errors (*mutations*) into the code and assesses whether the test suite finds them. A low score of detected mutants implies low coverage (i.e. the mutant was not executed) or low oracle quality (i.e. its effects were not checked). Unfortunately, mutation testing is costly; even with recent advancements, there still is manual work involved to weed out equivalent mutants.

In this work, we introduce an alternative, cost-efficient way to assess oracle quality. Using dynamic slicing, we determine the *checked coverage*—statements which were not only executed, but which actually contribute to the results checked by oracles. In Figure 10.1, the checked coverage is 0% because none of the results ever flows into a runtime check (again, in contrast to the 83% traditional coverage). However, adding a simple assertion that the result is non-null already increases the checked coverage to 65%; adding further assertions on the properties of the result further increases the checked coverage.

Using checked coverage as a metric rather than regular coverage brings significant advantages:

- Few or insufficient oracles immediately result in a low checked coverage, giving a more realistic assessment of test quality.
- Statements that are executed, but whose outcomes are never checked would be considered *uncovered.* As a consequence, one would improve the oracles to actually check these outcomes.

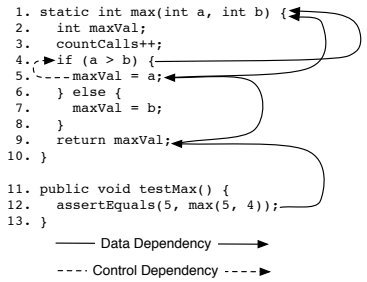

Figure 10.2: Dynamic data and control dependencies as used for checked coverage.

- Rather than focusing on executing as much code as possible, developers would focus on checking as many results as possible, catching more errors in the process.
- To compute checked coverage, one only needs to run the test suite once with a constant overhead—which is way more efficient than the potentially unlimited number of executions induced by mutation testing.

10.1 Checked Coverage

Our concept of checked coverage is remarkably simple: Rather than computing *coverage*,the extent to which code features are *executed* in a program, we focus on those code features that actually contribute to the results checked by oracles. For this purpose, we compute the *dynamic backward slice* of test oracles, i.e. all statements that contribute to the checked result. This slice then constitutes the *checked coverage.*

More formally, a program slice can be defined as the set of statements that influence the variables used in a given location, which is obtained by transitively following the control and data dependencies (for more details on program slicing see Section 2.4.6). The slices can be computed statically or dynamically. A static program slice consists of all statements that potentially influence the variables at a given point, while the dynamic program slice only consists of the variables that actually influence the variables during a concrete program run. We have chosen dynamic slices for our approach, because we use the slices to measure the quality of a concrete execution of the test suite.

Furthermore, we consider all occurrences of a statement during a program run for a dynamic slice. This is done by computing the union of the dynamic slices from every occurrence of a statement.

Figure 10.2 shows a test that exercises the `max` method, similar to the slicing examples (Figure 2.1 and Figure 2.2) in Section 2.4.6. To compute the checked coverage for this example, the dynamic backward slice from the call to `assertEquals` (statement 12) is built, and only the statements that are on this slice are considered to be covered.

10.1.1 From Slices to Checked Coverage

For the checked coverage, we are interested in the *ratio of statements* that contribute to the computation of values which are checked later by the test suite. To this end, we first identify all statements which check the computation inside the test suite. Then, we trace one run of the test suite, and compute the dynamic slice for all these statements. This gives us the set of statements that have a control or data dependency to at least one of the check statements. The checked coverage is then the percentage of the statements in the set relative to all coverable statements.

In order to define checked coverage in terms of a test requirement as introduced in Section 2.3, we define the set of statements that can be on a slice. Some statements of a program can never show up on a slice because there exists no data or control dependency from and to them. Therefore, for checked coverage only the statements that read or write variables or that can manipulate the control flow of the program are considered.

Definition 37 (Sliceable Statements) *For a program P, the set of sliceable statements $SL(P)$ consists of all statements that read or write a variable or manipulate the control flow.*

With the help of this set we can define the test requirements of checked coverage.

Definition 38 (Checked Coverage) *Checked coverage requires each sliceable statement $s \in SL(P)$ to be on at least one backward slice from an explicit check of the test suite.*

Checked coverage subsumes statement coverage because in order to reach full checked coverage every statement has to be on a dynamic slice, and thereby, it also has to be executed at least once. Thus, every test suite that reaches full checked coverage also reaches full statement coverage.

Note that in general checked coverage does not subsume all definitions coverage. There are two reasons for this. First, statements that manipulate the control flow can also define variables. Thus, they can end up on a slice because of control dependencies although their data dependencies are not exercised. Second, uses can also happen inside the test suite. In this case a definition is on a slice although no use of it in the program is exercised. If we account for theses two special cases, i.e. we do not allow definitions in control flow manipulating statements and we consider uses inside the test suite, checked coverage subsumes all definitions coverage. This is because a variable that is defined in a statement is always used when it appears on a slice. As the only way it can end up on a slice is via a data dependency which implies a use of the variable. Therefore, every test suite that satisfies checked coverage also satisfies all definitions coverage, under the premise that uses in the test suite are included and no definitions are made in control flow manipulating statements.

However, the main goal of checked coverage is to ensure that computations are also checked, and not just exercised. In order to find computations that are not checked, a developer can compare the level of statement coverage for a unit to the level of checked coverage. A high statement coverage level but a low checked coverage level for a unit indicates that it is exercised by the test suite, but the results are not checked. A developer can precisely tell which computations are not checked, by looking at the statements which are considered as covered by statement coverage but not by checked coverage.

Notice that checked coverage is focused on one type of checks: *explicit checks of the test suite*. Implicit checks of the runtime system or explicit checks in the program are not considered by checked coverage. The different checks, however, vary in the type of properties they check and how detailed they check. While explicit checks of the test suite verify the result for one concrete run and can make very detailed assumptions, the other checks can only examine generic properties that hold during all possible runs. Because the explicit checks of the program and the implicit checks are generic, they lend to static verification approaches very well, e.g. for many objects it can be proved that they never become `null` [24, 79]. In general explicit checks of the test suite cannot be efficiently proved because they make more complex assumptions. We believe that it is important for a test suite to have *strong explicit checks*, because the computed results should be *checked in detail*.

10.1.2 Implementation

For the implementation of our approach, we use Hammacher's JAVASLICER [33] as a dynamic slicer for JAVA. The slicer works in two phases: In the first phase, it traces a program and produces a *trace file*, and in the second one, it computes the slice from this trace file.

The tracer manipulates the JAVA bytecode of a program by inserting special tracing statements. At runtime, these inserted statements log all definitions and references of a variable, and the control flow jumps that are made. By using the JAVA agent mechanism, all classes that are loaded by the JAVA Virtual Machine (JVM) can be instrumented. There are a few exceptions, though, that are explained in Section 10.3. Since JAVA is an object-oriented language, the same variable might be bound to different objects. The tracer, however, needs to distinguish these objects. For that purpose, the slicer assigns each object a unique identifier. The logged information is directly written to a stream that compresses the trace data using the SEQUITUR [64, 63] algorithm and stores it to disk.

To compute the slices, an adapted algorithm from Wang and Roychoudhury [100, 101] is used. The data dependencies are calculated by iterating backwards through the trace. For the control dependencies, the control flow graph (CFG) for each method is built, and a mapping between a conditional statement and all statements it controls is stored. With this mapping, all the control dependent statements for a specific occurrence of a statement can be found.

Our implementation computes the checked coverage for JUNIT test suites, and works in three steps:

1. First, all checks and all coverable statements are identified. We use the heuristic that all calls to a JUNIT assert-method from the test suite are considered as checks. As coverable lines, we consider all lines that are handled by the tracer. Note that this excludes statements such as `try`, or simple `return` statements, as they do not translate to data or control flow manipulating statements in the byte code.

2. Second, all test classes are traced separately. This is a performance optimization, since we observed that it is more efficient to compute slices for several smaller files than computing it for one big trace file.

3. Finally, a union of all the slicing criteria that correspond to check statements is built, since the slicer supports to build a slice for a set of slicing criteria. By merging the slices from the test classes, we obtain a set of all statements that contribute to checked values. The *checked coverage score* is then computed by dividing the number of statements in this set by the—previously computed—number of coverable statements.

10.2 Evaluation

In the evaluation of our approach, we were interested whether checked coverage can help in *improving existing test suites* of mature programs. To this end, we computed the checked coverage for seven open-source programs that have undergone several years of development and come with a JUNIT test suite. We manually analyzed the results, and found examples where the test suites can be improved to more thoroughly check the computation results (see Section 10.2.2). We also detected some limitations of our approach, summarized in Section 10.3.

Furthermore, we were interested how sensitive our technique is to *oracle decay*—that is, oracle quality which was artificially reduced by removing checks. In a second automated experiment (Section 10.2.3), we removed a fraction of the assert-statements from the original test suites and computed the checked coverage for these manipulated test suites. This setting also allowed us to compare checked coverage against other techniques that measure test quality, such as statement coverage and mutation testing.

10.2.1 Evaluation Subjects

As evaluation subjects, we used seven open-source projects that were presented in Section 4.2 and in Table 4.2. Table 10.1 gives the results for computing the checked coverage (column 2), statement coverage (column 3) and the mutation score (column 4) for our subject projects. The statement coverage values are between 70% and 90% for all projects except for ASPECTJ and BARBECUE. For ASPECTJ the results are lower because we only used a part of the test suite in order to run our experiments in a feasible amount of time. Although we only computed the coverage of the module that corresponds to the test suite, test suites of other modules might also contribute to the coverage of the investigated module. For BARBECUE, we had to remove tests that address graphical output of barcodes, as we ran our experiments on a server that has no

Table 10.1: Checked coverage, statement coverage, and mutation score.

Project name	Checked coverage%	Statement coverage%	Mutation score%
ASPECTJ	13	38	63
BARBECUE	19	32	66
COMMONS-LANG	62	88	86
JAXEN	55	78	68
JODA-TIME	47	71	83
JTOPAS	65	83	73
XSTREAM	40	77	87
Average	**43**	**67**	**75**

graphics system installed and this causes these tests to fail. Consequently, these parts are not covered.

In all projects, checked coverage is lower than regular coverage, with an average difference of 24%. With 37%, this difference is most pronounced for XSTREAM. This is due to a library class that directly manipulates memory and is used by XSTREAM in a central part. As this takes place outside of JAVA in native code, some dependencies cannot be traced by the slicer, which leads to statements not being considered for checked coverage, although they should.

The traditional definition of the mutation score is the number of detected mutations divided by the total number of mutations. In our setting, we only consider the score for covered mutations (last column). These values are also lowest for ASPECTJ and BARBECUE because of the reasons mentioned earlier.

10.2.2 Qualitative Analysis

In our first experiment, we were interested whether checked coverage can be used to improve the oracles of a test suite. We computed checked and statement coverage for each class individually. Then, we manually investigated those classes with a difference between checked and regular coverage, as this indicates code that is executed without checking the computation results.

In the introduction, we have seen such an example for a test of `PatternParser`, a helper class for parsing XSLT patterns, from the JAXEN project (Figure 10.1). The

corresponding test class calls the `parse()` method with valid inputs (shown in Figure 10.1) and invalid inputs, and passes when no exception or an expected exception is thrown. The computation results of the `parse()` method, however, are not checked. Consequently, this leads to a checked coverage of 0%.

```
boolean checkCreateNumber(String val){
  try {
    Object obj =
        NumberUtils.createNumber(val);
    if (obj == null) {
      return false;
    }
  return true;
  } catch (NumberFormatException e) {
    return false;
  }
}
```

Figure 10.3: Another test with insufficient outcome checks.

Another example of missing checks are the tests for the `NumberUtils` class of the COMMONS-LANG project. Some statements of the `isAllZeros()` method, which is indirectly called by the `createNumber()` method, are not checked although they are covered by the tests. The test cases exercise these statements via the `checkCreateNumber()` method shown in Figure 10.3. This method then calls `createNumber()` and returns `false` when `null` is returned or an exception is thrown, or `true` otherwise. The result of `createNumber()`, however, is not adequately checked. It is only checked whether the result is not `null`. Adding a test that checks the result of `createNumber()` would include the missing statements for the checked coverage.

```
  public String next()
      throws TokenizerException {
    nextToken();
    return current();
 }
```

Figure 10.4: A method where the return value is not checked.

Figure 10.4 shows the `next()` method from the `AbstractTokenizer` class of the JTOPAS project. Although this method is executed several times by the test suite,

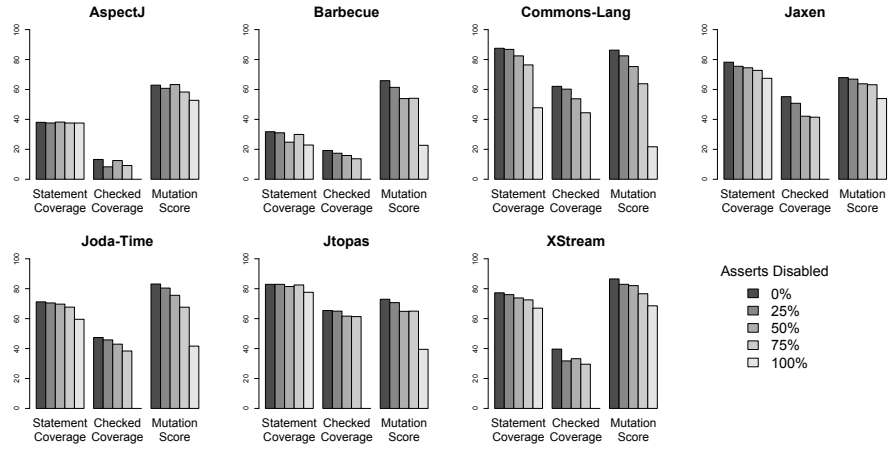

Figure 10.5: Coverage values for test suites with removed assertions.

its return value is never checked, and consequently, reported as missing from checked coverage. This means that the method could return any value and the test suite would not fail. In the same way as for the previous examples, adding an assertion which checks the return value properly solves this problem.

These examples illustrate how developers could use checked coverage in order to detect computations not checked by the test suite, and how to improve the test suite by introducing additional checks. Furthermore, these examples demonstrate that mature test suites, which have undergone several years of development, do not check all the computations that are covered by their inputs.

Mature test suites miss checks.

10.2.3 Disabling Oracles

In our first experiment, we have seen cases were a test suite covers parts of the program, but does not check the results well enough. As discussed earlier, this can be detected by checked coverage. In our second experiment, we explored the questions:

- *How sensitive is checked coverage to missing checks?*

- *How does checked coverage compare to other metrics that measure the quality of a test suite?*

To this end, we took the original JUNIT test suites of the seven projects, and produced new versions with *decayed oracles*, i.e., we systematically reduced oracle quality by removing a number of *calls to assert methods*. In JUNIT, assert methods constitute the central means for checking computation results; therefore, removing calls to assert methods means disabling checks and, therefore, reducing oracle quality.

To disable a call to an assert method in the source code, we completely removed the line that contains the call. In some cases, we had to remove additional statements, as the new test suites were not compilable anymore, or failed because they relied on the side effects of the removed assert methods. After the test suite could be successfully compiled and had no failing tests anymore, we computed the coverage metrics for each of the test suite.

An alternative way of removing calls to assert methods is to disable the method calls on bytecode level. The advantage of this approach would be that just the check is removed, and no recompilation step would be necessary. However, we choose to remove complete source code lines because we believe that this more closely simulates the scenario that a developer misses to write a check.

The results are given in Figure 10.5. For each of our subject programs there is a plot that shows the statement coverage value, checked coverage value, and mutation score for a test suite with all assertions enabled (0% removed), and with 25, 50, 75, and 100% of the assertions removed, respectively.

For almost all projects, all metric values decrease with a decreasing number of assertions. An exception is ASPECTJ where the statement coverage values stay constant for all test suites. This is due to the nature of the ASPECTJ test suite that does not have any computations inside assertions. Furthermore, the checked coverage value and the mutation score are higher for the test suite with 50% assertions removed than for the suite with 25% removed. As we chose the disabled assertions randomly each time, some assertions that check more parts of the computation were disabled in the 25% test suite and not disabled in the 50% test suite. This also explains the higher values for statement coverage in the 75% than the 50% test suite for BARBECUE, and the difference for XSTREAM and checked coverage between 50 and 25%.

All test quality metrics decrease with oracle decay.

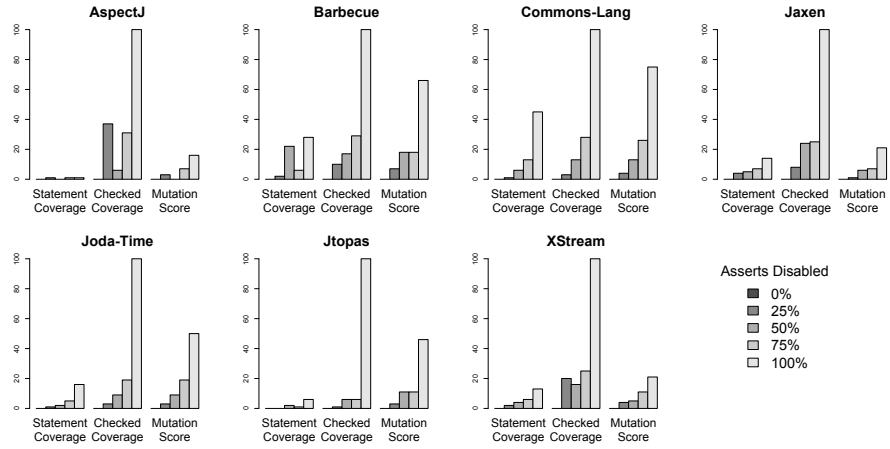

Figure 10.6: Decrease of the coverage values relative to the coverage values of the original test suite.

In order to compare the decrease of the different metrics, we computed the decrease of each metric relative to the value for the original test suite. Figure 10.6 shows the results for the seven subject programs.

For statement coverage, the decrease values are the lowest for all projects. This comes by no surprise, as it is not designed to measure the quality of checks in a test suite. Thus, it is the least sensitive metric to missing assert-statements.

Checked coverage and mutation score show a similar development for BARBECUE, COMMONS-LANG, JODA-TIME and JTOPAS for 0 to 75% of removed checks. For the other projects, there is a greater decrease for the checked coverage than for the mutation score. On average, when 75% of the tests are removed, checked coverage decreases by 23%, whereas the mutation testing score only decreases by 14%.

Checked coverage is more sensitive to missing assertions than statement coverage and mutation testing.

Note that when *all checks* are removed, checked coverage drops to 0% for all projects. This is caused by the construction of the approach, as there are no statements left to slice from once all checks are removed from the test suite. Since we are interested in *poor* rather than nonexistent oracles, the results obtained for the decays of 25 to 75% are much more meaningful.

10.2.4 Explicit and Implicit Checks

As mutation testing is also a measure of a test suite's check quality, we might also expect that the mutation score dramatically drops when no checks are left. However, in the previous experiment, we have seen that test suites with no assertions still detect a significant fraction of the mutants (43% on average). The reason for this is that a mutant can be detected by an *explicit check of the test suite*, an *explicit check of the program*, or by an *implicit check of the runtime system*. For modern object-oriented and restrictive programming languages like JAVA, many mutants are detected through these implicit runtime checks.

JAVALANCHE, the mutation testing tool used in this study, uses the distinction made by JUNIT to classify a mutation as either detected by an explicit check of the test suite or other checks. JUNIT classifies a test that does not pass either as a failure, or as an error. A test that does not pass is classified as a failure when an `AssertionError` was thrown. This error is thrown by JUNIT assert methods when a comparison failed, and by the JAVA assert keyword when its condition evaluates to `false`. Hence, failures can be caused by explicit checks of the test suite or program. Our subject programs, however, do not use the `assert` keyword. Thus, all failures are caused by explicit checks of the test suite. A test that does not pass is classified as an error, when any `Throwable` other than `AssertionError` is thrown. These exceptions can either be thrown explicitly inside the program or implicitly by the runtime system. Hence, errors can be caused by explicit checks of the program or implicit checks of the runtime system.

In order to see how the fraction of mutants detected by explicit checks (i.e. classified as failure) of the test suite changes with decreasing oracle quality, we computed this fraction for the original test suite and the test suite with all checks removed.

Table 10.2 gives the results for each project. The first two values are for the original test suite. First, the total number of detected mutants (column 2) is given and then the percentage of those that are detected by explicit checks of the test suite (column 3). The remaining mutants are detected by other checks. The two last columns give the corresponding values for the test suite with all assert-statements removed.

For the original test suite, 35 to 68% of the mutants are detected by explicit checks of the test suite, and consequently, 32 to 65% of the mutants are detected by other checks. The main fraction of the mutants detected by other checks are detected through implicit checks of the runtime system, namely `NullPointerExceptions` caused by mutants.

Table 10.2: Mutations detected by explicit checks of the test suite.

	Original test suite		All checks removed	
Project name	**Mutants detected**	**Detected by explicit test suite checks**	**Mutants detected**	**Detected by explicit test suite checks**
ASPECTJ	5736	63%	5529	53%
BARBECUE	1036	59%	357	6%
COMMONS-LANG	13415	68%	3377	16%
JAXEN	4154	38%	3298	12%
JODA-TIME	12164	56%	6094	15%
JTOPAS	1295	57%	597	0%
XSTREAM	7764	35%	5156	5%
Total	**45564**	**55%**	**24408**	**21%**

Almost half (45%) of the mutants are not detected by explicit checks of the test suite.

The test suites with all checks removed still detect 54% of the mutants that are detected by the original test suite; on average, 21% of the detected mutants are found by explicit checks of the test suites. For test suite with all checks removed, one might expect 0% of the mutants to be detected by explicit checks. However, assertions in external libraries—that are not removed— cause this 21%. Examples include the `fail()` method of JUNIT or the `verify()` method of a mocking framework.

One could argue that it does not matter how a mutation is being found—after all, the important thing is that it is found at all. Keep in mind, though, that implicit checks are not under control by the programmer. A mutation score coming from implicit checks thus reflects the *quality of the runtime checks* rather than the quality of the test suite checks. Also, the runtime system will only catch the most glaring faults (say, null pointer dereferences or out of bound accesses), but will not check any significant functional property. Therefore, having mutations fail on implicit checks only is an indicator for poor oracle quality—just as a low checked coverage.

A test suite with no assertions still detects over 50% of the mutants detected by the original test suite;
around 80% of these are detected by explicit checks of the program and implicit checks.

10.2.5 Performance

Table 10.3 shows the runtime for checked coverage and mutation testing. The checked coverage is computed in two steps. First, a run of the test suite is traced (Column 2), then, using the slicer, the checked coverage (Column 3) is computed from the previously produced trace file. Column 4 gives the total time needed to compute the checked coverage. For almost all projects, the slicing step takes much longer than tracing the test suite. XSTREAM, however, is an exception. Here the slicing takes less time because some of the central dependencies are not handled by the tracer (see Section 10.3). The last column gives the time needed for mutation testing the programs

Table 10.3: Runtime to compute the checked coverage and the mutation score.

Project name	Checked coverage			Mutation testing
	Trace	Slice	Total	
ASPECTJ	0:08:51	0:35:26	0:43:17	20:18:38
BARBECUE	0:06:10	0:15:30	0:21:40	0:06:07
COMMONS-LANG	0:32:07	3:40:37	1:12:44	1:29:06
JAXEN	0:24:21	0:37:18	1:01:39	1:29:00
JODA-TIME	0:23:53	1:38:10	2:02:03	0:45:43
JTOPAS	0:04:04	0:05:32	0:09:36	0:41:25
XSTREAM	0:40:13	0:16:35	0:56:48	1:49:13

with JAVALANCHE. When we compare the total time needed to compute the checked coverage with the time needed for mutation testing, checked coverage is faster for four of our projects and mutation testing is faster for three of the projects. Keep in mind, though, that JAVALANCHE reaches its speed only through a dramatic reduction in mutation operators; full-fledged mutation testing requires a practically unlimited number of test runs.

In terms of performance, checked coverage is on par with the fastest mutation testing tools.

10.3 Limitations

In some cases, statements that contribute to the computation of results later checked by oracles are not considered for the checked coverage due to limitations of JAVASLICER

or limitations of our approach.

Native code imposes one limitation to the tracer. In JAVA it is possible to call assembly code, written in other languages, via the JAVA Native Interface (JNI). This code cannot be accessed by the tracer as it only sees the bytecode of the classes loaded by the JVM. Regular programs rarely use this feature. In the JAVA standard library, however, there are many methods that use the JNI. Examples include the `arraycopy()` method of the `java.lang.System` class, or parts of the Reflection API. In these cases, the dependencies between the inputs and the outputs of the methods are lost. This limitation also caused the huge differences between normal coverage and checked coverage for the XSTREAM project. It uses the class `sun.misc.Unsafe` which allows direct manipulation of the memory in a core part. Therefore, many dependencies get lost and the checked coverage is lower than expected.

Another limitation imposed by the tracer is that the `String` class is currently not handled. This class is used heavily in core classes of both the JVM and the tracer which makes it difficult to instrument without running into circular dependencies. Handling this class would allow the slicer to detect dependencies that are currently missed.

```
try {
    methodThatShouldThrowException();
    fail("No exception thrown");
} catch(ExpectedException e) {
    // expected this exception
}
```

Figure 10.7: A common JUNIT pattern to check for exceptions.

A frequently used practice to check for exceptions is to call a method under such circumstances that it should throw an exception, and fail when no exception is thrown (Figure 10.7). When the exception is thrown, everything is fine, and nothing else is checked. In our setting, the statements which contribute to the exception are not on a slice. Thus, they do not contribute to the checked coverage. A remedy would be to introduce an assertion that checks for the exception.

Another limitation of our approach are computations that lead to a branch not being taken. The code shown in Figure 10.8, contains a boolean flag `inSaneState`. Later, an exception is thrown when this flag has the value `false`. Thus, only computations that lead to the variable being set to `false` can be on a dynamic slice, and computations that lead to the variable being set to `true` will *never* be on a dynamic slice. Such problems are inherent to dynamic slicing, and would best be addressed by computing static dependencies to branches not taken.

```
private boolean inSaneState = true;
...
if(!inSaneState)
    exceptionThrowingMethod();
...
```

Figure 10.8: Statements that lead to not taking a branch.

10.4 Threats to Validity

As with any empirical study, several limitations must be considered when interpreting its results.

External validity Can we generalize from the results of our study? We have investigated seven different open-source projects, covering different standards in maturity, size, domain, and test quality. But even with this variety, it is possible that our results do not generalize to other arbitrary projects.

Construct validity Are our measures appropriate for capturing the dependent variables? The biggest threat here is that our implementation could contain errors that might affect the outcome. To control for this threat, we relied on public, well-established open-source tools wherever possible; our own JAVALANCHE and JAVASLICER frameworks are publicly available as open-source packages to facilitate replication and extension of our experiments.

Internal validity Can we draw conclusions about the connections between independent and dependent variables? The biggest threat here is that we only use sensitivity to oracle decay as dependent variable—rather than a more absolute "test quality" or "oracle quality". Unfortunately, there is no objective assessment of test quality to compare against. The closest would be mutation testing [3], but as our results show, even programs without oracles can still achieve a high mutation score by relying on uncontrolled implicit checks. As it comes to internal validity, we are thus confident that sensitivity to oracle decay is the proper measure; a low checked coverage, therefore, correctly indicates a low oracle quality.

10.5 Related Work

10.5.1 Coverage Metrics

During structural testing, a program is tested using knowledge of its internal structures. Hereby, one is interested in the *quality* of the developed tests, and how to *improve* them in order to detect possible errors. To this end, different coverage metrics (see Section 2.3) have been proposed and compared against each other regarding their effectiveness in detecting specific types of errors, relative costs, and difficulty of satisfying them [42, 65, 104, 27]. Each coverage metric requires different items to be covered. This allows to compute a *coverage score* by dividing the number of coverable items by the number of items actually covered.

Best known, and most easy to compute is *statement coverage*. It simply requires each line to be executed at least once. Because some defects can only occur under specific conditions, more complex metrics have been proposed. The popular *branch coverage* requires each condition to evaluate to both `true` and `false` at least once; *decision coverage* extends this condition to boolean subexpressions in control structures. A more theoretical metric is *path coverage,* measuring how many of the (normally infinitely many) possible paths have been followed. Similar to our approach *data flow testing criteria* [83] also relate definitions and uses of variables. These techniques consider the relation between all defined variables inside the program and their uses. For example, the *all-uses* criterion requires that for each definition use pair a path is exercised that covers this pair. In contrast, our approach is only targeted at uses inside the oracles. Other definition use pairs are followed transitively from there.

Each of these proposed metrics just measures how well specific structures are exercised by the provided test input, and not how well the outputs of the program are checked. Thus, they *do not assess oracle quality* of a test suite.

10.5.2 Mutation Testing

A technique that aims at checking the quality of the oracles is *mutation testing* (see Chapter 3). Originally proposed by Richard Lipton [73, 18], mutation testing seeds artificial defects, as defined by *mutation operators*, into a program and checks whether the test suite can distinguish the mutated from the original version. A mutation is supposed to be detected (“killed”) if at least one test case fails on the mutated version

that passed on the original program. If a mutation is not detected by the test suite, similar defects might be in the program that have not been detected as well. Thus, these undetected mutants can give an indication on how to improve the test inputs and checks. However, not every undetected mutant helps in improving the test suite as it might also be an *equivalent mutant*; that is a mutation that changes the syntax but not the semantics of a program. In our experiments, we have also seen that mutation testing measures the quality of all types of checks, explicit checks of the test suite and program, and implicit checks of the runtime system, while checked coverage is focused on explicit checks of the test suite.

10.5.3 Program Slicing

Static program slicing was originally proposed by Weiser [102, 103] as a technique that helps the programmer during debugging. Korel and Laski [47] introduced dynamic slicing that computes slices for a concrete program run. Furthermore, different slicing variants have been proposed for program comprehension; *Conditioned Slicing* [54] is a mix between dynamic and static slicing, it allows some variables to have a fixed value while others can take all possible values. *Amorphous Slicing* [37] requires a slice only to preserve the semantics of the behavior of interest, while syntax can be arbitrarily changed, which allows to produce smaller slices.

Besides its intended use in debugging, program slicing has been applied to many different areas [97, 38]. Examples include minimization of generated test cases [50], automatic parallelization [34, 97], and the detection of equivalent mutants [39].

10.5.4 State Coverage

The concept closest to checked coverage is *state coverage* proposed by Koster and Kao [49]. It also measures the quality of checks in a test suite. To this end, all output defining statements (ODS) are considered. Output defining statements are statements that define a variable that can be checked by the test suite for a concrete run. The state coverage is defined as the number of ODS that are on a dynamic slice from a check divided by the total number of ODS. This differs from our approach, as we also consider statements that influence the computation of variables that are checked.

Furthermore, the number of ODS is dependent on the test input. For different inputs, different statements are considered as output defining. This can lead to cases

where a test suite is improved by adding additional tests (with new inputs), but the state coverage drops. Checked coverage stays constant or improves in such cases.

Unfortunately, there is no broader evaluation of state coverage that we can compare against. In a first short paper [49], a proof of concept based on a static slicer, and one small experiment is presented. A second short paper [48] describes an implementation based on *taint analysis* [13], but no experimental evaluation is provided.

10.6 Summary

In this chapter, we presented checked coverage. Checked coverage is a coverage metric that aims to measure how well the results of the program are checked by the test suite. This is in contrast to traditional metrics that only measure how well the test inputs exercise the program. Technically, checked coverage requires every statement to be on a dynamic backward slice from one of the explicit checks of the test suite.

We presented an implementation of checked coverage for JAVA that is based on the JAVASLICER by Hammacher. In a study on seven open-source programs, we have seen that even mature test suites miss checks, and that we can use checked coverage to detect them. A difference in the coverage level between the checked coverage and statement coverage indicates computations that are not well checked. In an experiment, we investigated the sensitivity of different metrics with regard to missing explicit checks. We compared checked coverage, mutation testing, and statement coverage. To this end, we removed a fraction of explicit checks from existing test suites and observed the change in the coverage levels of the different metrics. In comparison to the other metrics, checked coverage is most sensitive to missing assertions. Although the coverage level for the other metrics does also decrease when explicit checks are removed this effect is most pronounced for checked coverage. Because mutation testing was designed to measure the quality of checks, we then investigated why mutation testing did not react as strongly to missing checks as checked coverage. The reason for this is that mutations can be detected by three types of checks. They can be detected by implicit checks of the runtime system, explicit checks in the program, and explicit checks in the test suite. In an experiment, we compared the detection rates of the different type of checks for the original test suites and test suites with all explicit checks removed. For the original test suite the results indicated that 45% of the mutations were detected by implicit checks or explicit checks of the program. When we removed all checks, still 54% of the previously detected mutations were detected and 79% of them by implicit checks

the runtime system or explicit checks of the program. The remaining 21% were due to external test libraries from which we did not remove explicit checks.

Chapter 11

Conclusions and Future Work

The quality of checks plays an important role in a unit test's ability to detect defects. This is because a defect manifests itself during a program run through an infected state which propagates and results in a failure. Good checks help to distinguish normal expected behavior from unexpected failures, and thereby, to detect defects.

Coverage metrics, which are the state of the art for assessing the quality of unit tests, only focus on the quality of test inputs. An alternative technique that measures the quality of test inputs and checks is mutation testing. However, it also has two drawbacks. It is computationally expensive and equivalent mutants dilute the quality of its results.

This work makes the following contributions to improve mutation testing, and presented a novel alternative metric to assess the quality of a test suite's checks:

- We introduced JAVALANCHE, a mutation testing framework for JAVA, that was developed with a focus on automation and efficiency. To enable efficient mutation testing, JAVALANCHE applies several optimization techniques, and in our experiments we have shown that it scales to real-life programs. JAVALANCHE is published as open-source, and has been the basis for further experiments presented in this work. Moreover, it has been successfully used by other researchers. For example, it has been extended for mutation testing of multi-threaded code [29], used by an approach to automatically generate test oracles [28], used to assess the quality of test suites generated by students [1], and to compare different test generation strategies [92].

- We studied the extend of equivalent mutants on real-life programs, and the results show that equivalent mutants are a serious problem that effectively inhibits widespread usage of mutation testing. In a sample taken from seven JAVA programs about 45% of the undetected mutants are equivalent, and classifying a mutant took on average about 15 minutes.

- We introduced impact measures as a method to separate equivalent from non-equivalent mutants. The impact of a mutation is the difference between a run of the test suite on the original program, and a run on the mutated program. In this work, we presented impact metrics based on dynamic invariants, covered statements, and return values. Invariant impact measures how many invariants, obtained from the original program are violated by a mutated version. Coverage and data impact measure the differences in the execution frequency of statements and in return values of public methods between a run of the original program and a mutant. The results indicate that all impact metrics are well suited to detect non-equivalent mutants, i.e. when a mutation has an impact it is very likely that it is non-equivalent. However, in contrast to invariant impact, coverage and data impact are effective means to separate equivalent from non-equivalent mutants, i.e. when a mutation has no impact it is more likely that it is equivalent.

- We presented checked coverage, an alternative method to mutation testing, to assess the quality of the test suite's checks. It measures which parts of the covered computation are actually checked by oracles. Technically, it requires each statement to be on at least one dynamic backward slice from an explicit check of the test suite. Our experiments show that checked coverage and mutation testing are sensitive to oracle quality, and that checked coverage is more sensitive to missing explicit checks in the test suite.

To sum up, this works advances the state of the art by introducing the first approach to detect non-equivalent mutants on larger programs, and by introducing a novel coverage metric that assesses the quality of the checks. These approaches can be further extended and the following topics can be investigated:

Extensions to JAVALANCHE Currently, JAVALANCHE supports a limited set of mutation operators. For some scenarios, however, more mutation operators are needed. JAVALANCHE can be extended to support more mutation operators, and to apply mutations in source code. Some mutation operators can only be applied to source code. This is because there is not enough information available at the bytecode level to carry out these mutations. Examples include object-oriented

mutations which correspond to one single change on the source code level and several changes on the bytecode level. Although it is more costly to manipulate source code because a recompiling step is needed, the advantage is that more mutation operators can be supported.

Alternative impact measures While we consider violations of dynamic invariants, and changes in coverage and return values to be particularly useful predictors of failures, there are many ways to determine the impact of a change. One can measure impact in anything that characterizes a run; including different coverage criteria, sequences of executed methods [15], program spectra [84] or numerical ranges of data and increments [36]. Furthermore, several different impact metrics might be combined as they measure different aspects of a program run that can be impacted by a mutation.

Include static analysis In this work, we presented dynamic analyses approaches to separate non-equivalent from equivalent mutants. We choose dynamic analyses because our main goal was to have a scalable technique that works on real-life programs. However, static analysis techniques can also detect equivalent mutants for these programs. If it can be statically proved that a mutated statement cannot be reached, that a mutation cannot change the program state, or that an infection cannot propagate the mutation is equivalent. Our techniques would benefit from applying those analyses beforehand, because these mutations do not have to be considered for further investigation.

Extend checked coverage Checked coverage requires the tests to check each computation. The concept, that computations exercising a structure have to be checked, can be carried over to other coverage metrics. For example, branch coverage can be combined with the concept of checked coverage by requiring that the statement controlling the execution of a branch is on a dynamic slice when the branch is taken and when it is not taken. These metrics would impose more complex requirements on the inputs while ensuring that the computed results are checked. Another aspect which can be extended are the type of checks considered by checked coverage. Currently, only explicit checks of the test suite are considered. Checked coverage can be extended to also account for explicit checks of the program. This can be done by including the backward slice from assert statements in the code and from conditions that control branches leading to explicitly thrown exceptions.

Bibliography

[1] AALTONEN, K., IHANTOLA, P., AND SEPPÄLÄ, O. Mutation analysis vs. code coverage in automated assessment of students' testing skills. In *SPLASH/OOPSLA '10: Companion to the 25th Conference on Object-Oriented Programming, Systems, Languages, and Applications* (2010), pp. 153–160.

[2] AMMANN, P., AND OFFUTT, J. *Introduction to Software Testing*, 2 ed. Cambridge University Press, 2008.

[3] ANDREWS, J. H., BRIAND, L. C., AND LABICHE, Y. Is mutation an appropriate tool for testing experiments? In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering* (2005), pp. 402–411.

[4] ANDREWS, J. H., BRIAND, L. C., LABICHE, Y., AND NAMIN, A. S. Using mutation analysis for assessing and comparing testing coverage criteria. *Transactions of Software Engineering 32*, 8 (2006), 608–624.

[5] BALL, T., AND LARUS, J. R. Efficient path profiling. In *MICRO '96: Proceedings of the 29th International Symposium on Microarchitecture* (1996), pp. 46–57.

[6] BARBOSA, E. F., MALDONADO, J. C., AND VINCENZI, A. M. R. Toward the determination of sufficient mutant operators for C. *Software Testing, Verification Reliability (STVR) 11*, 2 (2001), 113–136.

[7] BEVAN, J., WHITEHEAD, JR., E. J., KIM, S., AND GODFREY, M. Facilitating software evolution research with kenyon. In *ASE' 05: Proceedings of the 10th European Software Engineering Conference held jointly with 13th International Symposium on the Foundations of Software Engineering* (2005), pp. 177–186.

[8] BOYAPATI, C., KHURSHID, S., AND MARINOV, D. Korat: Automated testing based on Java predicates. In *ISSTA '02: Proceedings of the 13th International Symposium on Software Testing and Analysis* (2002), pp. 123–133.

[9] BRUNETON, E., LENGLET, R., AND COUPAYE, T. ASM: A code manipulation tool to implement adaptable systems. In *Proceedings of the Workshop on Adaptable and Extensible Component Systems (Systèmes à composants adaptables et extensibles)* (2002).

[10] BUDD, T. A., AND ANGLUIN, D. Two notions of correctness and their relation to testing. *Acta Informatica 18*, 1 (1982), 31–45.

[11] BUDD, T. A., AND GOPAL, A. S. Program testing by specification mutation. *Computer Languages 10*, 1 (1985), 63–73.

[12] BUDD, T. A., LIPTON, R. J., DEMILLO, R., AND SAYWARD, F. The design of a prototype mutation system for program testing. In *Proceedings of the 1978 International Workshop on Managing Requirements Knowledge* (1978), pp. 623–627.

[13] CLAUSE, J. A., LI, W., AND ORSO, A. Dytan: A generic dynamic taint analysis framework. In *ISSTA '07: Proceedings of the 16th International Symposium on Software Testing and Analysis* (2007), pp. 196–206.

[14] DALLMEIER, V., LINDIG, C., WASYLKOWSKI, A., AND ZELLER, A. Mining object behavior with adabu. In *WODA 2006: Proceedings of the 2006 international workshop on Dynamic systems analysis* (2006), pp. 17–24.

[15] DALLMEIER, V., LINDIG, C., AND ZELLER, A. Lightweight defect localization for Java. In *ECOOP '05: Proceedings of 19th European Conference on Object-Oriented Programming* (2005), pp. 528–550.

[16] DELAMARO, M. E., AND MALDONADO, J. C. Proteum-a tool for the assessment of test adequacy for c programs. In *PCS '96: Proceedings of the Conference on Performability in Computing Systems* (1996), pp. 79–95.

[17] DELAMARO, M. E., MALDONADO, J. C., AND MATHUR, A. P. Integration testing using interface mutations. In *ISSRE '96: Proceedings of the 7th International Symposium on Software Reliability Engineering* (1996), pp. 112–121.

[18] DEMILLO, R. A., LIPTON, R. J., AND SAYWARD, F. G. Hints on test data selection: Help for the practicing programmer. *Computer 11*, 4 (1978), 34–41.

[19] DO, H., ELBAUM, S., AND ROTHERMEL, G. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering 10*, 4 (2005), 405–435.

[20] DOWSON, M. The ariane 5 software failure. *Software Engineering Notes 22*, 2 (1997), 84.

[21] ELBAUM, S., GABLE, D., AND ROTHERMEL, G. The impact of software evolution on code coverage information. In *ICSM '01: Proceedings of the 17th International Conference on Software Maintenance* (2001), pp. 170–179.

[22] ELLIMS, M., INCE, D., AND PETRE, M. The csaw c mutation tool: Initial results. In *Mutation '07: Proceedings of the 3rd International Workshop on Mutation Analysis* (2007), pp. 185–192.

[23] ERNST, M. D., COCKRELL, J., GRISWOLD, W. G., AND NOTKIN, D. Dynamically discovering likely program invariants to support program evolution. *Transactions on Software Engineering 27*, 2 (2001), 99–123.

[24] FLANAGAN, C., LEINO, K. R. M., LILLIBRIDGE, M., NELSON, G., SAXE, J. B., AND STATA, R. Extended static checking for Java. In *PLDI '02: Proceedings of the 23rd Conference on Programming Language Design and Implementation* (2002), 201-212, Ed.

[25] FRANKL, P., AND WEISS, S. An experimental comparison of the effectiveness of branch testing and data flow testing. *Transactions on Software Engineering 19*, 8 (1993), 774–787.

[26] FRANKL, P. G., AND IAKOUNENKO, O. Further empirical studies of test effectiveness. In *FSE '98: Proceedings of the 6th International Symposium on the Foundations of Software Engineering* (1998), pp. 153–162.

[27] FRANKL, P. G., WEISS, S. N., AND HU, C. All-uses versus mutation testing: An experimental comparison of effectiveness. *Journal of Systems and Software 38* (1997), 235–253.

[28] FRASER, G., AND ZELLER, A. Mutation-driven generation of unit tests and oracles. In *ISSTA '10: Proceedings of the 19th International Symposium on Software Testing and Analysis* (2010), pp. 147–158.

[29] GLIGORIC, M., JAGANNATH, V., AND MARINOV, D. Mutmut: Efficient exploration for mutation testing of multithreaded code. In *ICST '10: Proceedings*

of the 3rd International Conference on Software Testing, Verification and Validation (2010), pp. 55–64.

[30] GODEFROID, P., KLARLUND, N., AND SEN, K. Dart: directed automated random testing. In *PLDI '05: Proceedings of the 26th Conference on Programming Language Design and Implementation* (2005), pp. 213–223.

[31] GORADIA, T. Dynamic impact analysis: A cost-effective technique to enforce error-propagation. In *ISSTA '93: Proceedings of the 8th International Symposium on Software Testing and Analysis* (1993), pp. 171–181.

[32] GRÜN, B. J. M., SCHULER, D., AND ZELLER, A. The impact of equivalent mutants. In *Mutation '09: Proceedings of the 4th International Workshop on Mutation Analysis* (2009), pp. 192–199.

[33] HAMMACHER, C. Design and Implementation of an Efficient Dynamic Slicer for Java. Bachelor's thesis, Saarland University, November 2008.

[34] HAMMACHER, C., STREIT, K., HACK, S., AND ZELLER, A. Profiling Java programs for parallelism. In *IWMSE'09: Proceedings of the 2nd International Workshop on Multi-Core Software Engineering* (2009), pp. 49–55.

[35] HAMPTON, M., AND STEPHANE, P. Leveraging a commercial mutation analysis tool for research. In *Mutation '07: Proceedings of the 3rd International Workshop on Mutation Analysis* (2007), pp. 203–209.

[36] HANGAL, S., AND LAM, M. S. Tracking down software bugs using automatic anomaly detection. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering* (2002), pp. 291–302.

[37] HARMAN, M., BINKLEY, D., AND DANICIC, S. Amorphous program slicing. *Journal of Systems and Software 68*, 1 (2003), 45–64.

[38] HARMAN, M., AND HIERONS, R. M. An overview of program slicing. *Software Focus 2*, 3 (2001), 85–92.

[39] HIERONS, R., AND HARMAN, M. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability 9*, 4 (1999), 233–262.

[40] HOWDEN, W. E. Weak mutation testing and completeness of test sets. *Transactions on Software Engineering 8*, 4 (1982), 371–379.

[41] HU, J., LI, N., AND OFFUTT, J. An analysis of oo mutation operators. In *Mutation '11: Proceedings of the 6th International Workshop on Mutation Analysis (to appear)* (2011).

[42] HUANG, J. C. An approach to program testing. *Computing Surveys 7*, 3 (1975), 113–128.

[43] HUTCHINS, M., FOSTER, H., GORADIA, T., AND OSTRAND, T. J. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *ICSE '94: Proceedings of the 16th International Conference on Software Engineering* (1994), pp. 191–200.

[44] JIA, Y., AND HARMAN, M. An analysis and survey of the development of mutation testing. *Transactions on Software Engineering (to appear)*.

[45] JONES, J. A., AND HARROLD, M. J. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE '05: Proceedings of the 20th international Conference on Automated Software Engineering* (2005), pp. 273–282.

[46] KIM, S., ZIMMERMANN, T., JR., E. J. W., AND ZELLER, A. Predicting faults from cached history. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering* (2007), pp. 489–498.

[47] KOREL, B., AND LASKI, J. Dynamic program slicing. *Information Processing Letters 29*, 3 (1988), 155–163.

[48] KOSTER, K. A state coverage tool for JUnit. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering* (2008), pp. 965–966.

[49] KOSTER, K., AND KAO, D. State coverage: A structural test adequacy criterion for behavior checking. In *ESEC/FSE '07: Proceedings of the 11th European Software Engineering Conference held jointly with 15th International Symposium on the Foundations of Software Engineering* (2007), pp. 541–544.

[50] LEITNER, A., ORIOL, M., ZELLER, A., CIUPA, I., AND MEYER, B. Efficient unit test case minimization. In *ASE '07: Proceedings of the 22nd international conference on Automated Software Engineering* (2007), pp. 417–420.

[51] LEVESON, N. An investigation of the therac-25 accidents. *Computer 26*, 7 (1993), 18–41.

[52] LI, Z., TAN, L., WANG, X., LU, S., ZHOU, Y., AND ZHAI, C. Have things changed now? An empirical study of bug characteristics in modern open source software. In *ASID '06 Proceedings of the 1st workshop on Architectural and system support for improving software dependability* (2006), pp. 25–33.

[53] LIVSHITS, B., AND ZIMMERMANN, T. Dynamine: finding common error patterns by mining software revision histories. In *ESEC/FSE '05: Proceedings of the 10th European Software Engineering Conference held jointly with 13th International Symposium on the Foundations of Software Engineering* (2005), pp. 296–305.

[54] LUCIA, A. D., FASOLINO, A. R., AND MUNRO, M. Understanding function behaviors through program slicing. In *WPC '96: Proceedings of the 4th International Workshop on Program Comprehension* (1996), pp. 9–18.

[55] MA, Y.-S., OFFUTT, J., AND KWON, Y.-R. MuJava: a mutation system for Java. In *ICSE '06: Proceedings of the 28th International Conference on Software Engineering* (2006), pp. 827–830.

[56] MASRI, W., ABOU-ASSI, R., EL-GHALI, M., AND AL-FATAIRI, N. An empirical study of the factors that reduce the effectiveness of coverage-based fault localization. In *DEFECTS '09: Proceedings of the 2nd International Workshop on Defects in Large Software Systems* (2009), pp. 1–5.

[57] MATHUR, A. P. Performance, effectiveness, and reliability issues in software testing. In *COMPSAC '91: Proceedings of the 15th International Computer Software and Applications Conference* (1991), pp. 604–605.

[58] MCCAMANT, S., AND ERNST, M. D. Predicting problems caused by component upgrades. In *ESEC/FSE '03: Proceedings of the 9th European Software Engineering Conference held jointly with 11th International Symposium on the Foundations of Software Engineering* (2003), pp. 287–296.

[59] MEYER, B. *Object-Oriented Software Construction*, 2nd ed. Prentice-Hall, 1997.

[60] NAM, J., SCHULER, D., AND ZELLER, A. Calibrated mutation testing. In *Mutation '11: Proceedings of the 6th International Workshop on Mutation Analysis (to appear)* (2011).

[61] NAMIN, A. S., ANDREWS, J. H., AND MURDOCH, D. J. Sufficient mutation operators for measuring test effectiveness. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering* (2008), pp. 351–360.

[62] NETHERCOTE, N., AND SEWARD, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI '07: Proceedings of the 28th Conference on Programming Language Design and Implementation* (2007), pp. 89–100.

[63] NEVILL-MANNING, C. G., AND WITTEN, I. H. Linear-time, incremental hierarchy inference for compression. In *DCC '97: Proceedings of the 7th Data Compression Conference* (1997), pp. 3–11.

[64] NEVILL-MANNING, C. G., WITTEN, I. H., AND MAULSBY, D. Compression by induction of hierarchical grammars. In *DCC '94: Proceedings of the 4th Data Compression Conference* (1994), pp. 244–253.

[65] NTAFOS, S. C. A comparison of some structural testing strategies. *Transactions on Software Engineering 14*, 6 (1988), 868–874.

[66] OFFUTT, A. J. The coupling effect: fact or fiction. *Software Engineering Notes 14* (1989), 131–140.

[67] OFFUTT, A. J. Investigations of the software testing coupling effect. *Transactions on Software Engineering and Methodology 1*, 1 (1992), 5–20.

[68] OFFUTT, A. J., AND CRAFT, W. M. Using compiler optimization techniques to detect equivalent mutants. *Software Testing, Verification, and Reliability 4* (1994), 131–154.

[69] OFFUTT, A. J., LEE, A., ROTHERMEL, G., UNTCH, R. H., AND ZAPF, C. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology (TOSEM) 5*, 2 (1996), 99–118.

[70] OFFUTT, A. J., AND PAN, J. Detecting equivalent mutants and the feasible path problem. In *COMPASS '96: Proceedings of the 11th Conference on Computer Assurance* (1996), pp. 224–236.

[71] OFFUTT, A. J., AND PAN, J. Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification, and Reliability 7*, 3 (1997), 165–192.

[72] OFFUTT, A. J., PARGAS, R. P., FICHTER, S. V., AND KHAMBEKAR, P. K. Mutation testing of software using a mimd computer. In *ICPP '92: Proceedings of the 21st International Conference on Parallel Processing* (1992), pp. 257–266.

[73] OFFUTT, A. J., AND UNTCH, R. H. Mutation 2000: Uniting the orthogonal. In *Mutation '00: Proceedings of the 2nd International Workshop on Mutation Analysis (Mutation testing for the new century)* (2001), pp. 34–44.

[74] OFFUTT, VI, A. J., AND KING, K. N. A fortran 77 interpreter for mutation analysis. In *Proceedings of the 1st Symposium on Interpreters and interpretive techniques* (1987), pp. 177–188.

[75] ORSO, A., APIWATTANAPONG, T., AND HARROLD, M. J. Leveraging field data for impact analysis and regression testing. In *ESEC/FSE '03: Proceedings of the 9th European Software Engineering Conference held jointly with 11th International Symposium on the Foundations of Software Engineering* (2003), pp. 128–137.

[76] ORSO, A., APIWATTANAPONG, T., LAW, J., ROTHERMEL, G., AND HARROLD, M. J. An empirical comparison of dynamic impact analysis algorithms. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering* (2004), pp. 491–500.

[77] PACHECO, C., AND ERNST, M. D. Eclat: Automatic generation and classification of test inputs. In *ECOOP '05: Proceedings of the 9th European Conference on Object-Oriented Programming* (2005), pp. 504–527.

[78] PAN, K., KIM, S., AND WHITEHEAD, JR., E. J. Toward an understanding of bug fix patterns. *Empirical Software Engineering 14* (2009), 286–315.

[79] PAPI, M. M., ERNST, D., AND SMITH, C. Practical pluggable types for Java. In *ISSTA '08: Proceedings of the 17th International Symposium on Software Testing and Analysis* (2008), pp. 201–212.

[80] PERSON, S., DWYER, M. B., ELBAUM, S., AND PĂSĂREANU, C. S. Differential symbolic execution. In *FSE '08: Proceedings of the 16th International Symposium on the Foundations of Software Engineering* (Atlanta, Georgia, 2008).

[81] PEZZE, M., AND YOUNG, M. *Software Testing and Analysis: Process, Principles and Techniques*. John Wiley & Sons, 2008.

[82] RADIO TECHNICAL COMMISSION FOR AERONAUTICS (RTCA). *DO-178B: Software Considerations in Airborne Systems and Equipment Certification*, 1982.

[83] RAPPS, S., AND WEYUKER, E. J. Data flow analysis techniques for test data selection. In *ICSE '82: Proceedings of the 6th International Conference on Software Engineering* (1982), pp. 272–278.

[84] REPS, T., BALL, T., DAS, M., AND LARUS, J. The use of program profiling for software maintenance with applications to the year 2000 problem. In *ESEC/FSE '99: Proceedings of the 7th European Software Engineering Conference held jointly with 7th International Symposium on the Foundations of Software Engineering* (1999), pp. 432–449.

[85] ROTHERMEL, G., UNTCH, R. H., CHU, C., AND HARROLD, M. J. Prioritizing test cases for regression testing. *Transactions on Software Engineering 27*, 10 (2001), 929–948.

[86] RYDER, B. G., AND TIP, F. Change impact analysis for object-oriented programs. In *PASTE '01: Proceedings of the 3rd Workshop on Program Analysis for Software Tools and Engineering* (2001), pp. 46–53.

[87] SCHULER, D., DALLMEIER, V., AND ZELLER, A. Efficient mutation testing by checking invariant violations. In *ISSTA '09: Proceedings of the 18th International Symposium on Software Testing and Analysis* (2009), pp. 69–80.

[88] SCHULER, D., AND ZELLER, A. Javalanche: Efficient mutation testing for Java. In *ESEC/FSE '09: Proceedings of the 12th European Software Engineering Conference held jointly with 17th International Symposium on the Foundations of Software Engineering* (2009), pp. 297–298.

[89] SCHULER, D., AND ZELLER, A. (Un-)Covering equivalent mutants. In *ICST '10: Proceedings of the 3rd International Conference on Software Testing, Verification and Validation* (2010), pp. 45–54.

[90] SCHULER, D., AND ZELLER, A. Assessing oracle quality with checked coverage. In *ICST '11: Proceedings of the 4th International Conference on Software Testing, Verification and Validation* (2011), pp. 90–99.

[91] SEN, K., MARINOV, D., AND AGHA, G. Cute: A concolic unit testing engine for C. In *ESEC/FSE '05: Proceedings of the 10th European Software Engineering Conference held jointly with 13th International Symposium on the Foundations of Software Engineering* (2005), pp. 263–272.

[92] SHARMA, R., GLIGORIC, M., ARCURI, A., FRASER, G., AND MARINOV, D. Testing container classes: Random or systematic? In *FASE '11: Proceedings*

of the 14th International Conference on Fundamental Approaches to Software Engineering (2011), pp. 262–277.

[93] SOSIČ, R., AND ABRAMSON, D. A. Guard: A relative debugger. *Software Practice and Experience 27*, 2 (1997), 185–206.

[94] SPAFFORD, E. H. Extending mutation testing to find environmental bugs. Tech. rep., Purdue University, 1990.

[95] TASSEY, G. The economic impacts of inadequate infrastructure for software testing. Tech. rep., National Institute of Standards and Technology, 2002.

[96] TILLMANN, N., AND DE HALLEUX, J. Pex: white box test generation for .net. In *TAP '08: Proceedings of the 2nd International Conference on Tests and Proofs* (2008), pp. 134–153.

[97] TIP, F. A survey of program slicing techniques. *Journal of Programming Language 3*, 3 (1995), 121–189.

[98] UNTCH, R. H., OFFUTT, A. J., AND HARROLD, M. J. Mutation analysis using mutant schemata. In *ISSTA '93: Proceedings of the 8th International Symposium on Software Testing and Analysis* (1993), pp. 139–148.

[99] VISSER, W., PĂSĂREANU, C. S., AND KHURSHID, S. Test input generation with Java pathfinder. In *ISSTA '04: Proceedings of the 14th International Symposium on Software Testing and Analysis* (2004), pp. 97–107.

[100] WANG, T., AND ROYCHOUDHURY, A. Using compressed bytecode traces for slicing Java programs. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering* (2004), pp. 512–521.

[101] WANG, T., AND ROYCHOUDHURY, A. Dynamic slicing on Java bytecode traces. *Transactions on Programming Languages and Systems 30*, 2 (2008), 10:1–10:49.

[102] WEISER, M. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, 1979.

[103] WEISER, M. Program slicing. *Transactions on Software Engineering 10*, 4 (1984), 352–357.

[104] WEYUKER, E. J., WEISS, S. N., AND HAMLET, R. G. Comparison of program testing strategies. In *ISSTA '91: Proceedings of the 7th International Symposium on Software Testing and Analysis* (1991), pp. 1–10.

[105] WHEELER, D. A. Sloccount: http://www.dwheeler.com/sloccount/, 2004.

[106] ZELLER, A., AND HILDEBRANDT, R. Simplifying and isolating failure-inducing input. *Transactions of Software Engineering 28*, 2 (2002), 183–200.

[107] ZHANG, L., HOU, S.-S., HU, J.-J., XIE, T., AND MEI, H. Is operator-based mutant selection superior to random mutant selection? In *ICSE '10: Proceedings of the 32nd International Conference on Software Engineering* (2010), pp. 435–444.

[108] ZIMMERMANN, T., WEISSGERBER, P., DIEHL, S., AND ZELLER, A. Mining version histories to guide software changes. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering* (2004), pp. 563–572.