Universität Ulm
Institut für Angewandte Informationsverarbeitung

On Effective and Efficient Mutation Analysis for Unit and Integration Testing

Dissertation

Dissertation zur Erlangung des Doktorgrades Dr. rer. nat. an der Fakultät für Ingenieurwissenschaften und Informatik der Universität Ulm

vorgelegt von

René Just

aus Berlin

Oktober 2012

Amtierender Dekan:
Prof. Dr.-Ing. Klaus Dietmayer

Gutachter:
1. Gutachter: Prof. Dr. Franz Schweiggert
2. Gutachter: Prof. Dr. Helmuth Partsch
3. Gutachter: Dr. Gordon Fraser

Tag der Promotion: 24. Januar 2013

Abstract

Software testing is the most common technique to verify that a program meets certain quality standards. Sufficient manual testing is not only time consuming but also a complex and error-prone task. Additionally, due to the rapidly growing size and complexity of software systems, automating the software testing process is desirable for more cost-effective testing. Besides automating the testing process, the quality of the applied testing strategy has to be assessed to achieve reliable results. This is of particular importance to ensure that the employed tests are also effective in terms of their fault-finding capabilities, and hence the results adequately reflect the quality of the software. By focusing on mutation analysis and testing with partial oracles, this thesis addresses the automation and assessment of unit and integrations tests.

Mutation analysis, which is based on seeding artificial faults, is a well-known technique for the evaluation of a test suite's quality. It is, however, also associated with high, sometimes even prohibitive, costs, still preventing this technique from being widely applicable for large software projects. This thesis addresses this challenge and presents techniques to increase the efficiency of mutation testing. In addition, it describes and evaluates approaches that also improve the expressiveness of the corresponding metric.

While mutation testing is known to be computationally expensive and time consuming, it also lacks proper tool support for various purposes. Therefore, this thesis also presents a versatile and highly configurable mutation framework that implements the suggested approaches to enable further research as well as the application of efficient and effective mutation analysis for large software systems.

The automation of software tests often results in the oracle problem, another crucial challenge in software testing. In an attempt to alleviate this problem, leveraging partial oracles seem to be viable solution but their adequacy for different testing purposes has not been examined sufficiently. Therefore, this thesis investigates whether partial oracles are in principal adequate for unit and integration testing. By employing mutation analysis for this purpose, the thesis also analyzes how such partial oracles can be improved.

Acknowledgement

First of all, I would like to thank my supervisor Franz Schweiggert for all his help, guidance, and support while working on my PhD. I would also like to thank Helmuth Partsch and Gordon Fraser for being the second and third reviewer of this thesis. Also, I am grateful to Gregory M. Kapfhammer for his support and many valuable comments. Further thanks go to Kristina Steih, Johannes Mayer, Norbert Heidenbluth, and Steffen Fritzsche for many fruitful discussions and the always enjoyable working atmosphere. Finally, I am indebted to my wife Claudia for always encouraging and supporting me.

This work was partially supported by the Landesgraduiertenförderung Baden-Württemberg and by the Wilken Stiftung.

Contents

List of Figures

List of Tables

Part I

Preface and Introduction

Chapter 1

Preface

Software testing and debugging are crucial parts of the software development process and the costs associated with these techniques are estimated to constitute between 30 and 50% of the complete development process costs [Myers et al., 2011; Spillner et al., 2007]. This ratio is even considerably larger in safety critical environments, such as nuclear or avionic systems, where a more comprehensive test process is required.

Since the applicability of formal proofs of correctness are limited to small artifacts and programs, software testing is usually applied to increase the confidence in the developed system. Software testing cannot prove a system to be correct unless it is done exhaustively. This is, however, not feasible in general since the number of inputs and corresponding values leads to a tremendous number of combinations. As a consequence, software testing is usually done systematically to eliminate certain types of errors and to achieve some pre-defined end-of-test criteria.

Due to continuously growing software systems and the demand for reproducibility, automating the software test process is desirable. Moreover, the quality of software tests has to be assessed to ensure that the possibly automated tests are not only efficient but also effective. Therefore, this thesis addresses the generation as well as the evaluation of software tests by focusing on effective and efficient mutation analysis for unit and integration testing.

1.1 The Big Picture

Concerning the contributions of this thesis, Figure 1.1 illustrates the addressed areas and also connects these areas to the corresponding parts and chapters. Moreover, the diagram illustrates the connections between the different research fields and also shows perspectives and possible future work.

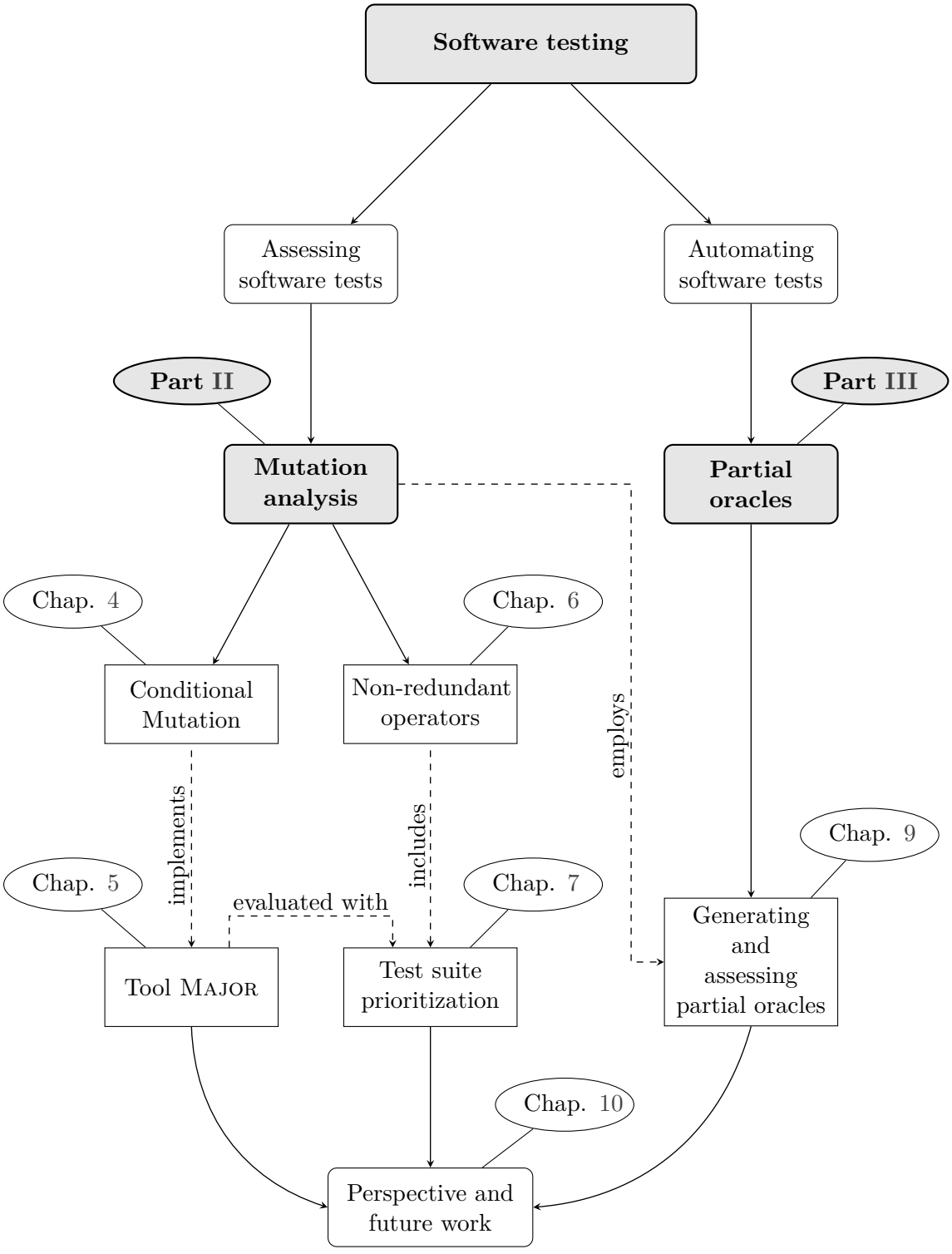


Figure 1.1: The big picture of the software testing areas addressed in this thesis.

1.2 Contributions

This thesis makes several contributions to the ongoing research in the field of software testing, especially mutation testing and testing with partial oracles. The suggested approaches and presented results within this thesis have been published in refereed publications and address the following aspects:

- Existing methods for cost reduction of mutation analysis are still not sufficient to enable efficient and also effective mutation analysis for large software systems, thus preventing this technique from being widely used in industry. To address these challenges, we suggest a new approach to mutant generation, called conditional mutation, that minimizes the generation time and enables efficient mutation analysis. The algorithm for conditional mutation and the empirical evaluation of this approach are published in [Just et al., 2011a].

- State of the art mutation analysis tools and frameworks are designed for and often limited to a certain testing scenario, yet they are mostly efficient for their purpose. To provide a flexible and easy-to-use mutation tool, we separate the mutant generation phase from the analysis part. While aiming at a fast, flexible, and lightweight mutator for Java programs that is applicable in any Java-based environment, we have implemented conditional mutation into the Java Standard Edition compiler. Design and implementation details are published in [Just et al., 2011b].

- Mutation analysis is computationally expensive, primarily due to the substantial number of generated mutants that have to be analyzed. However, when applying the mutation operators with their original definitions, the resulting set of mutants exhibits redundancies that are harmful to both efficiency and effectiveness. Considering the mutation operator that replaces conditional expressions, we define a non-redundant version of this operator. The theoretical results in conjunction with an empirical evaluation are published in [Just et al., 2012a,c].

- Assessing existing evolved and potentially long-running test suites is essential when aiming at a wider acceptance of mutation analysis in industry. Therefore, we analyze several real world software systems to identify certain characteristics of existing test suites and to ultimately suggest an optimized workflow that significantly reduces the runtime of mutation analysis. The results of the analyses, the workflow, and its empirical evaluation are published in [Just et al., 2012b].

- Partial oracles are a suitable approach to alleviate the oracle problem but their quality needs to be assessed due to their characteristics of potentially producing false-negative results. Moreover, partial oracles are generally not equally suitable for different testing phases such as unit and integration testing. They also manifest a considerable discrepancy in effectiveness for different software units. Nevertheless, their effectiveness can be improved by adequately combining several partial oracles. Based on a bipartite approach, which employs mutation analysis, these findings are published in [Just and Schweiggert, 2010] and [Just and Schweiggert, 2011].

The contents of the aforementioned publications have been revised, extended, and partially rewritten for this thesis.

1.3 Outline

The rest of this thesis is structured as follows: First, Chapter 2 provides an introduction to software testing by focusing on the challenges associated with automating software tests. This chapter also introduces techniques for test data generation and outlines some well established criteria for assessing the adequacy of (generated) tests. Since this thesis makes contributions to two areas of software testing, the subsequent chapters are divided into two parts, where Part II deals with efficient and effective mutation analysis and Part III examines the suitability of partial oracles for unit and integration testing.

Within the mutation analysis part, Chapter 3 discusses the necessary background of mutation analysis and outlines its limitations and research challenges. Then, Chapter 4 presents and evaluates a new approach to mutant generation that minimizes the necessary runtime to generate all mutants and enables efficient mutation analysis. Based on this approach, Chapter 5 describes the design and implementation of the corresponding compiler-integrated mutation tool. Furthermore, this chapter presents a domain specific language for mutation analysis and also describes the implementation and integration of a compiler for this domain specific language. Thereafter, the following chapters describe and evaluate further research approaches that are enabled by the developed tool, where Chapter 6 focuses on redundancies of mutation operators and presents a subsumption hierarchy and a non-redundant version of the operator that mutates conditional expressions. Finally, Chapter 7 suggests and empirically evaluates an optimized workflow for mutation analysis that exploits redundancies and runtime differences in existing test suites, which are to be assessed with mutation analysis.

With regard to Part III, which relies on mutation analysis to assess the effectiveness of partial oracles, Chapter 8 provides an introduction to the oracle problem and partial oracles. By focusing on metamorphic relations, a certain type of partial oracles, Chapter 9 thereafter investigates the adequacy of such partial oracles for unit and integration testing. This chapter also presents possibilities for efficiency and effectiveness improvements for the analyzed partial oracles.

By summarizing the results of all chapters in Part II as well as Part III, Chapter 10 finally concludes this thesis and discusses possible extensions and areas for future work.

Chapter 2

Introduction to Software Testing

Software testing is an integral part of the software development process to ensure that the developed software meets certain quality requirements. According to Myers et al. [2011] *"Testing is the process of executing a program with the intent of finding errors."* This definition of testing implies that a test is successful if it reveals an error. Conversely, the quality of a given program cannot be related to the fault-finding capabilities of a test, if this test does not detect any faults but is not assessed, based on a certain metric.

2.1 Software Testing Levels

Considering the different levels of abstraction within the software development process, the software testing process can also be divided into several levels, as shown in Figure 2.1.

Unit Testing

Unit testing denotes testing at the lowest level where the individual software units or modules are tested independently. Such a software unit may be a particular method or even an entire class but it is crucial to state that unit testing does not consider any dependencies or interactions between several units. Aiming at verifying the functionality of the software units, usually a test driver is necessary to emulate the environment in which the software unit is executed.

Integration Testing

At the integration testing level, the independently tested software units are integrated into larger subsystems and there are different strategies on how to best integrate the individual software units into subsystems (cf. [Spillner et al., 2007]). Regardless of which strategy is used, integration testing generally aims at checking whether the software units exhibit the expected behavior when integrated within a subsystem. That is, integration testing

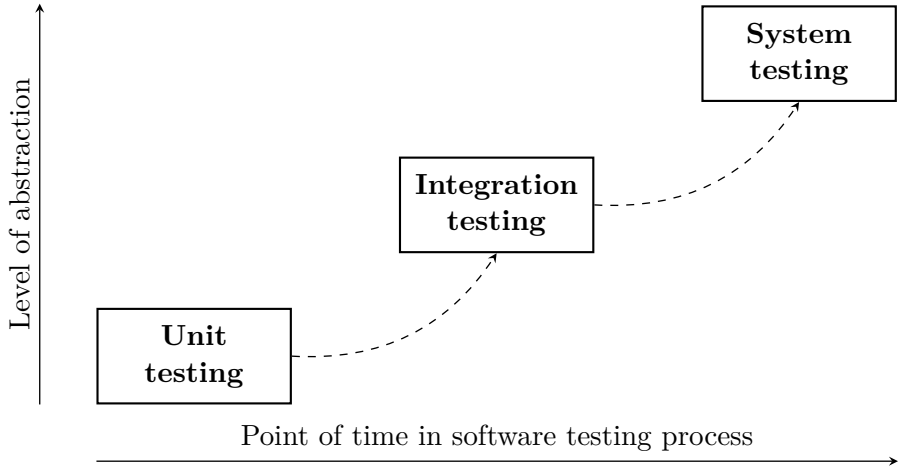


Figure 2.1: Different levels of software testing.

verifies the interfaces and the interactions between the individual software units. At this testing level, a test driver is still necessary to emulate the missing components.

System Testing

Testing a software system in which all units and subsystems are fully integrated is referred to as system testing. Since a software system that is verified at this testing level is fully integrated, a test driver is not necessary to execute it. System testing is, for instance, applied to verify the input and output characteristics of the system based on a given specification.

This thesis focuses on the lower testing levels, namely unit and integration testing. Even though the techniques and results presented in this thesis may also be transferable to a higher testing level, this matter is not investigated and discussed. It is also important to note that other higher testing levels, such as acceptance or robustness testing, exist that investigate the fully integrated software system but have a different view on it.

2.2 Automated Software Testing

Since software systems are constantly growing in size and complexity, manual testing becomes time consuming, sometimes even prohibitively so. Therefore, software tests should be automated, not only for efficiency reasons but also for reproducibility. Figure 2.2 illustrates a simplified testing process in which the system under test is executed with a certain input and the corresponding output is verified by means of an output-based oracle.

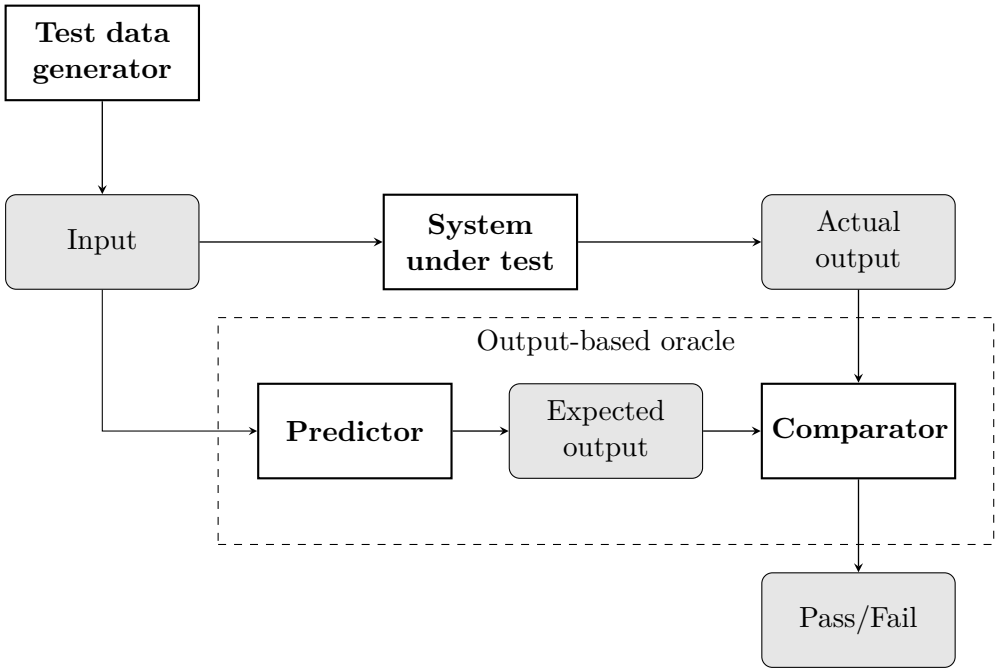


Figure 2.2: Overview of a simplified test process with an output-based test oracle.

Throughout this thesis, we use the term *testing strategy* to denote the union of the two parts necessary to test a software system, namely a model for generating test data and a test oracle to verify the corresponding output. It is important to state that we use the term *output* as synonym for every observable state, behavior, or outcome of the system under test. This might be, but is not limited to, the actual, observable outcome of the investigated program. Furthermore, this definition of testing strategy also applies to manual testing, where the test data may be predefined or hand-crafted and the human tester plays the role of the test oracle. Considering the following four individual sub-processes of the diagram in Figure 2.2:

- Generating input values
- Executing the system under test
- Predicting the expected output
- Comparing the actual to the expected output

In order to automate the entire software test process all of these four sub-processes have to be automated. With regard to the execution of the system under test, several testing frameworks exist that can automate this step. JUnit [2012] is for instance the most popular framework for testing applications written in the Java programming language. Furthermore, a lot of automated test data generation models exist that for instance rely on random data generation or on a directed search guided by a certain test adequacy

criterion. An introduction to test adequacy criteria as well as test data generation models is provided by the subsequent Sections 2.3 and 2.4.

Concerning the last two parts that form the test oracle, namely the determination of the expected output and the comparison of the actual and expected one, the comparator can also be easily automated if both the actual and the expected output are available. However, the prediction of the correct output often results in the oracle problem if such an predictor is not available or cannot be generated with a reasonable effort. Intuitively, such a predictor can be generated by re-implementing the system under test. Yet, this effort is considered to be disproportionate and testing the implemented predictor would again result in the oracle problem. Concerning the oracle problem, various standard solutions (cf. [Binder, 1999]) exist which are, however, only employable in rare situations. On the other hand a promising class of oracles, the *partial oracles* [Weyuker, 1982], are considered to be easily automatable, more often applicable and should therefore be used for automating software tests [Bertolino, 2007].

Since Part III addresses testing with partial oracles, Chapter 8 provides a more detailed introduction to the oracle problem and several types of partial oracles.

2.3 Test Adequacy Criteria

The famous corollary of Dijkstra [1970]: *"Program testing can be used to show the presence of bugs, but never to show their absence!"* clearly indicates that a good software quality cannot be deduced from software tests that do not find any faults. The applied testing technique could simply be inadequate with regard to the assessed software system. As a consequence, there is clearly a need for methods and metrics to also asses the quality of software tests. This section provides a basic introduction to test adequacy criteria, focusing on code coverage and fault injection. It is important to note that this section does not aim at providing a comprehensive survey of test adequacy criteria but rather gives an intuitive introduction to the different categories. A more detailed introduction to test coverage and adequacy is, for instance, provided by the survey of Zhu et al. [1997] and the introductory book of Ammann and Offutt [2008].

Generally, a coverage criterion defines rules or requirements that have to be fulfilled by a test suite in order to be adequate with regard to this criterion. Consequently, the corresponding metric is the level of coverage l_c that can be generally defined as follows:

Definition 2.1 *Level of coverage* l_c

$$l_c = \frac{\textit{Number of satisfied requirements}}{\textit{Number of all (satisfiable) requirements}}$$

It is important to note that some requirements are unsatisfiable and therefore have to be excluded to enable an investigated test suite to achieve a level of coverage of 100%.

```
1 public static int max(int a, int b){
2     int max = a;
3
4     if (b > max){
5         max = b;
6     }
7
8     return max;
9 }
```

Listing 2.1: Example method that determines the maximum of two integer values.

2.3.1 Control Flow Coverage

A control flow graph is a directed graph with two kinds of nodes and edges that represent a flow between these nodes. With regard to the nodes, the graph contains the individual statements of a program and decisions, that is every conditional statement in the program, which result in two branches. Having only these two kinds of nodes is sufficient since loops are represented by decisions with one branch being a back edge to the node representing the loop header. Additionally, a sequence of statements that does not contain any decision is referred to as basic block. Now, based on such a control flow graph, several coverage criteria can be defined.

Two well-know, yet rather simple, control flow coverage criteria are statement and branch coverage. Statement coverage only requires that every statement has to be executed at least once. Branch coverage additionally requires that every branch has to be covered. This is of particular importance for empty branches that occur for instance in if statements without a corresponding else part. Hence, branch coverage is the stronger criterion and subsumes statement coverage. Statement and branch coverage only lead to equivalent results for programs that do not contain any empty branches – this is, however, rarely the case for non-trivial programs.

Concerning the depicted control flow graph of Figure 2.3, statement coverage can easily be achieved when the condition, that is `b>max`, is fulfilled due to the fact that the else part of the condition does not contain any statements. In contrast, branch coverage requires the execution of both paths, even though there is no additional statement on the path that is executed when the condition is not fulfilled.

With regard to decisions that are composed of several sub-expressions, the modified condition/decision coverage (mc/dc), for instance required by the Radio Technical Commission for Aeronautics (RTCA) [2011] for critical systems, is a much stronger criterion than statement and branch coverage. It requires that the entire condition as well as every sub-expression evaluates to `true` and `false`. Moreover, it requires that the impact of every sub-expression on the outcome of the decision is tested independently. That is, the result

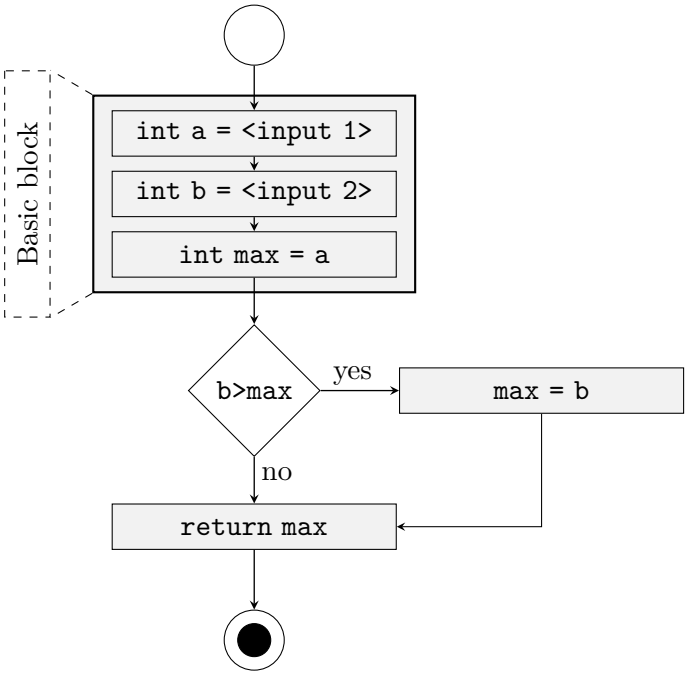


Figure 2.3: The control flow graph for the max method of Listing 2.1.

of the decision should change when the boolean value of a sub-expression changes. This implies that only combinations that fulfill this requirement have to be tested to avoid the cost of combinatorial testing.

2.3.2 Data Flow Coverage

A data flow graph provides a different view on code abstraction and describes the correlation of variable definitions and assignments with their use. Figure 2.4 illustrates a data flow graph, again for the method shown in Listing 2.1. The following definitions are usually used to describe a certain coverage criterion on such a data flow graph:

Definition

A definition (def) of a variable is either a declaration in conjunction with an initialization or an assignment, where the variable is used as left hand side.

Use

A use occurs whenever a variable is accessed, meaning that its value is read and used. Concerning the actual use of the variables value, two kinds can be distinguished:

c-use: A use of the value within a computation.

p-use: A use of the value within a predicate of a logical expression.

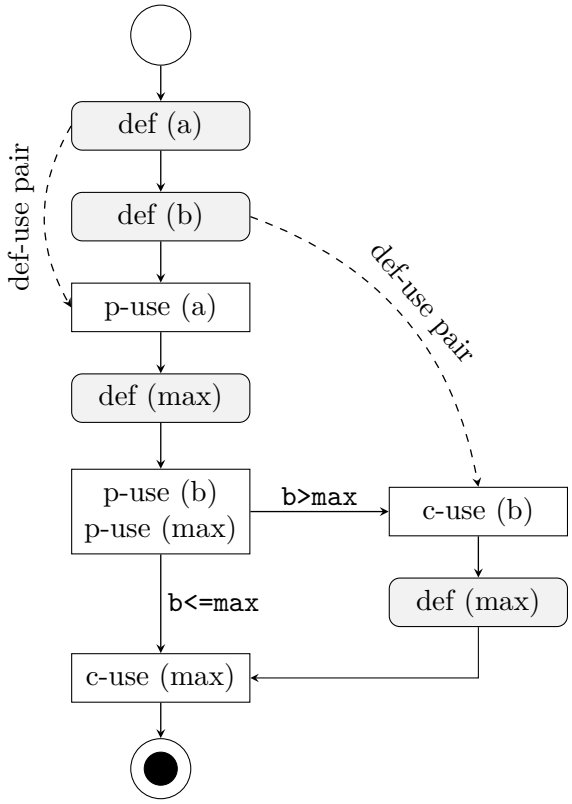


Figure 2.4: The data flow graph for the max method of Listing 2.1.

Definition-clear path

A definition-clear path within a data flow graph is a path from a def to a use node for a given variable that does not contain other def nodes of the same variable.

Def-use pair

A def-use pair for a certain variable consists of a def and a use node of the same variable connected by a definition-clear path. It is crucial to state that a def-use pair may have several different definition-clear paths. A certain definition-clear path for a given def-use pair is also referred to as *def-use path*.

Concerning these definitions, several criteria can be defined such as all-defs that requires the coverage of all definitions with an arbitrary def-use path. In contrast, all-uses requires the coverage of all def-use pairs with one definition-clear path. Apparently, the latter criterion subsumes the first one and is generally much harder to fulfill. Figure 2.4 shows the corresponding data flow graph of the simple max method.

2.3.3 Fault-based Criteria

All code coverage criteria, regardless of which category, manifest the same weakness with regard to the assessment of tests. A fulfilled code coverage criterion does not imply that the investigated test case or test suite is also effective in detecting faults. In fact, a test case that reaches a 100% coverage may not detect any fault if the corresponding output is not verified by an appropriate oracle. Hence, coverage criteria can only assess one part of a testing strategy, namely the test data generation model.

To overcome this problem, fault-based approaches assess the quality of a given test set with regard to its ability to reveal seeded faults. Hence, these approaches also assess the second part of a testing strategy, namely the test oracle, since such an oracle is necessary to eventually detect a fault. Usually, a fault-based technique is applied in order to increase the confidence that the tested software does not contain a certain type or class of faults (cf. [Morell, 1990]). Hence, the power of fault-based techniques clearly depends on whether the seeded faults are subtle and representative for the type of faults intended to eliminate.

Error seeding is a classical fault-based approach, where an experienced software engineer seeds faults led by the intuition where a programmer could have made a mistake (cf. [Mills, 1972]). However, the faults obtained from this approach are hardly reproducible and moreover they are based on the programmers opinion rather than on formal definitions. Besides these generalizability issues, error seeding for larger software systems is either almost prohibitively expensive or the number of the obtained faults is to small to adequately assess a test suite for the corresponding application.

Mutation testing, which systematically injects artificial faults and also belongs to this category of test adequacy criteria, tries to overcome the challenges of manual error seeding.

Since this thesis makes several contributions to the field of mutation analysis, this technique is introduced more detailed in Chapter 3.

2.4 Test Data Generation

With regard to automated test data generation, several approaches have been proposed. This section summarizes some of these techniques and discusses their advantages and drawbacks. Most systematic test data generation techniques aim at generating tests that reach a certain level of coverage for a given criterion (e.g., statement or branch coverage). Additionally, mutation is also a common criterion for test data generation – the goal for these techniques is to generate test sets that reveal as much mutants as possible.

Random Generation

Intuitively, random test data generation is an easy and efficient way to obtain almost arbitrary input values or more complex test data. However, a pure randomized generation technique cannot guarantee that a certain level of coverage or particular paths are reached. Nevertheless, due to the low computational costs, several randomized iterations can be

performed to achieve satisfying results. Since random testing is furthermore an unbiased technique and offers very good scalability, even for large software systems, it is often used to generate test data.

Randomly generated tests, however, may produce invalid or useless test cases, for instance sequences of method calls that are interrupted during their executing due to a thrown exception. Moreover, the probability of reaching a path that is rarely executed due to its path constraints is low. As a consequence, the test suite size may become relatively large to achieve a certain level of coverage. To overcome this problem Pacheco et al. [2007] suggested a feedback-directed test generation, which builds valid sequences by reusing already existing and verified results of prior method calls in the same sequence. Yet, checking whether a method call or generally a sequence extension is valid, leads to considerable effort. Nevertheless, the obtained test suites clearly outperform randomly generated tests with respect to effectiveness.

Constraint-based Generation

Another popular approach is the generation of test data by solving path constraints utilizing symbolic execution (cf. [King, 1976]). Instead of real input values symbolic values are used and outputs are represented by a function of these symbolic values. Furthermore, branches or paths within the code are represented by path conditions, that is the accumulation of all constraints which have to be fulfilled to execute this path. Hence, a real input value that eventually reaches a certain statement or branch in the code is determined by solving the assembled path constraints. This technique is for example implemented in *Java PathFinder* [Visser et al., 2004] or *Pex* [Tillmann and De Halleux, 2008]. Besides these implementations, DeMillo and Offutt [1991] suggested constraint-based test data generation using mutation instead of structural code coverage. Even though constraint-based test data generation achieves generally a good level of coverage, it is compared with other techniques rather costly in terms of memory consumption and runtime.

Search-based Generation

In order to avoid the drawbacks of pure random and constraint-based test data generation, metaheuristic, that is search-based, approaches are another class of test data generation techniques. McMinn [2004] surveyed and compared well-known algorithms and existing approaches that belong to this class. While most approaches focus on one test goal at a time and therefore generate quite large test sequences, Fraser and Arcuri [2012b] proposed a technique that focuses on the entire test suite to achieve a high level of coverage and a small test size at the same time.

Similar to random and constrained-based generation, mutation can also be used as fitness function to search for adequate test cases. Fraser and Zeller [2012] investigated mutation's adequacy for this purpose on 10 open-source libraries and found that compared with the manually written, existing test suites most of the generated test suites manifest shorter sequences. Moreover, the generated test suites clearly outperformed the existing test suites in terms of the detection rate of seeded faults. This approach is among others implemented in the *EvoSuite* tool [Fraser and Arcuri, 2011].

Even though this thesis does not address research questions in test data generation, this section clearly indicates, that mutation analysis plays an important role in current research on test data generation. Hence, it is crucial to focus on efficient and also effective mutation analysis in order to be able to employ mutation as criterion for test data generation.

Part II

Efficient and Effective Mutation Analysis

Chapter 3

Background and Limitations of Mutation Analysis

This chapter is based on [Just et al., 2011a,b].

This chapter deals with the fundamentals of mutation analysis and provides the necessary basic knowledge of this technique. After describing the history, background, and fundamental hypotheses of mutation analysis, this chapter also outlines several limitations. Additionally, it discusses related work and the state of the art, both necessary to later distinguish the contributions of this thesis presented in the subsequent chapters.

3.1 Mutation Analysis Background

Originally introduced in [Budd, 1980; DeMillo et al., 1978], mutation analysis is a well known technique for assessing the quality of a given test set or testing strategy. In this fault-based approach, artificial faults are systematically seeded into a System Under Test (SUT) and the corresponding test sets or testing strategies are examined with respect to their ability to reveal these faults. It is crucial to state that the faults are injected methodically, contrary to the classical approach where error seeding is led by the intuition of experienced engineers (cf. [Mills, 1972]). Thus, mutation analysis can be regarded as an unbiased technique and leads, furthermore, to reproducible results.

The way of applying mutation analysis is specified by *mutation operators* and the resulting faulty versions of the SUT are referred to as *mutants*. Examples of mutation operators are the replacement of variables by constants or the swapping of operators in conditions (cf. [King and Offutt, 1991; Ma et al., 2002, 2006]). Generally, mutation analysis is programming language independent but the mutation operators depend on the chosen language since they have to cover the corresponding syntax and semantics. If a test case reveals a fault, the corresponding mutant is said to be *killed* and consequently, an undetected mutant is referred to as *live mutant*. Hence, relating the number of all the

killed mutants to the number of generated mutants is intuitively an appropriate way to measure the quality of the applied test set.

However, a mutant cannot be killed in certain circumstances. In fact, when there exists no test case that can detect the mutant, it is said to be *equivalent*. It is important to note that this equivalence applies to the semantics since a mutant represents a valid syntactic change of the original program. Mutants manifest a semantic equivalence to the original program if the mutation is either not reached and executed or it has no effect on any observable state. Trivial examples for such equivalent mutants are the mutation of unused or dead code fragments and the manipulation of a variable which is no longer used afterwards. Let p be the original program with its corresponding set of input values and parameters π, that is π denotes the union of all input values and parameters necessary to execute p. For a given input domain Π, a mutant m is equivalent to p, written as $m \equiv p$, if and only if it produces the same output as p for all possible input tuples:

$$m \equiv p \Leftrightarrow m(\pi) = p(\pi), \forall \pi \in \Pi \tag{3.1}$$

With regard to the aforementioned intuitive quality metric, the ratio of killed to generated mutants cannot yield 100% if the set of mutants contains equivalent ones. Thus, it is necessary to investigate live mutants in order to remove those which are equivalent. Even though this task is mostly done manually, some sound and also heuristic approaches exist to identify certain types of equivalent mutants automatically [Offutt and Pan, 1997; Schuler and Zeller, 2010]. Disassociating the equivalent mutants from the complete set leads to a more precise metric, and thus provides an improved assessment of the effectiveness of the analyzed test set or testing strategy. This measure is referred to as *mutation score S* and it is defined as follows:

Definition 3.1 *Mutation Score S*

$$S = \frac{M_k \text{ (Number of killed mutants)}}{M_t \text{ (Number of non-equivalent mutants)}}$$

It is important to note that mutation analysis is not feasible without appropriate tool support. Mutation tools have been developed for many common programming languages, such as Fortran, C, C#, or Java. While each system has its own strengths and weaknesses, most of them are designed as a comprehensive framework for certain analyses.

3.2 Hypotheses of Mutation Analysis

The acceptance of mutation analysis being a good adequacy criteria is based on two fundamental hypotheses, namely the competent programmer hypothesis and the coupling effect. Both hypotheses were initially postulated, investigated, and discussed by DeMillo et al. [1978], Acree et al. [1979], Budd [1980], and Budd et al. [1980].

3.2.1 Competent Programmer Hypothesis

The main assumption of the competent programmer hypothesis is that programmers write programs that are almost perfect. This means that the developed program is close to being correct, and thus within the neighborhood of the correct version. As a consequence, it is sufficient to inject small defects to simulate or represent real faults. Moreover, the determined adequacy of a test case or test suite that is based on artificial faults is a good approximation of the test's adequacy for real faults, provided that the competent programmer hypothesis holds.

3.2.2 The Coupling Effect

The definition of coupling, either informal or formal, is inconsistently used in the literature (e.g., [Offutt, 1989, 1992] and [Morell, 1984, 1990]). However, we refer to the term coupling effect as the capability of a test case that detects simple faults also to reveal more complex faults, which are composed of the simple ones. Transferred to mutation analysis, the coupling effect implies that tests that kill first order mutants are likewise effective in killing second or higher order mutants. Assuming that the coupling effect holds, it is sufficient to use only first order mutants to assess the quality of a test set.

Given the strong evidence that the coupling effect actually holds [Lipton and Sayward, 1978], Offutt [1989, 1992] investigated the coupling effect for first, second, and third order mutations. The results support the validity of the coupling effect but also showed the existence of decoupled higher order mutants, even though the number was almost negligible. Moreover, Jia and Harman [2009] showed lately that higher order mutations exist that are harder to detect than the simple defects, that is the first order mutation, of which they were composed. Nevertheless, the hypothesis of the coupling effect in mutation testing is still widely accepted due to the significant number of coupled first order mutants.

3.3 Limitations of Mutation Analysis

Even though many prior studies have shown that mutation analysis is a powerful metric [Jia and Harman, 2011], it may be still prohibitively time consuming and computationally expensive in comparison to other methods, such as those that rely on code coverage criteria. So, mutation analysis is not only powerful but also associated with high costs in terms of manual and computational effort. As a consequence, some limitations exist that preclude an application of mutation analysis for large software systems. We therefore discuss several limitations here since they are addressed within the following chapters.

3.3.1 Scalability Issues

Based on the underlying hypotheses, mutation analysis usually employs first-order mutants, meaning that a mutant contains only one mutation at a time. Thus, the mutation

of large or complex applications results in a substantial number of mutants when all possible mutation operators are applied. In consideration of this large number of mutants, there are two challenges concerning the scalability, namely the compilation time and the runtime necessary to execute all the mutants. The limiting factor depends on the chosen approach, that is interpretative or non-interpretative mutation analysis.

For instance, the compilation time can be avoided on interpretative systems. However, the runtime of the mutation analysis is clearly longer on those systems. Obviously, the complete runtime t_t includes the compilation time and the execution time of the individual mutants. With regard to a certain number of mutants M and a given test set of size N, the total runtime, made up of the compilation time per run t_c and the mean execution time for one test case t_e, can be described with the following equation:

$$t_t = \underbrace{(M+1) \cdot t_c}_{compilation} + \underbrace{(M+1) \cdot N \cdot t_e}_{execution} \tag{3.2}$$

It is important to note that both the compilation and the execution have to be executed $M+1$ times since, besides all mutants, the original version is also compiled and executed. Considering the two individual parts of Equation 3.2, the compilation is usually faster than the execution. Nevertheless, the compiler constitutes a significant proportion of the total runtime and was regarded as a bottleneck in the late 1980s, thus explaining why the first mutation systems were designed as interpretative ones [King and Offutt, 1991]. To provide some real numbers, we consider two examples. On the one hand we use the triangle application, a very small yet widely-used example program that classifies triangles [Myers, 1979] together with a manually created test suite. On the other hand we use the apache commons-lang library, a medium-sized system, which includes a comprehensive test suite, used in several empirical evaluations in this thesis. It has to be pointed out that we do not aim at assessing the corresponding test suites but rather illustrate the ratios between compile and execution time.

TRIANGLE: $t_t = 36.5 = (21.4 + 15.1)$ seconds

- Mean compilation time t_c: 170 milliseconds
- Number of mutants M: 125
- Number of test cases N: 20
- Mean runtime per test t_e: 6 milliseconds

COMMONS-LANG: $t_t = 134.6 = (3.5 + 131.1)$ hours

- Mean compilation time t_c: 380 milliseconds
- Number of mutants M: 33,065
- Number of test cases N: 2,039
- Mean runtime per test t_e: 7 milliseconds

While the compilation and execution times are similar for the triangle application, they differ by two orders of magnitude for the commons-lang library. Generally, the execution time is becoming the crucial factor if the system and the corresponding test suite are growing. Nevertheless, the compilation process constitutes a significant proportion of the total runtime.

3.3.2 Infinite Loops

Another challenge in mutation testing is the prolonged or even infinite runtime of mutants. When, for instance, mutating loop conditions as shown in Listing 3.1 and 3.2, some mutations lead to infinite loops. Generally, such non-terminating mutants have to be detected or interrupted in order to prevent the mutation analysis process from getting stuck. It is, however, undecidable in general whether or not a given mutation leads to an infinite loop, and hence does not terminate. This problem is getting even worse if the mutant is only non-terminating for certain inputs. As a consequence heuristic methods such as timeouts or loop counters are applied to alleviate this problem.

3.3.3 Equivalent and Redundant Mutants

As previously mentioned, some mutants cannot be detected due to a semantic equivalence to the original program. These equivalent mutants are harmful for two reasons. First, an equivalent mutant clearly prolongs mutation testing since it has to be executed for every test in the entire test suite – simply to draw the conclusion that this mutant cannot be killed by the test suite. Also, unless detected and removed, equivalent mutants misrepresent the mutation score, and thus underestimate the quality of the analyzed test suite. This point is crucial if the mutation score is used as an end-of-test criterion due to the fact that a level of coverage of 100% cannot be achieved if the set of mutants contains equivalent ones.

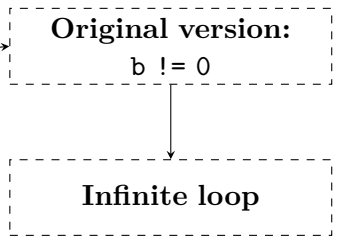
Besides the equivalent mutant problem, redundant mutants are also harmful. A mutant is said to be redundant if the result of its execution can be predicted based on the execution of other mutants. Mutants are, for instance, redundant if they are semantically equivalent to each other or if they are subsumed by other mutations. As a consequence, these redundant mutants overestimate the mutation score since they are easy to kill, provided that the subsuming mutant or the counterpart is detected. Moreover, a redundant mutant also incurs a runtime overhead due to its unnecessary execution – the result can be determined a priori. Listing 3.3 and 3.4 illustrate the redundancy between two mutants that are derived from two different mutation operators but manifest a semantic equivalence to each other.

Overall, it is important to detect and remove equivalent and redundant mutants for efficiency reasons and, even more important, to provide an accurate metric for the assessment of a test suite's quality.

```
1  public static int  gcd(int a, int b){
2
3      while (b != 0) {
4          if (a > b) {
5              a = a - b;
6          } else {
7              b = b - a;
8          }
9      }
10
11     return a;
12 }
```

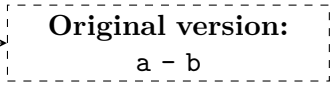
Listing 3.1: Original version of an iterative implementation that determines the greatest common divisor of two integer numbers.

```
1  public static int  gcd(int a, int b){
2
3      while (b >= 0) {
4          if (a > b) {
5              a = a - b;
6          } else {
7              b = b - a;
8          }
9      }
10
11     return a;
12 }
```



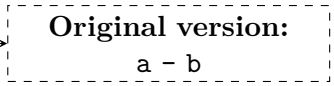
Listing 3.2: Faulty version of the greatest common divisor implementation of Listing 3.1 with a mutated loop condition leading to an infinite loop.

```
1  public static int  gcd(int a, int b){
2
3      while (b != 0) {
4          if (a > b) {
5              a = a + b;
6          } else {
7              b = b - a;
8          }
9      }
10
11     return a;
12 }
```



Listing 3.3: Mutated version of the greatest common divisor implementation of Listing 3.1 with a replaced binary arithmetic operator.

```
1  public static int  gcd(int a, int b){
2
3      while (b != 0) {
4          if (a > b) {
5              a = a - -b;
6          } else {
7              b = b - a;
8          }
9      }
10
11     return a;
12 }
```



Listing 3.4: Mutated version of the greatest common divisor implementation of Listing 3.1 with an inserted unary arithmetic operator.

3.4 State of the Art

As already mentioned in Section 3.3, applying all suitable mutation operators to large or complex implementations leads to a huge number of mutants, thus having a considerable impact on the runtime of mutation analysis. Considering the computational costs of mutation analysis, several approaches have been discussed in the literature (cf. [Jia and Harman, 2011]). As noted by Offutt and Untch [2000], all techniques and strategies aiming at reducing these costs, and thus making mutation more efficient, can be related to one of the three categories *do fewer*, *do smarter*, and *do faster*. In addition to this categorization, *higher order mutation* is another technique that primarily aims at improving the effectiveness of mutation analysis. Since it involves the reduction of mutants, higher order mutation also has an effect on the efficiency.

Do Fewer

Do fewer approaches are sampling and selective techniques that reduce the number of mutants either by decreasing the number of operators or by selecting just a subset of the generated mutants. Offutt et al. [1996] investigated the effectiveness of mutation operators and determined that the mutants generated with a smaller subset of sufficient operators are almost as hard to kill as the complete set of mutants. Thus, the reduced set of mutation operators can be applied much more efficiently without a major loss of information.

Do Smarter

Do smarter techniques exploit for instance the possibilities of running mutation analysis in a distributed environment [Choi et al., 1989]. Since every mutant is generated independently, the computation can be parallelized. Another do smarter approach is weak mutation testing [Howden, 1982] where a mutant is said to be killed if its internal state, after executing the mutated code, differs from the original program. Hence, only necessary conditions can be verified by applying weak mutation. Nevertheless, the minor loss of information is proportionate to the considerable decrease in time overhead.

Do Faster

Do faster approaches generally aim at improving the runtime of mutation analysis without using reduction or parallelization. Considering the conventional way of applying mutation analysis, every mutant is a copy of the original program apart from a small syntactical change. According to that fact, compiling every mutant as an independent source file is a substantial time overhead during compilation. In order to alleviate the costs of compiling every mutant, DeMillo et al. [1991] proposed a compiler-integrated technique. They modified a compiler to create patches on the machine code level. Thus, the original program was compiled just once and the mutants were created by applying corresponding patches to the compiled original program. However, the effort to implement or adapt the necessary compiler is significant.

With respect to the Java programming language, which uses an intermediate language, bytecode transformation is a similar technique which directly transforms compiled code. While these modifications are usually faster than source code transformations since they obviate additional compilation steps, there are still some drawbacks to this approach. First of all, the bytecode has been simplified or even optimized during the compilation process. Therefore, errors might be injected which could never have been introduced by a programmer at the source code level. Additionally, all semantic information collected during compilation phases, such as building and attributing the abstract syntax tree, has to be gathered redundantly by parsing the bytecode again.

The mutant schemata approach is another do faster technique that encodes all mutations within generic methods and replaces the original instructions, which shall be mutated, with a call of the corresponding generic method. Accordingly, the necessary time to compile all mutants is reduced. As described by Untch et al. [1993], *"A mutant schema has two components, a metamutant and a metaprocedure set"*. The effective mutation of the metamutant is determined at runtime within the generic methods by means of appropriate flags. An example for mutant schemata is the replacement of the built-in arithmetic operators by a generic method `AOP`:

```
int a = AOP(b, c, '+'); ⟵↦ int a = b + c;
```

The generic method `AOP` can now perform a different arithmetic operation at runtime which leads to a mutation of the original statement.

Generally, the creation of metamutants can be regarded as a template-based technique. However, the introduction of several indirections, when implemented with method calls [Untch et al., 1993], implies an additional overhead which may have a significant impact on the runtime of the compiled program.

Higher order mutation

Higher order mutation is, in addition to these three categories, another approach to generate fewer, but more subtle mutants. Generating mutants by means of the combination of two simple mutants, called first order mutants, is referred to as second order mutation. Accordingly, higher order mutation denotes generally the combination of two or more first order mutants. The computational costs for second and higher order mutation are considerably higher because of the huge number of possible combinations. Nevertheless, recent work has shown that second and higher order mutants exist that are harder to kill than the first order mutants of which they have been generated [Jia and Harman, 2009]. Hence, applying these higher order mutants would provide a better assessment for mutation analysis. The problem, however, is to identify appropriate decoupled higher order mutants in an efficient way. Search-based approaches seem to be a feasible solution for this problem [Jia and Harman, 2009].

3.5 Summary

This chapter has provided the necessary basic knowledge about mutation analysis and has outlined the fundamental hypotheses on which this technique is based on. Moreover, it has discussed limitations and challenges of mutation analysis addressed in this thesis, which can be summarized as follows:

- Scalability issues: A significant number of mutants can be generated for large and complex systems, hence leading to a computational effort and runtime that is almost prohibitive.
- Infinite loops: Mutants that result in infinite loops have to be detected in order to ensure that the mutation analysis process will eventually terminate.
- Redundant and equivalent mutants: Mutants that are subsumed or that cannot be detected are harmful to both efficiency and effectiveness.

Chapter 4

Conditional Mutation

The content of this chapter has been published in [Just et al., 2011a]

This chapter describes and evaluates a versatile technique, called *conditional mutation*, which increases the efficiency of mutation analysis. Conditional mutation is a compiler-integrated approach that transforms the abstract syntax tree, significantly reduces the time overhead for generating the mutants, and enables efficient mutation analysis.

4.1 Introduction

Mutation analysis is among other approaches suitable for assessing the quality of test suites [Andrews et al., 2005]. However, applying mutation analysis to large software systems is problematic since it is a time-consuming technique. Addressing this challenge, this chapter describes and evaluates conditional mutation, an approach to increase the efficiency of mutation analysis.

Conditional mutation is based on transforming the abstract syntax tree (AST) to provide all mutants in conjunction with the original program within the resulting assembled code. The name is derived from the conditional statements and expressions which are inserted to encapsulate the mutants. In comparison to prior approaches, as for instance [Untch et al., 1993; Ma et al., 2006; Schuler and Zeller, 2010], it is more general because it operates at the source code level on expressions as well as statements. Furthermore, conditional mutation can be integrated into the compiler as an additional AST transformation.

The evaluation of conditional mutation within this chapter is based on eight investigated programs with a total of 750,000 lines of code leading to almost 800,000 generated mutants. In order to obtain the corresponding performance results, conditional mutation has been integrated into the Java 6 Standard Edition compiler, thus ensuring that it is widely applicable in any Java-based environment and not limited to a certain testing tool or framework.

In consideration of this compiler-integrated approach, the runtime to generate and compile the mutants is reduced to a minimum. For instance, the total overhead for compiling 406,000 mutants for the largest investigated program, namely the *aspectj* compiler, is only 33% compared with the default compile time. This time overhead includes the cost of both mutant generation and compilation and is thus orders of magnitude less than compiling the mutants individually. In addition, the time to run the test suites, which are released with the investigated programs, is on average only 15% higher than normal testing time in the worst-case scenario. The worst-case scenario of testing the instrumented program is associated with executing all of the conditional statements and expressions that enable mutation analysis.

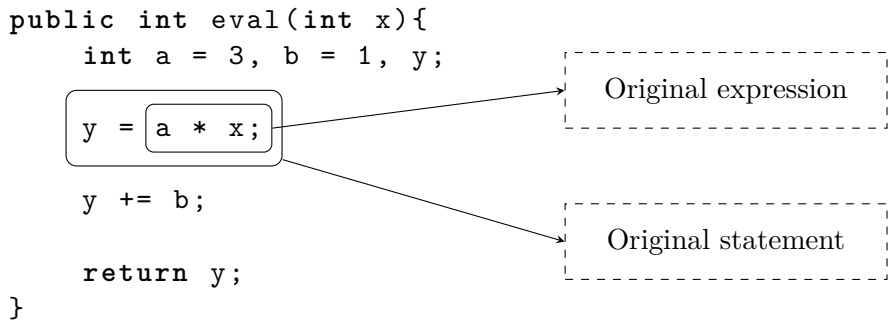
The remainder of this chapter is structured as follows: Section 4.2 motivates and presents the approach. Besides a description of the implemented algorithm, this section also outlines several optimizations, the support for higher order mutation, and implementation details. Thereafter, Section 4.3 evaluates the approach as well as the implementation by means of an empirical study with eight software systems. Section 4.4 then discusses related techniques by focusing on the mutant schemata approach and bytecode transformation and it also outlines general limitations. Finally, Section 4.5 summarizes this chapter.

4.2 Conditional Mutation

The conventional way of generating mutants, as for instance implemented in MuJava [2009], results in a set of source files with the convention that each file contains exactly one mutant. This approach to mutant creation incurs a high time overhead because it must repeatedly load and compile every mutant file. Moreover, the system under test has to be executed repeatedly with every mutant to determine the mutation score. Now, if every mutant is compiled to an individual class, every corresponding file has to be loaded to execute the mutated code.

The basic idea of conditional mutation is to accomplish all mutations in the corresponding source file and more precisely in the same entity. This means that all mutants are encoded in the same block or scope and within the same method and class as the original piece of code. Hence, a conditional mutant is a valid substitution of an arbitrary instruction based on conditional evaluations and it contains all mutations as well as the original instruction. Thus, every conditional mutant preserves the scope and visibility within the AST.

Regarding the example in Listing 4.1, we can distinguish between statements and expressions. Intuitively, every program instruction which is terminated, for example with a semicolon, is a statement. An expression is also an instruction but it represents a value which can be or has to be evaluated within a statement, depending on the corresponding language. For instance, `int a = 3` is a statement which can be used as a single instruction. In contrast, the expression `a * x` represents a value and cannot be used as a statement, in the Java programming language, by adding a terminating semicolon. So, an expression is always part of a surrounding statement with the exception of so-called expression statements. These are expressions like method calls, unary increment/decrement operators, or assignment operators which can be used either as an expression or a statement.



Listing 4.1: Method with statements and expressions.

The conditional mutation approach aims at retaining the basic structure of a program's AST. This means that unnecessary local variables, statements, or blocks must not be inserted. Thus, every expression or statement which is to be mutated has to be replaced with an appropriate expression or statement, respectively. Therefore, conditional statements or conditional expressions are inserted where the `THEN` part contains the mutant and the `ELSE` part the original code. The condition for these conditional expressions or statements may contain an arbitrary expression which determines when the mutant should be triggered. Regarding mutation analysis, where every mutant should be executed, the enumeration of all mutants and the insertion of a global mutant identifier (e.g., a global variable `M_NO`) is advisable. This variable makes it possible to dynamically choose the mutant to be executed. Thus, the expression of the condition is a comparison of the identifier with the mutant's number.

4.2.1 Tail-Recursive Algorithm

The proposed algorithm is applicable for both expressions and statements but it is explained based on expressions. In order to apply it to statements, *expr* has to be replaced by *stmt* and a conditional statement *CondStmt* has to be used instead of *CondExpr* within the recursive function (4.4).

The available mutation operators depend on the expression to be mutated. Thus, the union of all applicable operators can be defined as a set for a certain expression:

$$MOP(expr) = \{mop_1, \cdots, mop_n\}, n \in \mathbb{N} \tag{4.1}$$

Considering a binary arithmetic expression, examples for the mutation operators mop_i would be the replacement of the arithmetic operator or the permutation of the operands.

Next, the syntax tree is traversed and every expression for which at least one mutation operator exists will be replaced by an expression $expr'$:

$$expr' \longleftarrow expr, \forall expr : MOP(expr) \neq \emptyset \tag{4.2}$$

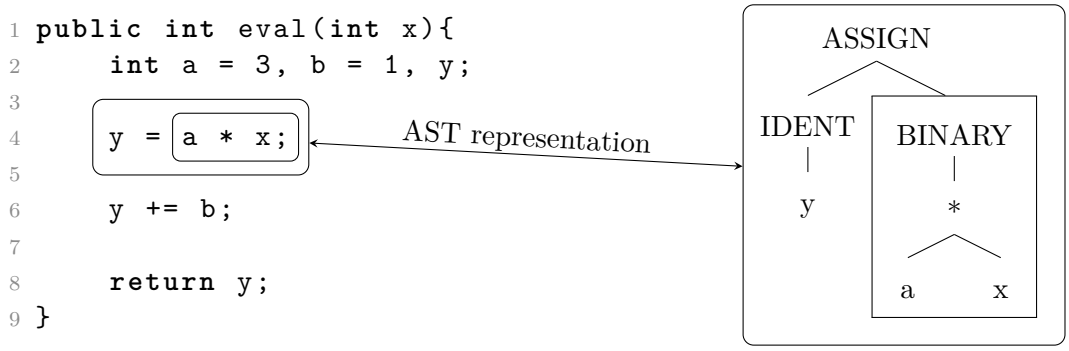


Figure 4.1: AST subnode of an assignment with a binary expression as right hand side.

In order to apply the first k mutation operators given by the set $MOP(expr)$, a recursive algorithm can be defined. In the base case the expression $expr$ is replaced by a conditional expression $CondExpr$ which contains the condition $cond_1$, the mutant, determined by the evaluation of $mop_1(expr)$, and the original expression $expr$. Any further mutation is encapsulated within a conditional expression which in turn contains the result of the previous mutation step.

$$expr' = mut_k, k \in \mathbb{N} \wedge k \leq n \tag{4.3}$$

$$mut_i = \begin{cases} CondExpr(cond_1, mop_1(expr), expr), & i = 1 \\ CondExpr(cond_i, mop_i(expr), mut_{i-1}), & i > 1 \end{cases} \tag{4.4}$$

Since function 4.4 is tail-recursive it can also be implemented as an iterative algorithm if the compiler of the corresponding programming language does not support tail-recursion elimination. By means of an appropriate ordering of the set $MOP(expr)$ in conjunction with the parameter k, sampling strategies or selective mutation can be applied. Exemplary results of using the algorithm are illustrated in Listing 4.2 and Figure 4.2. Reconsidering the assignment `y = a * x`, shown in Figure 4.1, the binary expression which shall be mutated is a subnode of the statement's AST. Hence, just this subnode is replaced in accordance with the conditional mutation approach. It has to be pointed out that a possible mutation operator $mop_i \in MOP(expr)$ for an expression $expr \in stmt$ must not occur in the set $MOP(stmt)$ of the surrounding statement. For instance, this means that the replacement shown in Listing 4.3 is invalid according to conditional mutation since this mutation can be applied at the expression level. This constraint is of particular importance for nested expressions, block statements, and loops because the complete outer expression or statement should not be duplicated.

Regarding the modified AST in Figure 4.2, the framed node, including its children `a` and `x`, is the original node of Figure 4.1. Therefore, any further transformations on child nodes, such as replacing `x` by a constant literal, would be applied exclusively on the framed

```
1  public int eval(int x){
2      int a = 3, b = 1, y;
3
4      y = (M_NO==1)? a - x:
5          (M_NO==2)? a + x:
6                     a * x;
7
8      if(M_NO==3){
9          y -= b;
10     }else{
11         y += b;
12     }
13
14     return y;
15 }
```

Original expression

Original statement

Listing 4.2: Mutated statement and expression.

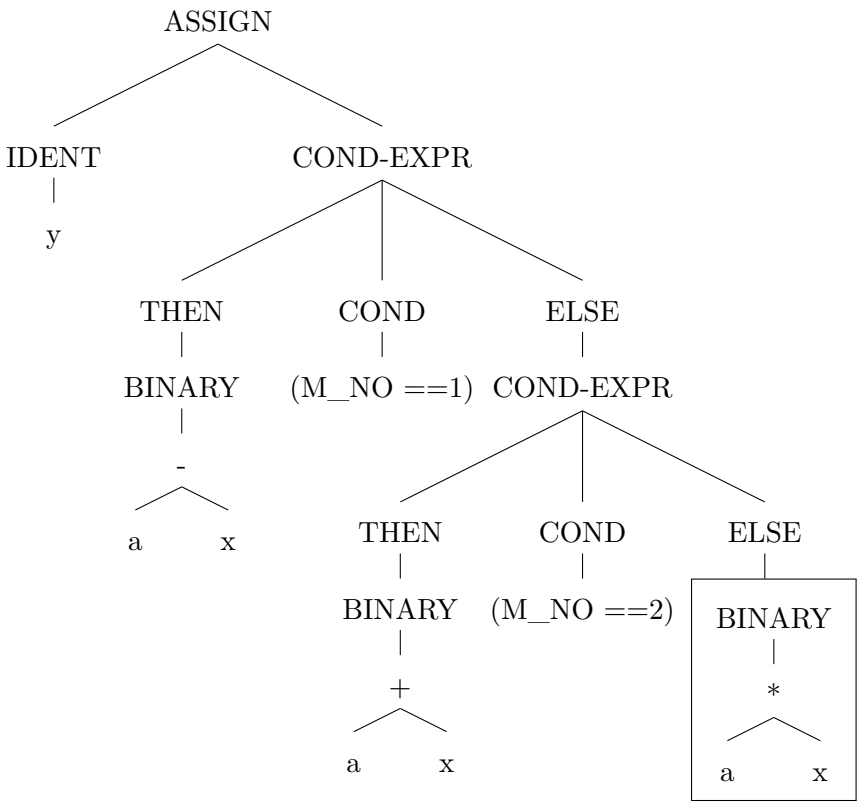
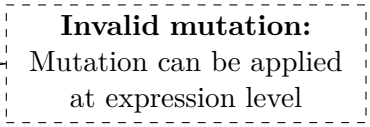


Figure 4.2: Multiple mutated binary expression as right hand side of an assignment.

```
1  public int eval(int x){
2      int a = 3, b = 1, y;
3
4      if(M_NO==1){
5          y = a - x;
6      }else{
7          y = a * x;
8      }
9
10     y += b;
11
12     return y;
13 }
```



Listing 4.3: Invalid mutation of the eval method according to the criterion that only the innermost expression or statement may be affected.

node in order to have exactly one mutant in each `THEN` part. For nested expressions, this condition is crucial for preventing the algorithm from recursively applying transformations on already mutated nodes.

4.2.2 Runtime Optimization with Mutation Coverage

According to the characteristics proposed by Voas [1992], the following three necessary conditions have to be fulfilled to ultimately kill a mutant:

1. The mutated code has to be covered (i.e., reached and executed).
2. The executed mutation has to observably change the internal state.
3. The changed internal state has to be propagated to the output.

This means in turn that all mutants which cannot be covered, meaning that they are not reached and executed, cannot be killed under any circumstances. As a consequence, these mutants could be declared as live mutants without executing the SUT. With respect to conditional mutation where all mutants are encoded together with the original version, the original expression or statement can be replaced again by a conditional expression or statement which additionally collects coverage information. This information should be gathered if and only if the original version is executed, represented by the mutant identifier set to zero. For this purpose the condition $cond_{cov}$ is inserted which is in turn

```
1  public int eval(int x){
2      int a = 3, b = 1, y;
3
4      y = (M_NO==1)? a - x:
5          (M_NO==2)? a + x:
6          (M_NO==0 && COVERED(1,2))?
7              a * x : a * x; // original expr
8
9      if(M_NO==3){
10         y -= b;
11     }else{
12         if(M_NO==0 && COVERED(3,3)){
13             y += b;
14         }else{
15             y += b; // original stmt
16         }
17     }
18
19     return y;
20 }
```

Listing 4.4: Collecting the coverage information with conditional mutation.

a concatenation of the equivalence condition $cond_0$ (`M_NO==0`) and a call of the *covered* method (`COVERED`), which collects the coverage information:

$$CondExpr(cond_{cov}, expr, expr) \longleftarrow\!\shortmid expr \tag{4.5}$$

$$cond_{cov} = (cond_0 \;\&\&\; covered) \tag{4.6}$$

The `COVERED` method takes, as depicted in Listing 4.4, a range as parameters in order to efficiently record expressions or statements which are mutated more than once. In addition, lazy evaluation is exploited by using the logical connector `and` (i.e., `&&`) within the condition and the method always returns `false` in order to fulfill the condition that the right most `ELSE` part contains the original expression or statement. It should be mentioned that the original statement or expression in the `THEN` part could be omitted due to the fact that it is never executed.

Concerning the realized implementation of the approach within the Java Standard Edition compiler, the conditional mutation step is an optional transformation, which can be enabled and configured by means of certain compiler options. Additionally, the global mutant identifier (`M_NO`) and the method to gather the coverage information (`COVERED`) are implemented in an externalized driver class. Chapter 5 describes the compiler integration as well as implementation details and also provides an insight on the available configuration options.

```
1  public int eval(int x){
2      int a = 3, b = 1, y;
3
4      y = MUTANTS[1] ? a - x:
5          MUTANTS[2] ? a + x:
6                       a * x;
7
8      if( MUTANTS[3] ){
9          y -= b;
10     }else{
11         y += b;
12     }
13
14     return y;
15 }
```

Higher order mutation:
Both conditions can be
true at the same time

Listing 4.5: Enhanced conditional mutation to support higher order mutation.

4.2.3 Support for Higher Order Mutation

Conditional mutation can also be extended to support higher order mutation. The key advantage of having all mutations within one file triggered by a certain condition provides the opportunity to adapt the conditions so that multiple mutants are executed. Depending on the order of higher order mutation (i.e., the number of combined first order mutants), there are, for instance, the two following options:

- Use a bitmask to encode several mutant identifiers
- Use an array of mutant identifiers

The bitmask option might be more efficient but is limited to a small number of identifiers. The concrete number is determined by the trade-off between the maximum number of first order mutants and the level of higher order mutation. Using 4 identifiers with 16 bits each would therefore limit the number of first order mutants to 65,536.

A proof of concept using an array of mutant identifiers has also been implemented into the compiler and preliminary runtime results are also promising. Listing 4.5 shows the result of the higher order mutation transformation where the mutant identifier `M_NO` has been replaced by an array access of `MUTANTS`. Now, several mutants can be enabled at the same time by setting the corresponding boolean value in the array to `true`. However, since the focus of this chapter is the design and empirical evaluation of conditional mutation, we leave the complete investigation of higher order conditional mutation as future work, which is further discussed in Chapter 10.

Table 4.1: Summary of the applications investigated in the empirical study.

Application	Source files	Program LOC*	Number of mutants	Number of tests	Test LOC*
aspectj	1,975	372,751	406,382	859	17,069
apache ant	764	103,679	60,258	1,624	24,178
jfreechart	585	91,174	68,782	4,257	48,026
itext	395	74,318	124,184	63	1,393
java pathfinder	543	47,951	37,331	165	12,567
commons math	408	39,991	67,895	2,169	41,907
commons lang	85	19,394	25,783	1,937	32,503
numerics4j	73	3,647	5,869	219	5,273
total	4,828	752,905	796,484	11,293	182,916

*Physical lines of code as reported by sloccount (non-comment and non-blank lines)

4.3 Empirical Study

Conditional mutation has been integrated as an optional transformation in the Java Standard Edition (SE) 6 compiler in order to evaluate the approach. Two aspects are of particular interest in this empirical study, namely, the runtime and the memory footprint of the compiler with the enabled conditional mutation option and the runtime of the compiled and instrumented programs. For this purpose, we use the eight applications in Table 4.1 that range from 3,647 to 372,751 lines of code. All of the investigated applications provide a corresponding test suite, whose size in terms of the number of source files and lines of code is also illustrated in the table.

Moreover, the effectiveness of the individual test suites is shown in Table 4.2, where covered means that the corresponding mutants were reached and executed whereas killed denotes all mutants that were eventually detected by the test suite.

In accordance with the selective mutation approach [Offutt et al., 1996], a reduced, but sufficient set of mutation operators [Namin et al., 2008] has been chosen to assess conditional mutation. Table 4.3 shows the mutation operators that have been implemented in a first step. Besides the description, the table also shows an example for each operator. All implemented operators are configurable via compiler options, and thus can be enabled or disabled for the empirical study.

Table 4.2: Number of the generated and covered mutants for all investigated applications.

Application	Generated mutants	Covered mutants	
aspectj	406,382	20,144	(5.0%)
apache ant	60,258	28,118	(46.7%)
jfreechart	68,782	29,485	(42.9%)
itext	124,184	12,793	(10.3%)
java pathfinder	37,331	8,918	(23.9%)
commons math	67,895	54,326	(80.0%)
commons lang	25,783	21,144	(82.0%)
numerics4j	5,869	4,900	(83.5%)

Table 4.3: Implemented mutation opertors.

	Description	Example
AOR	Arithmetic operator replacement	`a + b` ⟼ `a - b`
LOR	Logical Operator Replacement	`a ^ b` ⟼ `a | b`
COR	Conditional Operator Replacement	`a || b` ⟼ `a && b`
ROR	Relational Operator Replacement	`a == b` ⟼ `a >= b`
SOR	Shift Operator Replacement	`a >> b` ⟼ `a << b`
ORB	Operator Replacement Binary: The union of AOR, LOR, COR, ROR, and SOR	
ORU	Operator Replacement Unary	`-a` ⟼ `~a`
LVR	Literal Value Replacement:	
	Change to a positive or negative value and zero	`0` ⟼ `1`
	Reference initializations are replaced by `null`	`a = ref` ⟼ `a = null`

Table 4.4: Compiler runtime to generate and compile the mutants for the aspectj project.

Operator	Mutants	Runtime*	Overhead	Overhead per 1,000 mutants
NONE	0	18.94	0.00%	0.00%
AOR	70,989	21.15	11.67%	0.16%
LOR/ COR	81,733	21.71	14.63%	0.18%
ROR	137,297	22.90	20.91%	0.15%
SOR	47,830	20.84	10.03%	0.21%
ORB	337,849	25.02	32.10%	0.10%
ORU	802	19.21	1.43%	1.78%
LVR	67,731	20.83	9.98%	0.15%
ALL	406,382	25.15	32.79%	0.08%

*Compiler runtime reported in seconds

Table 4.4 illustrates the necessary compile times for applying these operators to the *aspectj* project. All shown runtimes throughout the empirical study are the median of ten individual runs[1]. We do not report additional descriptive statistics or perform further statistical analysis since the runs are deterministic and the timing results exhibit little, if any, dispersion. In addition to the total runtime and overhead, the last column shows a normalized overhead per 1,000 mutants. A smaller value in this column means less overhead and is thus the better result. Regarding the quantity of 406,382 mutants, the total overhead of 33% for generating and compiling these mutants is almost negligible.

In addition to the detailed results for *aspectj*, Figure 4.3 shows the compiler runtimes for all other analyzed projects. The trend lines in this diagram have been computed by means of the gnuplot fit function which uses the method of least mean square error. The gradients of all trend lines are decreasing for a larger number of mutants. That implies that the relative overhead per mutant is decreasing for an increasing number of

[1]Reported runtimes are measured on a Linux machine with Intel Centrino CPU, 4GB of RAM, and kernel version 2.6.32-5-amd64.

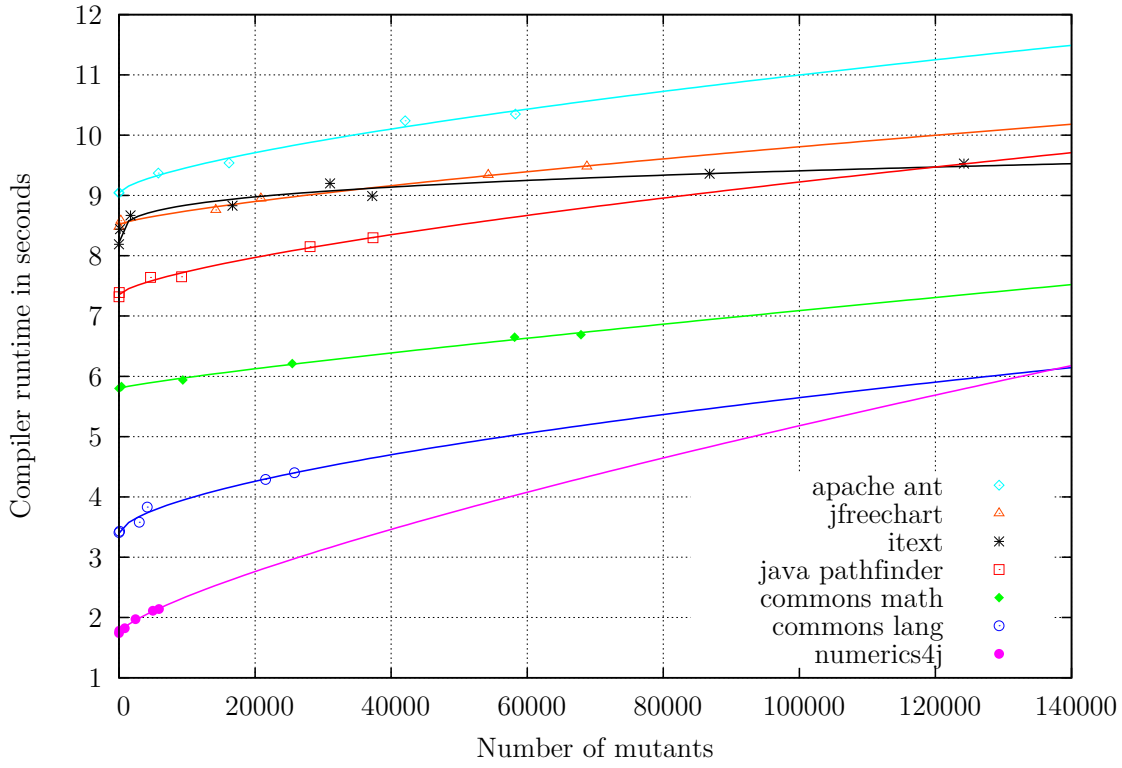


Figure 4.3: Compiler runtime for generating and compiling the mutants for all investigated projects except aspectj.

mutants. The relative overhead per mutant decreases in particular due to the overhead of the initialization of the compiler and the conditional mutation approach. Since the initialization is performed only once, it is less crucial for a larger number of mutants. In order to avoid obscuring the visualization, Figure 4.3 does not contain the higher runtimes associated with generating and compiling the many mutants for *aspectj*. Nevertheless, the trend for the *aspectj* project is equal to the other investigated programs since the relative overhead per 1,000 mutants is also decreasing for a larger number of mutants.

4.3.1 Space Overhead of Mutant Generation

Apart from the runtime of the compiler, we also consider the space overhead in terms of compiler memory consumption and program size. As depicted in Table 4.5, the memory consumption of the compiler ranges from 19.2% to 49.8% and the overhead due to the larger program size varies between 18.0% and 66.1%. Generally, the space overhead is predominantly determined by the ratio of number of mutants to lines of code, as shown in Table 4.1. Concerning the memory footprint of the compiler, the average overhead is 36.2%. Thus, the enhanced compiler with conditional mutation can easily run on commodity workstations, even for large projects that yield a significant number of mutants.

Table 4.5: Incurred space overhead when applying conditional mutation.

Application	Memory consumption*			Size of bytecode*		
	original	instrumented		original	instrumented	
aspectj	559	813	(45.4%)	18,368	30,508	(66.1%)
apache ant	237	293	(23.6%)	6,976	8,228	(17.9%)
jfreechart	220	303	(37.7%)	4,588	5,896	(28.5%)
itext	217	325	(49.8%)	4,140	6,580	(58.9%)
java pathfinder	182	217	(19.2%)	4,052	5,096	(25.8%)
commons math	153	225	(47.1%)	3,124	4,464	(42.9%)
commons lang	104	149	(43.3%)	968	1,456	(50.4%)
numerics4j	73	90	(23.3%)	408	508	(24.5%)

*Memory consumption of the compiler in MB and size of compiled program in KB

4.3.2 Time Overhead of Mutant Execution

Besides the incurred time and space overhead of the enhanced compiler, the execution time of the instrumented and compiled programs is also important. In order to determine the runtime overhead associated with the insertion of the conditional statements and expressions, the analyzed applications are executed by means of their test suites. To establish an upper bound on time overhead, we consider the worst-case scenario for conditional mutation when `M_NO==0` and thus every condition has to be evaluated. Furthermore, mutation coverage, as described in Section 4.2, can be applied to determine the mutants that cannot be killed by the test suite. This information is collected by means of method calls which represent an additional time overhead. Therefore, we also measure the runtime of the test suites with mutation coverage enabled.

The corresponding results for both runtime analyses are depicted in Table 4.6, where *worst case* and *cov* denote *worst-case scenario* and *coverage*, respectively. The time overhead for the worst-case scenario ranges from 1.2% for apache ant to 29.4% for java pathfinder. For the worst-case scenario with mutation coverage enabled the overhead ranges between 4.5% and 70.6%. On average the overhead for *worst case* and *worst case + cov* is 15% and 36%, respectively. Gathering the coverage information leads to a larger overhead due to the additional method calls. Nevertheless, it has to be pointed out that the coverage information is determined only once for an instrumented application with a corresponding test suite and hence the overhead is not crucial. The actual overhead depends on the type of application. For instance, *apache ant* does a lot of costly file system operations and thus the additional costs for conditional mutation are negligible. In contrast, *java pathfinder*

Table 4.6: Incurred runtime overhead when applying conditional mutation.

Application	**Original***	**Instrumented***			
		worst case		**worst case + cov**	
aspectj	4.3	4.8	(11.6%)	5.0	(16.3%)
apache ant	331.0	335.0	(1.2%)	346.0	(4.5%)
jfreechart	15.0	18.0	(20.0%)	23.0	(53.3%)
itext	5.1	5.6	(9.8%)	6.3	(23.5%)
java pathfinder	17.0	22.0	(29.4%)	29.0	(70.6%)
commons math	67.0	83.0	(23.9%)	98.0	(46.3%)
commons lang	10.3	11.8	(14.6%)	14.8	(43.7%)
numerics4j	1.2	1.3	(8.3%)	1.6	(33.3%)

*Test suite runtime reported in seconds

and *commons math* represent applications that almost exclusively perform computations, thus explaining why the overhead is more noticeable.

With regard to the empirical results, some threats to validity have to be considered. The choice of the applied mutation operators could be a threat to internal validity since different operators may affect the runtime of the compiler. However, the chosen operators are frequently used in the literature and therefore provide comparable results [Offutt and Untch, 2000; Namin et al., 2008]. A potential threat to external validity might be the representativeness of the selected applications. We cannot guarantee that the depicted overheads will remain the same for other programs. Nevertheless, the investigated applications differ considerably in size, complexity, and operation purpose and most of them are widely used. So, we judge that the reported results are meaningful. Defects in the compiler-integrated prototype could be a threat to construct validity, but we controlled this threat by testing our implementation with a developed test suite and by manually checking the results of several example programs. Thus, we judge that the implementation worked correctly.

4.4 Related Techniques and Limitations

This section discusses two related techniques, namely mutant schemata and bytecode transformation, which both aim at improving the efficiency of mutation analysis.

4.4.1 Mutant Schemata

When compared with mutant schemata, conditional mutation is more general since it handles expressions and statements within the scope of the mutated instruction. Furthermore, not every mutation operator can be implemented by the method call approach described by Untch et al. [1993]. For instance, if the parameter passing method is call by value, which is used exclusively in Java, the following replacements are not valid because they do not preserve the semantics of the original statement:

- `int a = OP(b);` $\not\longmapsto$ `int a = ++b;`
- `OP(a,b);` $\not\longmapsto$ `a += b;`

Even though these statements could be rewritten so that they preserve the semantics while being mutated by means of method calls, this additional transformation can only be achieved with an extension of the original mutant schemata approach. These mutations can, however, be represented with conditional mutants since they exist in the same scope as the original expression.

Moreover, while conditional statements can easily express faults of omission (e.g., a forgotten `continue` or `break` statement), method calls cannot represent this type of mutant. Generally, the mutation with method calls is only applicable to expressions.

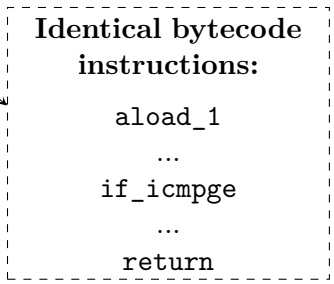
4.4.2 Bytecode Transformation

Another related technique is applying mutation at the bytecode level. Intuitively, an advantage of this technique is that the source code does not have to be available. Another considerable benefit, with regard to the Java virtual machine (JVM), is that the manipulation of bytecode provides mutations for all languages that compile to the JVM. Moreover, the JVM bytecode language is much simpler than the Java programming language, thus reducing the effort of implementing mutation operators.

However, there are several drawbacks to this approach. Since the compiled code is usually simplified or even optimized, this approach might generate mutants that could not be injected at the source code level. Furthermore, it could be infeasible to map the mutants, injected at the bytecode level, back to the source code level due to irreversible mappings or ambiguity. Listing 4.6 shows an example for the Java programming language where two different looping constructs are compiled to identical bytecode due to the fact that the Java compiler transforms the illustrated for-each loop to an ordinary for loop. Now, mutation at the bytecode level would generate several mutants that could not be introduced at the source code level. Moreover, mapping the generated mutants from bytecode to source code is not feasible.

A similar problem arises when taking compiler optimization modifiers, such as `final`, into account. These modifiers are not available at the bytecode level but are checked by the compiler. Hence, a `final` variable, which is considered to be constant by the compiler, could be manipulated at the bytecode level.

```
1  public void printArray (int[] array){
2      for(int i : array){
3          System.out.println(i);
4      }
5  }
6
7  public void printArray (int[] array){
8      int[] tmp = array;
9      int length = tmp.length;
10     for(int i=1; i<length; ++i){
11         int k = tmp[i];
12         System.out.println(k);
13     }
14 }
```



Listing 4.6: Two different looping constructs that compile to the same bytecode.

Overall, we judge that mutation at the source code level provides a more realistic and more comprehensive set of defects that could have been done by a programmer. We thus exclusively rely on mutation at the source code level. However, the bytecode approach might be more beneficial for other mutation purposes, such as mutation-based test data generation (e.g., [Fraser and Arcuri, 2011]).

4.4.3 Limitations

The JVM specification [Lindholm and Yellin, 2000] defines an upper bound of 64k instructions for the size of a method represented in bytecode. Given this restriction, a large or complex method could, when mutated many times, violate this constraint due to the embedded conditional instructions. However, this restriction is generally a limiting factor for all template-based approaches including those discussed in this section. Possible solutions to overcome this problem are for instance:

- Reduce the number of mutants
- Add method clones and indirections where each clone contains a disjoint subset of all generated mutants

During all empirical studies, we encountered only one case where the limitation was violated when a statically initialized array with more than 20,000 entries was mutated. So, we judge that this limit is not a major problem in general and do not further investigate this matter.

4.5 Summary

This chapter has addresses the challenges associated with increasing the efficiency of mutation analysis. It presents a new method called conditional mutation that reduces the generation time and enables efficient mutation analysis. It is, compared with the conventional way, much more efficient and offers the possibility of applying mutation analysis to large software systems. Moreover, mutation testing, where mutants are generated and executed iteratively, becomes more feasible since the generation time is reduced to a minimum. The approach is versatile, programming language independent, and can be integrated within the compiler.

So far, conditional mutation has been implemented in the Java Standard Edition compiler and it has been applied to applications up to 372,751 lines of code. Furthermore, conditional mutation can be easily combined with other do smarter and do fewer approaches since the set of mutation operators is configurable and the mutation analysis can be parallelized and processed in a distributed environment. .

Chapter 5

Mutation Analysis in a Java Compiler

Parts of the content of this chapter have been published in [Just et al., 2011b]

The previous chapter presented a new approach to mutant generation and provided an empirical study for its evaluation. Within this chapter, we focus on the details of the corresponding implementation, which is called Major. Besides presenting various configuration options of Major, this chapter also motivates, defines, and describes a domain specific language for mutation analysis. Although this language is generally implementation independent and applicable for any mutation tool for Java programs, the corresponding compiler is currently implemented to be compatible with Major. Therefore, this chapter additionally outlines the characteristics and the integration of the implemented domain specific language compiler.

5.1 Introduction

Mutation analysis is an effective, yet often time-consuming and also difficult-to-use method for the evaluation of testing strategies. Generally, mutation analysis is not feasible for real-world applications without proper tool support. With regard to the tool support, various systems and frameworks for different programming languages have been developed, which differ with respect to their efficiency, flexibility, available mutation operators, and the degree of automation. While some tools such as MuJava [2009] are outdated due to incompatibility with new language standards and common testing frameworks, others do not provide configurable and adaptable mutation operators or simple fault seeding (e.g., [Irvine et al., 2007; Schuler et al., 2009]). Hence, there is a need for a new flexible tool that enables basic research as well as the application of mutation analysis.

Addressing these aspects, this chapter presents Major, a compiler-integrated and non-invasive tool that provides fast fault seeding for arbitrary purposes, thus enabling efficient mutation analysis. Designed as a non-invasive enhancement of the Java Standard Edition reference compiler [OpenJDK, 2012] for use in any Java-based development environment,

Major implements the conditional mutation approach as an optional transformation of the abstract syntax tree. Moreover, Major's configuration options for specifying and adapting mutation operators also makes it flexible and extensible. The name Major is an acronym reflecting that it is a tool for **m**utation **a**nalysis in a **J**ava c**o**mpile**r**. In contrast to existing tools such as Jumble [Irvine et al., 2007], MuJava [Ma et al., 2006], or Javalanche [Schuler et al., 2009], Major is integrated into the Java compiler and does not require a specific mutation analysis framework. Hence, it can be used in any Java-based development environment. The main contributions of Major can be summarized as follows:

- Efficient mutation analysis by means of embedded mutants and mutation coverage information
- Integrated into the Java Standard Edition compiler
- Non-invasive and compiler-integrated implementation
- Fast and flexible fault seeding with built-in mutation operators
- Extensibility through its domain specific language

In the remainder of this chapter, Section 5.2 briefly reviews the implemented approach to mutation analysis to ensure that this chapter is self-contained. Next, Section 5.3 provides detailed information about the implementation of the individual components in Major. This section also focuses on the externalized configuration options and the necessary driver. Then, Section 5.4 motivates and presents a domain specific language for mutation analysis, which is called Mml. Moreover, this section explains implementation details and how the Mml compiler is integrated with Major. Finally, Section 5.5 concludes this chapter.

5.2 Conditional Mutation

The conditional mutation approach generates mutants by transforming the abstract syntax tree, as described in Chapter 4. It uses conditional expressions and statements to encapsulate all of the mutants and the original version of the program in the same basic block. A mutated example method is shown in Listings 5.1, where the binary expression `a * x` is mutated. A concrete mutant can be triggered by setting the mutant identifier `M_NO` to the mutant's number at runtime. It is important to note that this mutant identifier has to be externalized to provide a unique access to it and a consistent mutation numbering in multi class software systems.

A mutant that is not reached and executed cannot be detected under any circumstance. In order to avoid evaluating these uncovered mutants, conditional mutation supports the collection of additional information about the coverage of the mutations at runtime. Listing 5.2 shows the extension of the mutated binary expression, which provides the mutation coverage information. For efficiency reasons, the covered mutants are reported as ranges within the `COVERED` method, which again has to be externalized for consistency reasons.

```
1 public int eval(int x){
2     int a = 3, b = 1, y;
3
4     y = (M_NO==1)? a + x:
5         (M_NO==2)? a / x:
6         (M_NO==3)? a % x:
7                    a * x; // original
8
9     y += b;
10
11    return y;
12 }
```

Listing 5.1: Example method with an expression mutated with conditional mutation.

```
1 public int eval(int x){
2     int a = 3, b = 1, y;
3
4     y = (M_NO==1)? a + x:
5         (M_NO==2)? a / x:
6         (M_NO==3)? a % x:
7         (M_NO==0 && COVERED(1,3))?
8            a * x : a * x; // original
9
10    y += b;
11
12    return y;
13 }
```

Listing 5.2: Gathering coverage information for a mutated expression with conditional mutation.

5.3 Implementation Details

In general, a compiler transforms the source code representation of a given program into intermediate or assembled code. With regard to the Java programming language, this process of compiling a source file into bytecode can be divided into the following consecutive compiler phases:

- Parse: Build the abstract syntax tree (AST) by parsing the source code
- Attribute: Enhance the AST with semantic information (e.g., type information)
- Flow: Traverse the AST and perform semantic and data flow analyses
- Lower: Simplify the AST and remove syntactic sugar
- Generate: Generate bytecode from simplified AST

Now the question arises as to the best method for integrating conditional mutation into the compilation process. Obviously, a parsed AST is necessary to apply a transformation to it. Furthermore, it is advisable to transform the AST before the flow analysis and the lower step for two reasons. On the one hand the code which shall be mutated should not be simplified or desugared previously. On the other hand the mutated code should also be checked by the compiler in order to avoid an incorrect AST and thus invalid code.

As a consequence, only two options remain for the integration, namely before or after the attribution step. Applying conditional mutation after the attribution step is slightly more complex since the additional nodes and subtrees which shall be inserted also have to be attributed by the mutation process. However, an attributed AST provides a lot of semantic information (e.g., type information) which offers more subtle mutations. Overall, the advantages of the second option outweigh the first and hence the conditional mutation approach is implemented as an additional, but optional, transformation after the attribution step.

Within Major, conditional mutation is implemented as an optional transformation of the compiler's AST, which can be enabled by means of common compiler options. If the conditional mutation step is not chosen, then the compiler works exactly as if it were unmodified. The compile-time configuration of conditional mutation and the necessary runtime driver are stored externally in order to avoid dependencies and to provide a non-invasive tool. Overall, Major's implementation was driven by the following considerations:

- The default behavior of the compiler must not be changed. This criterion is obligatory in order to have one compiler applicable within the build environment.
- The necessary changes within existing compiler classes should be kept to a minimum to ensure that conditional mutation can be implemented in future releases of the Java compiler with little or no additional effort.
- All configuration possibilities have to be externalized to forestall the need to recompile Major for different purposes. Moreover, conditional mutation within the compiler must support the default command-line interface so that it can be used standalone, in integrated development environments (IDEs), and with frequently used build systems such as Apache Ant.
- Major must provide a sufficient set of mutation operators in order to provide meaningful results that are comparable to prior empirical studies [Offutt et al., 1996; Namin et al., 2008].

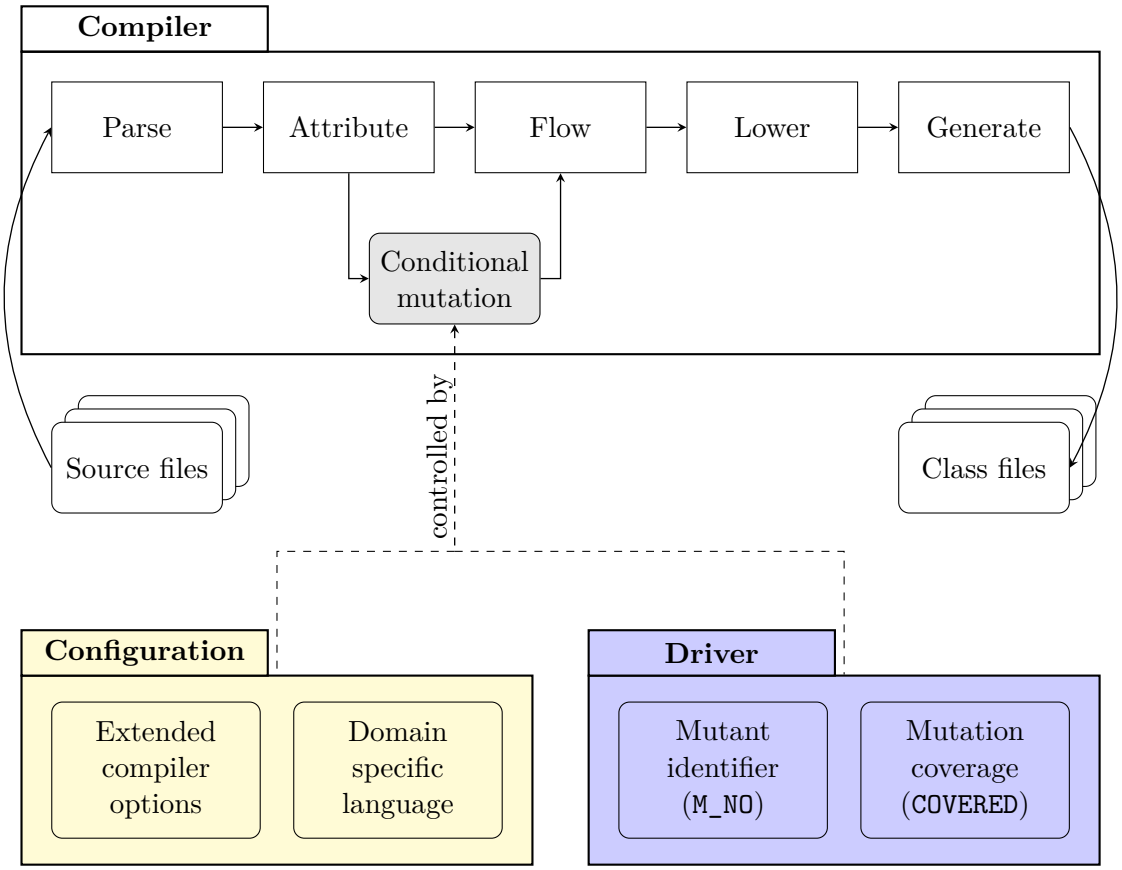


Figure 5.1: Overview of compiler-integrated conditional mutation with externalized configuration and driver.

Figure 5.1 visualizes the integration of conditional mutation into the Java compiler. It is important to note that the implementation of the optional mutation phase requires a modification of the existing code base since there is currently no support for arbitrary tree transformation phases. The diagram, moreover, depicts the two externalized components, namely the configuration and driver, used to control the mutation phase.

Concerning the integration of this mutation phase into the OpenJDK [2012] Java compiler, the existing architecture has to be taken into account. All transformation and analysis phases are based on visitor implementations (cf. [Gamma et al., 1995]) performing a tree traversal where two kinds can be distinguished:

- TreeVisitor: Traverses the tree without modification
- TreeTranslator: Traverses and modifies the tree

Figure 5.2 offers a more detailed view on the implementation by visualizing a simplified UML class diagram of the extensions necessary to enhance the Java compiler.

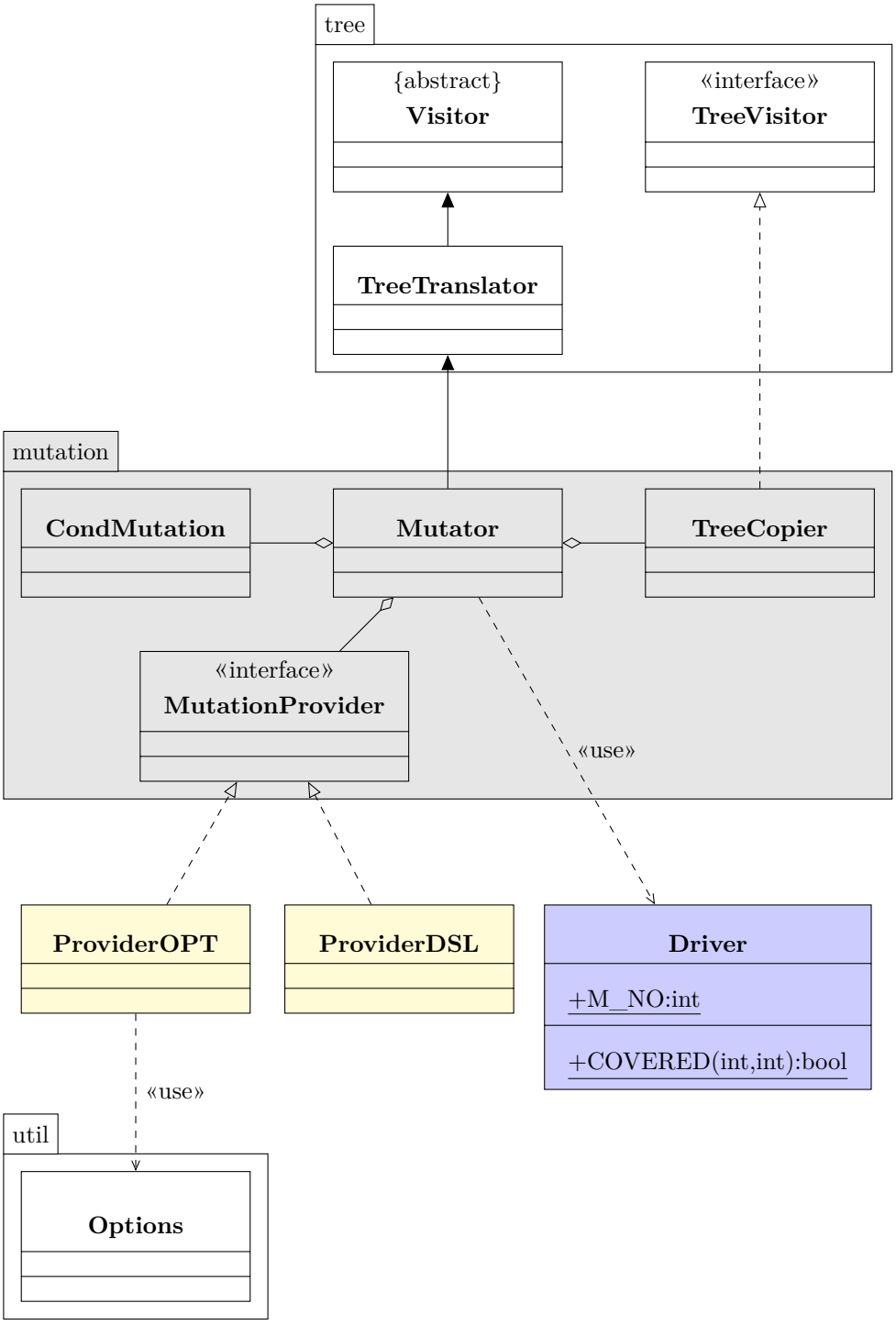


Figure 5.2: UML class diagram of the necessary extensions of the Java compiler.

As shown in Figure 5.2, the `Mutator` is the main component that modifies the AST, and hence has to be a `TreeTranslator`. Based on a given `MutationProvider`, the `Mutator` injects the chosen mutants by applying the `CondMutation` strategy. Since every mutant is a slightly modified version of the original statement or expression, a `TreeCopier` is used that provides an exact clone of an arbitrary AST node. Due to the fact that the copier does not change the original AST nodes, it is implemented as a `TreeVisitor`.

5.3.1 Configuration

In order to use the mutation capabilities of Major, the conditional mutation step has to be generally enabled at compile-time using the corresponding compiler option. Major also provides either additional compiler options or mutation scripts to support the compile-time configuration of the mutation process. The compiler options can be used to choose from the various built-in mutation operators that apply predefined replacements (e.g., for fast fault seeding). The following mutation operators and operator groups are currently available within Major via compiler options (the operators are described with examples in detail in Table 4.3, page 38):

- **ORB**: Operator Replacement Binary (AOR,LOR,ROR,COR,SOR)
- **ORU**: Operator Replacement Unary
- **LVR**: Literal Value Replacement

In order to avoid potential conflicts with future releases of the Java compiler, Major extends the non-standardized `-X` options of the compiler. The conditional mutation step can generally be enabled with `-XMutator`. Furthermore, this option provides a wildcard and a list of valid sub-options, which correspond to the names of the mutation operators. For instance, the following two commands enable all operators by means of the wildcard `ALL` (1) and a specified subset of the available operators `AOR`, `ROR`, and `ORU` (2):

(1) `javac -XMutator:ALL ...`

(2) `javac -XMutator:AOR,ROR,ORU ...`

Instead of using compiler options, Major can parse mutation scripts written in a domain specific language. These scripts enable a detailed definition and a flexible application of mutation operators. For example, the replacement list for every operator in an operator group can be specified and mutations can be enabled or disabled for certain packages, classes, or methods. The scripting language enhances the predefined mutation options while using the compiler options' keywords for the operators. Major's domain specific language is explained and described in detail in the subsequent Section 5.4.

```
1 package major.mutation;
2
3 public class Driver{
4     public static final int MAX_NO=100000;
5     public static int[] COV = new int[MAX_NO];
6
7     public static int M_NO=0;
8
9     public static boolean COVERED(int from, int to){
10        for(int i=from; i<=to; ++i){
11            COV[i]++;
12        }
13        return false;
14    }
15 }
```

Listing 5.3: Simple driver class with mutant identifier and coverage method.

5.3.2 Driver Class

The conditional mutation components reference the external driver to gain access to the mutant identifier `M_NO`. Additionally, the driver has to furnish the mutation coverage method `COVERED` if mutation coverage has been enabled within the compiler. Listing 5.3 shows an example of a driver class that provides both the mutant identifier and the mutation coverage method that gathers the coverage information at runtime. The identifier and the coverage method must be implemented in a static context to avoid any overhead caused by polymorphism and instantiation. Nevertheless, the fully qualified name of the driver class itself can be configured.

In order to keep Major non-invasive, the driver class does not have to be available on the classpath during compilation. That means that Major does not try to resolve the driver class at compile-time but instead assumes that the mutant identifier and the coverage method will be available in this class at runtime. Thus, the mutants can be generated without having a driver class available during compilation.

5.4 Major Mutation Language

As previously indicated in this chapter, mutation analysis can only be effective if the applied mutation operators can be tailored to generate subtle mutants. Moreover, fundamental research and replications of previous studies require configurable mutation operators, and thus prevent from using a hard-coded integration. Overall, detailed control over

the mutation process is desirable but demands a highly configurable mutation tool. For instance, the following, not necessarily exhaustive, list of aspects have to be considered when designing a mutation tool with adjustable parameters:

Extensive class hierarchies

Generally, object-oriented software systems are not designed to be monolithic and they therefore contain several classes, mostly structured in hierarchical packages. In order to focus on a specific part of the software system, a mutation tool has to provide an option to define which package, class, or method should be included or excluded.

Several Mutation operators

There is a significant number of applicable mutation operators and especially the replication of previous studies and fundamental research in the field of selective mutation requires a configuration possibility to define which operators shall be included or excluded.

Different Replacement lists

When applying mutation operators that replace programming language operators (e.g., binary arithmetic operators), different replacements for operators of the same type might be sufficient. Hence, a mutation tool should provide the possibility to define replacement lists for the individual mutation operators.

Even though the number of these example parameters is relatively small, the number of their possible values is rather large, and thus leading to a tremendous number of feasible combinations of all possible parameter values. As a consequence, using simple configuration flags or files providing key-value pairs is not suitable to cover the whole range of configuration options. To enable both fundamental research as well as empirical studies and to take into account the aspects mentioned above, Major is designed to support a wide variety of configurations by means of its own domain specific language. This section describes this language called Mml (Major mutation language) in detail, provides implementation details of the corresponding Mml compiler, and also gives some example source files.

5.4.1 Grammar for Mml

We employ ANTLR [Parr, 2007; Parr and Fisher, 2011] to generate the lexer and parser of the Mml compiler for two reasons. It is a sophisticated tool that has been successfully applied to a wide variety of projects and it is based on the Java programming language, hence providing many customization options for this language. In order to employ ANTLR to generate a recursive descent parser, we have to provide a LL grammar that it can process. The following syntax definitions focus on some selected properties of the domain specific language that are necessary to later explain the implementation details. The complete grammar for the Mml lexer and parser can be found in Appendix A.

Generally, a Mml script contains a sequence of an arbitrary number of statements, where a statement represents one of the following entities:

- Variable definition (*vardef_stmt*)
- Invocation of an operator (*call_stmt*)
- Replacement definition (*replace_stmt*)
- Definition of own operators (*opdef_stmt*)
- Line comment (`LINE_COMMENT`)

Figure 5.3 visualizes the grammar rules by means of a syntax diagram in which all non-terminals are represented by rectangles and lower-case labels. The terminal symbols are visualized with upper-case labels in the rectangles with rounded corners. While the first three statements are terminated by a semicolon, an operator definition is encapsulated by curly braces and a line comment is terminated by the end-of-line.

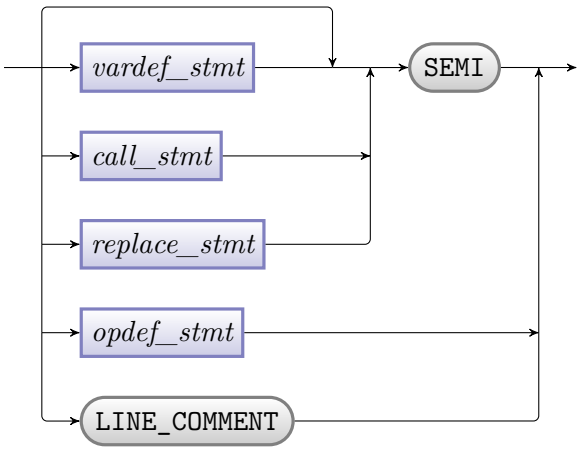


Figure 5.3: Syntax diagram for the definition of a statement.

In order to support the mutation of a certain package, class, or method within large software systems, Mml provides statement scopes for replacement definitions and operator invocations. Figure 5.4 depicts the definition of a statement scope, which can cover software units at different levels of a given hierarchy, from a specific method up to an entire software package. It is important to note that the first rule of Figure 5.4 implies that such a statement scope is optional and can be omitted. That is, if no statement scope is provided, the corresponding replacement definition or operator call is applied to the highest level of the given hierarchy. The scope's corresponding software entity, that is package, class, or method, is determined by means of its fully qualified name, which is referred to as flatname. Such a flatname can be either provided within delimiters (`DELIM`) or by means of a variable identified by `IDENT`.

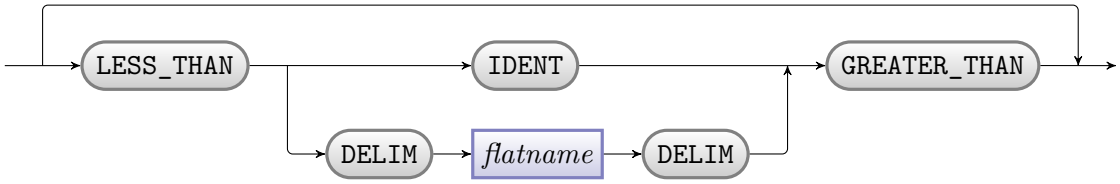


Figure 5.4: Syntax diagram for the definition of a statement scope.

The grammatical rules for assembling a flatname are shown in Figure 5.5, again by means of a syntax diagram. The naming conventions for valid identifiers (`IDENT`) are based on those of the Java programming language due to the fact that a flatname identifies a certain entity within a Java software system. The following three examples, are for instance valid flatnames for a package, class, and method:

- "java.lang"
- "java.lang.System"
- "java.lang.System@exit"

It has to be pointed out that the definition of a flatname also supports the identification of innerclasses and constructors, consistent with the naming conventions of the Java compiler. For Example, the subsequent definitions address an inner class, a constructor, and a static class initializer:

- "foo.Bar$InnerClass"
- "foo.Bar@<init>"
- "foo.Bar@<clinit>"

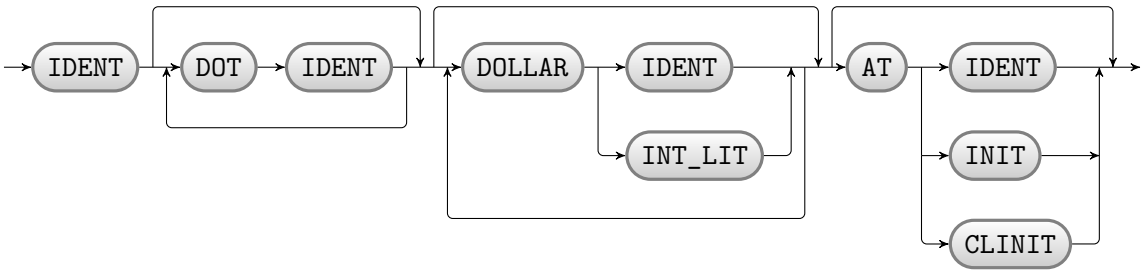


Figure 5.5: Syntax diagram for the definition of a flatname.

5.4.2 Integration with MAJOR

Recalling the design goals of Major, we have to consider the following two aspects in order to decide how to integrate the Mml compiler with Major:

- The configuration has to be externalized to prevent recompilation.
- Major must not have any dependencies to be non-invasive.

In consideration of the second requirement, integrating the Mml compiler into Major is only feasible if the Mml compiler does not manifest dependencies to any libraries apart from the default ones provided by the Java runtime environment. Due to the fact that we employ ANTLR for lexer and parser generation, we cannot fulfill this requirement with the generated parts since they reference the ANTLR runtime library. Therefore, we externalize the entire Mml compiler and use an intermediate output, which can be interpreted by Major. Naturally, this output format has to be readable without any additional libraries, again in compliance with the second requirement.

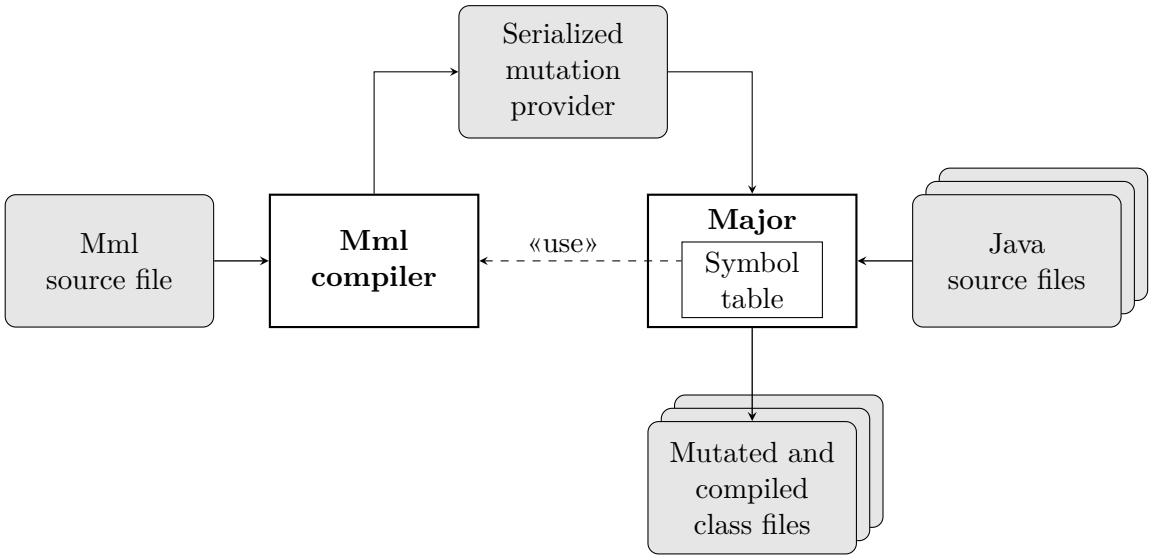


Figure 5.6: Integration of the Mml Compiler with Major using standard Java serialization as intermediate output.

In order to avoid any additional parsing overhead within Major and also to provide a checked intermediate output, we rely on the standard Java serialization. Figure 5.6 shows an overview of the interaction between the Mml compiler and Major. It is important to note that the Mml compiler references Major to gain access to its symbol table and to the interface and class definition of the mutation provider. The former is essential to allow the Mml compiler to semantically check the defined operator replacements and to use the correct operator symbols when producing the output file by serializing the mutation provider.

The mutation provider is based on a tree structure and provides accessor methods for getting the enabled mutation operators and the corresponding replacement lists. Figure 5.7 visualizes such a mutation tree that contains the defined operator replacement lists as well as the enabled and disabled mutation operators, which are highlighted with the gray rectangles. Generally, the accessor methods do not manipulate the tree and hence the attributed tree is immutable, once it has been created by the Mml compiler. Within the given example, the LVR mutation operator is enabled and three replacements for relational operators are defined on the root node. Furthermore, the ROR mutation operator is enabled for the package *org* and for the class *de.uni.ulm.Foo*. For the latter, the COR mutation operator is also enabled and the LVR operator is disabled, thus overriding the definition for the root node. It is crucial to state, that the enabled mutation operators and replacement lists for a given node, that is a certain package, class, or method, is determined by traversing the tree and analyzing all visited nodes, where the individual definitions may carry additional flags.

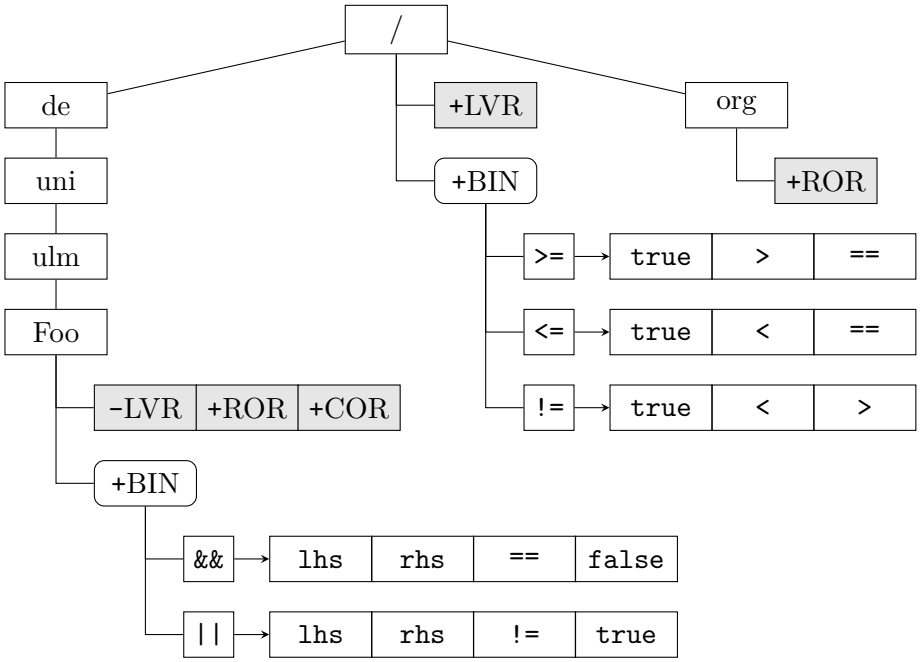


Figure 5.7: Attributed mutation tree that provides replacement lists and enables/disables certain mutation operators.

The mutation operators can either be enabled (`+`) or disabled (`-`) and this behavior can be defined for each node. If there are several definitions on a given path, as for instance shown in Figure 5.7, the innermost node, that is the last one on the path, determines the result. With regard to the replacement definitions, there are two different possibilities. Individual replacements can be added (`+`) to the list or the entire replacement list can be overridden (`!`), where the latter possibility represents the default case if no additional flag is provided.

```
1 // Define own replacement list for AOR
2 BIN(*) -> {/,%};
3 BIN(/) -> {*,%};
4 BIN(%) -> {*,/};
5
6 // Define own replacement list for ROR
7 BIN(>)  -> {<=,!=,==};
8 BIN(==) -> {<,!=,>};
9
10 // Enable and invoke mutation operators
11 AOR;
12 ROR;
13 LVR;
```

Listing 5.4: Simple script to define replacements for the AOR and ROR mutation operators and to enable AOR, ROR, and LVR on the root node.

5.4.3 Script Examples

Listing 5.4 shows a simple example of a mutation script that includes the following tasks:

- Define specific replacement lists for AOR and ROR
- Invoke the AOR and ROR operators on reduced lists
- Invoke the LVR operator without restrictions

This script does not use any scoping or overriding capabilities and can thus be applied to an arbitrary software system for which only AOR, ROR, and LVR mutants shall be generated.

Reconsidering the mutation tree depicted in Figure 5.7, the corresponding script is shown in Listing 5.5. Within this script we exploit the scoping capabilities of the Mml in line 8 and 12-17. Additionally, we take advantage of the possibility to define a variable in line 11 to avoid code duplication in the subsequent scope declarations. Both features are useful if only a certain package, class, or even method shall be mutated in a hierarchical software system.

In order to further avoid code duplication for the repeating application of equal definitions, that is the same replacements and enabled mutation operators for several packages, classes, or methods, the Mml provides the possibility to declare own operator groups. The last given example in Listing 5.6 visualizes this grouping feature that allows for a definition of an own mutation operator group. Such a group may in turn contain any statement that is valid in the context of the Mml, apart from a call of another operator group.

```
1 // Definitions for the root node
2 BIN(>=)->{TRUE,>,==};
3 BIN(<=)->{TRUE,<,==};
4 BIN(!=)->{TRUE,<,> };
5 LVR;
6
7 // Definition for the package org
8 ROR<"org">;
9
10 // Definitions for the class Foo
11 foo="de.uni.ulm.Foo";
12 BIN(&&)<foo>->{LHS,RHS,==,FALSE};
13 BIN(||)<foo>->{LHS,RHS,!=,TRUE };
14
15 -LVR<foo>;
16 ROR<foo>;
17 COR<foo>;
```

Listing 5.5: Mutation script that produces the mutation tree illustrated in Figure 5.7.

```
1 myOp{
2     // Definitions for the operator group
3     BIN(>=)->{TRUE,>,==};
4     BIN(<=)->{TRUE,<,==};
5     BIN(!=)->{TRUE,<,> };
6     BIN(&&)->{LHS,RHS,==,FALSE};
7     BIN(||)->{LHS,RHS,!=,TRUE };
8
9     ROR;
10    COR;
11 }
12
13 // Calls of the defined operator group
14 myOp<"org">;
15 myOp<"de">;
16 myOp<"com">;
```

Listing 5.6: Mutation script with a definition of an own mutation group and corresponding calls for different scopes.

5.5 Summary

This chapter has described Major, a fault seeding and mutation analysis system integrated into the Java Standard Edition compiler. Designed as a non-invasive tool, it is applicable in every Java-based environment and customizable with common compiler options and its own domain specific language Mml. Major implements conditional mutation to transform the attributed abstract syntax tree and enables efficient mutation analysis by means of several optimizations, such as mutation coverage information. Its domain specific language Mml also makes it extensible and configurable. Due to its ease-of-use, efficiency, and extensibility, Major is an ideal platform for fundamental research and the application of mutation analysis.

Chapter 6

Non-Redundant Mutation Operators

The content of this chapter has been published in [Just et al., 2012a]

The high cost of mutation analysis is due, in part, to the fact that many mutation operators generate redundant mutants that may both misrepresent the mutation score and increase the runtime of the mutation analysis process. This chapter shows how the mutation operator for the replacement of conditional operators can be defined in a redundant-free manner. Furthermore, it investigates how prevalent redundant mutants are and how they affect the effectiveness and efficiency of mutation analysis.

6.1 Introduction

A wide variety of mutation operators have been proposed for different purposes and programming languages (cf. [King and Offutt, 1991; Ma et al., 2006; Jia and Harman, 2011]). However, applying all mutation operators results in a substantial number of mutants, especially for large software systems, and thus executing and analyzing all of the mutants can be very expensive. In response to this challenge, Offutt et al. [1996] studied the effectiveness of a subset of mutation operators, revealing that this smaller group of sufficient mutation operators could be applied without a substantial loss of information. Furthermore, Namin et al. [2008] confirmed the results that a subset of all applicable mutation operators is sufficient to achieve a meaningful result. However, these studies focused on reducing the number of mutation operators without incurring a major loss in the accuracy of the mutation score and regarded the operators to be atomic. That is, an mutation operator was either excluded or applied with all valid transformations or replacements.

This chapter considers these operators at a fine-grained level and shows that their original definition implies redundancy in the resulting set of mutants. While the actual subset of mutation operators that can be employed depends on the programming language, this

chapter considers the following set of operators that were commonly used in previous experiments (e.g., [Namin et al., 2008; Schuler and Zeller, 2009]). A more detailed description of the mutation operators is depicted in Table 4.3, page 38.

- Operator Replacement Unary (ORU): Replace all occurrences of unary operators with all valid alternatives.
- Operator Replacement Binary (ORB): Replace all occurrences of binary operators with all valid alternatives, including AOR, LOR, SOR, ROR, and COR.
- Unary Operator Insertion (UOI): Insert a unary boolean operator to negate boolean expressions. This operator is also applied to subexpressions and boolean literals.
- Literal Value Replacement (LVR): Replace all literals with a positive value, a negative value, and zero. Additionally, all reference type variables are replaced by a reference to `null`.

The empirical study in this chapter additionally demonstrates how prevalent redundancies in mutant sets are in real-world applications and how the inclusion of redundant mutants leads to an inaccurate mutation score, thus making this metric less meaningful. In addition to focusing on effectiveness, this chapter empirically demonstrates how reducing the set of mutants decreases the runtime of the mutation analysis process. In consideration of the effect of redundant mutants on efficiency and effectiveness of mutation analysis, the contributions of this chapter can be summarized as follows:

- A definition of non-redundant mutants that exhibit a minimal impact and collectively form a sufficient set of replacements for binary operators.
- A demonstration, based on the given definition, that the COR operator for replacing conditional binary operators with all valid alternatives should only apply a subset of replacements to avoid the creation of redundant or trivial mutants.
- A determination of the actual number of mutants generated by applying the COR and ROR operators. Using a well-known subset of mutation operators, the empirical study computes the ratio of the number of COR and ROR mutants to the size of the entire set of mutants.
- An empirical study that investigates how redundant mutants affect the effectiveness and efficiency of mutation analysis for ten real-world programs that range in size from 3,000 to more than 110,000 lines of code.
- A comparison of the computed mutation score with the mutation and code coverage ratios for the same ten real-world applications. Additionally, the study measures the necessary runtime to compute the three ratios and discusses the results with regard to the quality of the investigated test suites.

In the rest of this chapter, Section 6.2 furnishes a more detailed view on the mutation operators and defines a sufficient and minimal set of replacements for the COR operator. Next, Section 6.3 describes the empirical study and reports on the corresponding results. Section 6.4 discusses related work and, finally, Section 6.5 concludes.

6.2 A Detailed View on Mutation Operators

To ensure that mutation analysis is effective, it is often important for a mutant to have only a small impact on the output, thus making it hard to detect. Trivial and redundant mutants also should be avoided to reduce the runtime of the mutation analysis process and to not misrepresent the mutation score. We refer to mutants that result in a wrong output for all possible input values as trivial mutants.

Previous studies on the effectiveness of mutation analysis that investigated the reduction of mutation operators (e.g., [Offutt et al., 1996; Namin et al., 2008]) did not take the definitions of the operators into account and considered them to be atomic. More recently, Kaminski et al. [2011] investigated the relational operator replacement and showed that only three replacements are necessary to subsume all the others. This section considers the conditional operator replacement (COR) mutation operator for the logical connectors `&&` and `||` at a fine-grained level. It first formally describes the requirements for mutants that manifest a minimal impact in order to avoid the creation of subsumed and trivial mutants. Based on the given definitions, it then suggest a sufficient set of mutations for both logical connectors.

Generally, the COR mutation operator replaces an expression `a <op> b` where `a` and `b` denote boolean expressions or literals and `<op>` is one of the logical connectors `&&` or `||`. With regard to binary conditional operators, valid mutations belong to one of the following three categories:

1. Apply conditional operator
    - Apply logical connector `AND`: `a && b`
    - Apply logical connector `OR`: `a || b`
    - Apply equivalence operator: `a == b`
    - Apply exclusive OR operator: `a != b`
2. Apply special operator
    - Evaluate to left hand side: `lhs`
    - Evaluate to right hand side: `rhs`
    - Always evaluate to true: `true`
    - Always evaluate to false: `false`
3. Insert unary boolean operator
    - Negate left operand: `!a <op> b`
    - Negate right operand: `a <op> !b`
    - Negate expression: `!(a <op> b)`

It is important to note that we omit the three logical operators `^`, `|`, and `&` for two reasons. First, the logical exclusive OR operator `^` is, when applied to boolean values, semantically equivalent to the included `!=` operator. This means that `(a^b)` = `(a!=b)`, $\forall$`(a,b)`$\in \{0,1\} \times \{0,1\}$. Hence, the inclusion of both operators would obviously introduce redundancy. Furthermore, the logical operators `|` and `&` produce the same boolean output as the conditional equivalent with the exception that they do not exploit the possible short-circuit evaluation. Short-circuit means that the right hand side of a binary conditional expression is only evaluated if and only if the output is not already defined by the value of the left hand side. As a consequence, the mutants associated with the logical operators can only be killed if the missing short-circuit evaluation leads to an exception or a side effect that changes the internal state. Thus, we ignore the logical operators since this type of fault is also represented by the mutations included in the three categories.

Generally, an expression `a <op> b` can also be expressed in prefix notation, which we use throughout the following definitions:

$$\texttt{a <op> b} \longmapsto op(a,b)$$

Let p be the original program with its corresponding set of input values and parameters π. That is, π denotes the tuple of all input values and parameters necessary to execute p. For a given input domain Π, a mutant m is semantically equivalent to p, written as $m \equiv p$, if and only if it produces the same output as p for all possible input tuples:

Definition 6.1 *Program equivalence*

$$m \equiv p :\Leftrightarrow m(\pi) = p(\pi), \forall \pi \in \Pi$$

With regard to this program equivalence definition, an equivalent mutant of a binary operator *op* that maps input tuples $(a,b) \in \Pi$ to an output $o \in \Omega$ can be defined as:

Definition 6.2 *Equivalent mutant* m_{eq}

$$m_{eq}(a,b) = op(a,b), \forall (a,b) \in \Pi$$

As previously mentioned, easy-to-kill or redundant mutants should be avoided by the mutant generation process. To achieve this goal, the generated mutants should have a minimal impact in order to be hard to detect. We refer to the number of input tuples for which the resulting output of the mutant differs from the original version as impact of the mutant. Moreover, we use the index set I_Π on the input domain Π with a bijective mapping, meaning that I_Π is an enumeration of all input tuples $(a,b)_i \in \Pi$ (e.g., $I_\Pi = \{0, ..., |\Pi| - 1\}$). A minimal impact mutant m^i_{min} for a certain input tuple $(a,b)_i$ is now defined as:

Definition 6.3 *Minimal impact mutant* m^i_{min}

$$\exists! i \in I_\Pi : m^i_{min}(a,b)_i \neq op(a,b)_i$$

This definition indicates that there is only one input tuple for which the outputs of the minimal impact mutant and the binary operator differ. Having all possible minimal impact mutants and a given index set $I_\Sigma \subseteq I_\Pi$, we can now define the sufficient set of mutants M_{suf} as the union of the m^i_{min} mutants in compliance with the uniqueness of the inputs:

Definition 6.4 *Sufficient set of mutants* M_{suf}

$$M_{suf} \coloneqq \{m^i_{min} : i \in I_\Sigma\} \; with \; |M_{suf}| = |I_\Sigma|$$

It is important to note that the sufficient set is defined on the domain $\Sigma \subseteq \Pi$, which contains only the valid, or feasible combinations, of the input domain Π. Moreover, the constraint $|M_{\text{suf}}| = |I_\Sigma|$ is necessary to ensure that exactly one minimal impact mutant per input tuple is included – there might be several minimal impact mutants for a certain tuple. Based on Definition 6.3, the output of a minimal impact mutant differs from the output of the binary operator for exactly one input combination. Now, if this tuple is not an element of the domain Σ, the corresponding mutant is equivalent on Σ:

$$(a,b)_i \notin \Sigma \Rightarrow m^i_{min}(a,b)_j = op(a,b)_j, \forall (a,b)_j \in \Sigma$$

$$\Rightarrow m^i_{min} \text{ is equivalent on } \Sigma \quad \square$$

With regard to the minimal impact mutants, we can now define every additional and non-equivalent mutant as subsumed mutant m_s. Let $I_\Delta(m_s) = \{i \in I_\Pi : m_s(a,b)_i \neq op(a,b)_i\}$. A mutant m_s is subsumed if:

Definition 6.5 *Subsumed mutant* m_s

$$\exists m^i_{min} : m^i_{min}(a,b)_i = m_s(a,b)_i, \forall i \in I_\Delta(m_s)$$

Intuitively, $I_\Delta(m_s)$ represents all input tuples for which the output of the mutant m_s differs from the original version. Since there is a minimal impact mutant for each of those inputs, the mutant is subsumed. Additionally, we define a mutant that produces a wrong output for every feasible input tuple as trivial mutant m_t:

Definition 6.6 *Trivial mutant* m_t

$$m_t(a,b)_i \neq op(a,b)_i, \forall i \in I_\Sigma$$

Based on the given Definition 6.4 for a sufficient set of mutants M_{suf}, Table 6.1 shows the sufficient replacements for the logical connector `&&`. The mutants within this set are minimal impact mutants, in compliance with Definition 6.3, since they change the output of the original clause for exactly one input tuple. The actual changes of the corresponding outputs are highlighted with circles to show that the mutations indeed manifest a minimal impact. Furthermore, all sufficient mutations are disjoint and their union forms the sufficient set M_{suf}, as required by Definition 6.4.

Table 6.1: Sufficient replacements for the logical connector AND.

Literals		**Original clause**	**Sufficient mutations**			
a	b	a && b	false	lhs	rhs	a==b
0	0	0	0	0	0	(1)
0	1	0	0	0	(1)	0
1	0	0	0	(1)	0	0
1	1	1	(0)	1	1	1

Table 6.2: Subsumed mutations for the logical connector AND.

Literals		**Original clause**	**Subsumed mutations**		**Subsumed operator UOI**			
a	b	a && b	a ‖ b	a!=b	true	!(a && b)	!a && b	a && !b
0	0	0	0	0	[1	[1	0	0
0	1	0	[1	[1	1	1	[1]	0
1	0	0	1]	1	1]	1	0	[1
1	1	1	1	0]	1	0]	[0]	0]

Table 6.2 shows the subsumed and trivial mutants for the logical connector `&&`. Besides showing the output of the original expression, the table emphasizes the outputs of the subsumed mutants that differ from the original clause within rectangles. All of the depicted mutants do not fulfill Definition 6.3, since they change the output of more than one input combination, as highlighted by the rectangles, and hence manifest a higher impact. Therefore, none of the mutants can be included in the sufficient set M_{suf}. As already discussed, the sufficient mutations of Table 6.1 collectively cover all possible combinations. Hence, for every subsumed mutant there exists a minimal impact mutant that produces the same wrong output for the corresponding input combination. Therefore, all subsumed mutants fulfill Definition 6.5. For example, the `lhs` operator produces the wrong output for a certain combination of the literals `a` and `b`, that is `a=true` and `b=false`. Additionally, most of the subsumed mutations produce the same wrong output for this combination.

Table 6.3: Sufficient replacements for the logical connector OR.

Literals		**Original clause**	**Sufficient mutations**			
a	b	a ‖ b	a!=b	rhs	lhs	true
0	0	0	0	0	0	(1)
0	1	1	1	1	(0)	1
1	0	1	1	(0)	1	1
1	1	1	(0)	1	1	1

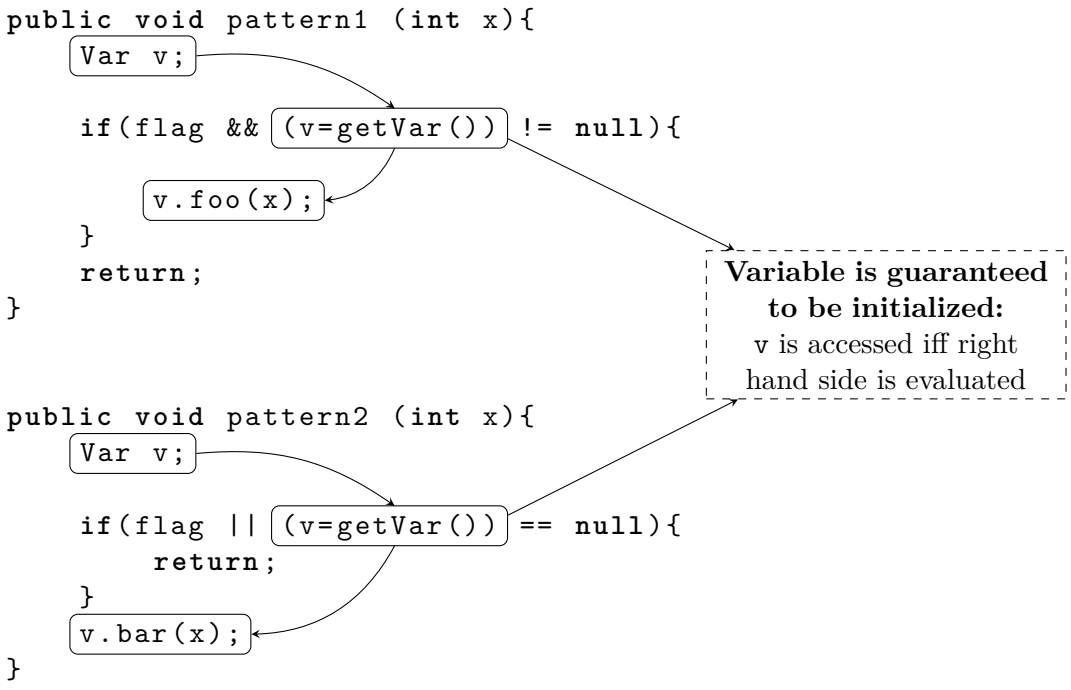
Table 6.4: Subsumed mutations for the logical connector OR.

Literals		**Original clause**	**Subsumed mutations**			**Subsumed operator UOI**		
a	b	a ‖ b	a && b	a==b	false	!(a ‖ b)	!a ‖ b	a ‖ !b
0	0	0	0	[1	0	[1	[1]	[1
0	1	1	[0	0	[0	0	1	0]
1	0	1	0]	0]	0	0	[0]	1
1	1	1	1	1	0]	0]	1	1

Thus, if a test detects the `lhs` mutant, it is also guaranteed that the same test detects all of the other subsumed mutations. According to Definition 6.6, the logical negation of the original clause leads furthermore to a trivial mutant since this mutant produces the wrong output for every input combination, as visualized by the rectangle that covers all four outputs.

With regard to the logical connector `||`, Table 6.3 shows and highlights the minimal impact mutants that form the sufficient set M_{suf}. Additionally, Table 6.4 illustrates the subsumed mutants for this logical connector, where all subsumed mutants again exhibit a higher impact, as visualized by the rectangles.

For both logical connectors, the sufficient set of mutations not only subsumes all of the other mutations but also another entire mutation operator, namely the unary operator



Listing 6.1: Two common patterns with uninitialized local variables exploiting the short-circuit property of the logical connectors && and ||.

insertion (UOI). Hence, this mutation operator should be omitted when mutating conditional expressions due to the manifested redundancy. It has to be pointed out that the depicted subsumption hierarchy only holds for conditional expressions with one logical connector. We do not further investigate composed conditional expressions and leave this matter open for future research.

By employing only the four sufficient out of ten possible mutations, the reduction of the number of mutants generated by the COR mutation operator is approximately 60%. The actual saving depends on the applicability of the mutations and the number of feasible input tuples. The latter one is related to the equivalent mutant problem, which is not further discussed in this chapter. Concerning the applicability of the mutations within Java programs, there are two common patterns that exploit the short-circuit property of the logical connectors `&&` and `||`, as shown in Listing 6.1. Within this listing the short-circuit property is utilized to avoid an unnecessary initialization of local variables.[1] Since the Java compiler strictly requires that every local variable is initialized before use, replacing the logical connector `&&` by `||`, and vice versa, is not valid in the illustrated methods. The application of the special operators `true`, `false`, and `lhs` also leads to invalid mutants that may access an uninitialized local variable. Hence, the actual decrease depends on the

[1]The Java Virtual Machine is a stack machine, and thus a local variable is only stored on the stack once it has been initialized.

Table 6.5: Summary of the applications investigated in the empirical study.

	Application	Version	LOC*	Relational operators	Conditional operators	Tests
trove	GNU Trove	3.0.2	116,750	7,937	1,945	544
chart	jFreeChart	1.0.13	91,174	2,762	781	2,130
itext	iText	5.0.6	76,229	5,293	1,760	75
math	Commons Math	2.1	39,991	3,233	428	2,169
time	Joda-Time	2.0	27,139	1,324	364	3,855
lang	Commons Lang	3.0.1	19,495	1,618	695	2,039
jdom	JDOM	2beta4	15,163	1,023	216	1,723
jaxen	Jaxen	1.1.3	12,440	815	159	699
io	Commons IO	2.0.1	7,908	345	139	309
num4j	Numerics4j	1.3	3,647	312	133	218
total			409,936	24,662	6,620	13,761

*Physical lines of code as reported by sloccount (non-comment and non-blank lines)

number of occurrences of the depicted pattern. Moreover, the total reduction of the entire mutation set depends on the ratio of COR mutants to all generated mutants.

6.3 Empirical Evaluation

To examine both the frequency and the effect of redundant mutants, we conducted an empirical study with ten open-source applications. Table 6.5 summarizes the investigated applications, showing how they differ in size, complexity, and operation purpose. Since the study focuses on the reduction of mutants associated with applying the COR and ROR mutation operators, the table also gives the counts for the occurrences of relational and conditional operators, in addition to the number of files, tests, and lines of code. The number of tests, depicted in the last column of Table 6.5, represents the quantity of existing unit tests that are provided and released with the corresponding application.

Throughout the empirical study, we employ Major, the developed compiler-integrated research tool for the mutation analysis of Java programs, to mutate the applications and to perform the mutation analysis process. Major also provides all relevant data about

the number of generated mutants and the necessary runtime for the mutation analysis, thus enabling an investigation of the following four research questions:

Q1: What is the ratio of the number of mutants generated by the COR and ROR operators to the number of mutants generated by applying all operators?

Q2: Are conditional expressions with only one logical connector, like those studied in Section 6.2, predominant in real-world applications?

Q3: What is the actual savings in the runtime of mutation analysis due to the use of the reduced set of mutants?

Q4: How does the elimination of redundant mutants affect the overall mutation score?

6.3.1 The frequency of the COR and ROR mutants

To answer the first research question, we determined the number of mutants generated by applying the COR and ROR operators with all of the possible replacements, as defined by Namin et al. [2008]. We also ascertained the number of mutants that can be generated by using all of the operators, including COR and ROR. Figure 6.1 visualizes the ratio of mutants associated with the COR and ROR operators (the dark gray and black bars) to the number of mutants generated by applying all mutation operators (light gray bar). Ranging from 30.9% for *math* to 63.6% for *trove*, the number of mutants generated by only applying the COR and ROR operators is a substantial portion of all the induced mutants. With a mean value of 45.1%, this range suggests that there is a notable potential for effectiveness and efficiency improvements through the removal of the redundant mutants associated with COR and ROR.

6.3.2 The number of connectors in conditional expressions

As stated in Section 6.2, we can only guarantee that the reduced set of mutants generated by the COR operator is sufficient and redundancy-free for conditional expressions with one logical connector. Therefore, in order to ascertain the benefit of this partial solution, we calculated the ratio of conditional expressions with one connector to the remaining number of conditional expressions. For each application, Figure 6.2 illustrates the distribution of the number of logical connections in the conditional expressions. With a mean value of 80.1% across the ten studied programs and a range between 63.3% for *num4j* and 85.9% for *math*, the number of conditional expressions with only one connector is predominant for all applications. Thus, for almost 80% of the conditional expressions, the suggested subset of replacements, as given in Section 6.2, provides a sufficient and redundancy-free set of mutants.

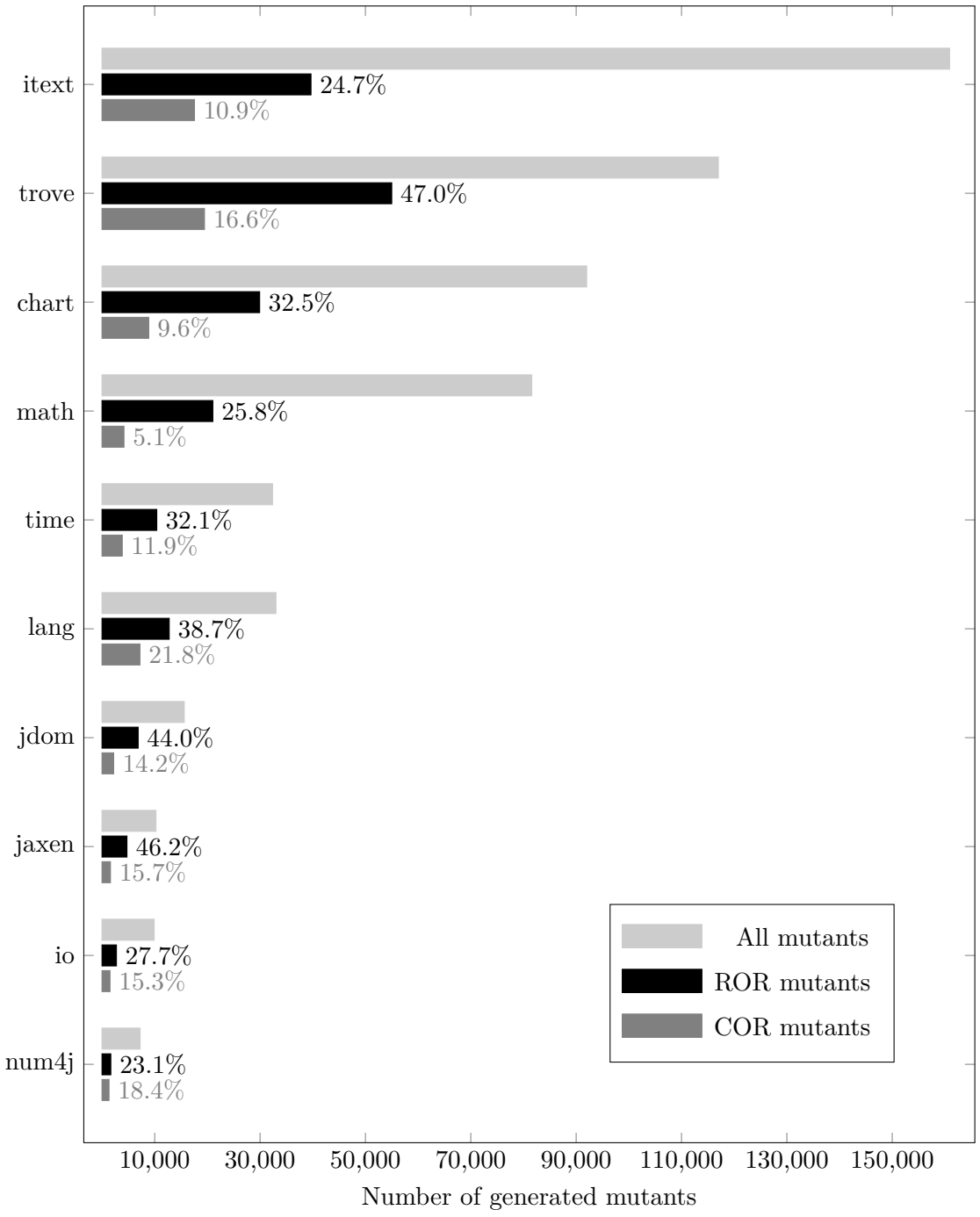


Figure 6.1: Ratio of the number of COR and ROR mutants to the number of all generated mutants for the investigated applications.

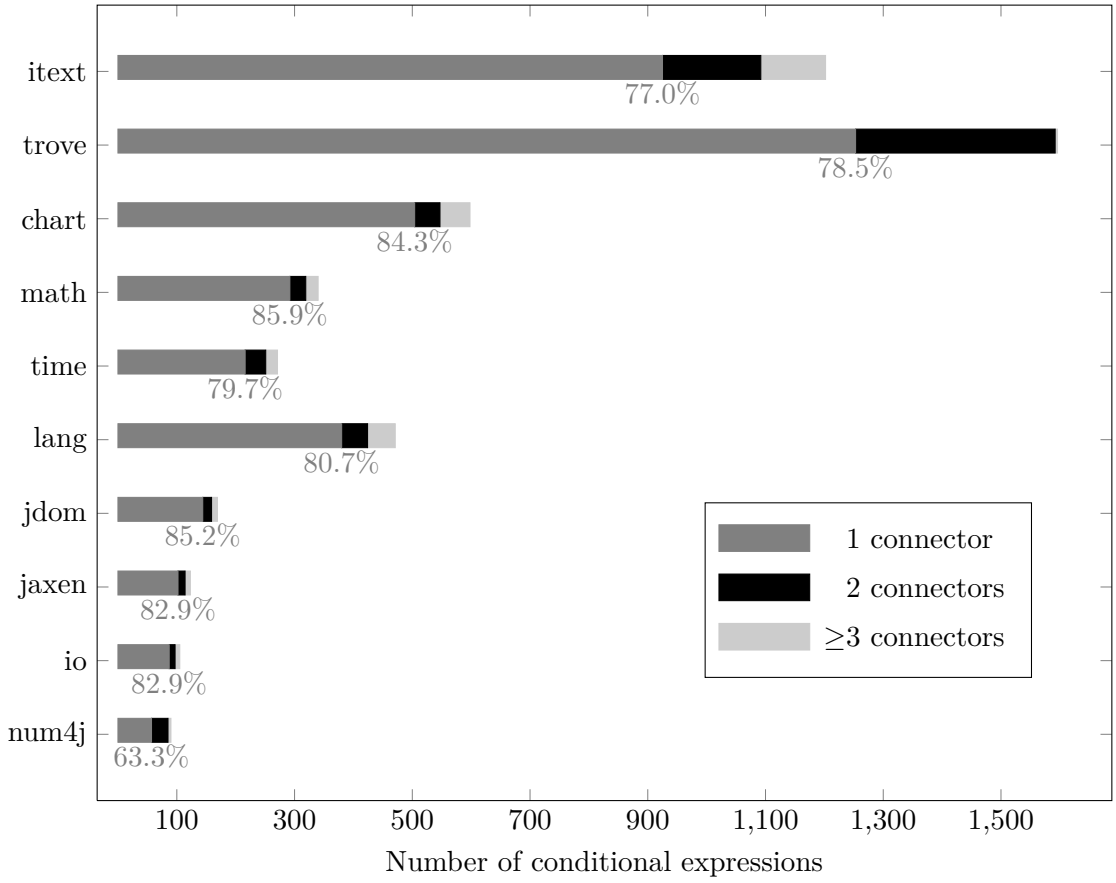


Figure 6.2: Distribution of the number of logical connectors in conditional expressions for the investigated applications.

6.3.3 Decreasing the runtime of the mutation analysis

The smaller subset of replacements for the COR and ROR operators means that fewer mutants have to be generated and hence the number of necessary executions during the mutation analysis process is also reduced. With regard to the reduction of mutants and the decrease in runtime, we distinguish between generated and covered mutants, with covered meaning that a mutant is reached and executed but not necessarily killed. Hence, the mutation coverage is a necessary but not sufficient condition to kill a mutant. Table 6.6 shows the decrease in the number of generated and covered mutants according to the following two sets of mutations:

- M_{all}: Set of mutants generated by applying all mutation operators with all valid alternatives for replacement operators.
- M_{red}: Reduced set of mutants generated by all available mutation operators but only with sufficient replacements for the COR and ROR operators.

Table 6.6: Decrease in the number of generated and covered mutants.

	Generated mutants		Covered mutants	
	M_{all}	M_{red}	M_{all}	M_{red}
itext	160,891	126,781 (-21.2%)	25,650	19,541 (-23.8%)
trove	116,991	72,959 (-37.6%)	9,494	6,137 (-35.4%)
chart	92,000	68,503 (-25.5%)	50,735	36,298 (-28.5%)
math	81,577	66,787 (-18.1%)	74,327	60,148 (-19.1%)
time	32,380	23,781 (-26.6%)	27,661	19,577 (-29.2%)
lang	33,023	21,056 (-36.2%)	32,034	20,196 (-37.0%)
jdom	15,616	10,800 (-30.8%)	14,998	10,266 (-31.6%)
jaxen	10,247	7,132 (-30.4%)	6,675	4,679 (-29.9%)
io	9,901	7,319 (-26.1%)	5,166	4,255 (-17.6%)
num4j	7,234	5,437 (-24.8%)	7,007	5,243 (-25.2%)
total	559,860	410,555 (-26.7%)	253,747	186,340 (-26.6%)

The reduction of the mutations associated with the COR and ROR operators significantly affects the number of generated mutants, even when applying all available mutation operators. Depending on the ratio of the COR and ROR mutants to all other mutants, the decrease ranges between 18.1% for *math* and 37.6% for *trove*, as shown in Table 6.6. Concerning the covered mutants, also depicted in the last two columns of Table 6.6, the decrease in the number of covered mutants is comparable to the decrease in the number of generated mutants for all applications except *io* for two reasons. Apparently, the test suite for *io* exhibits a low mutation coverage ratio of approximately 50%. In addition to this relatively low ratio, a lot of COR and ROR mutants are not covered and hence, the effect of reducing the replacements for COR and ROR is more noticeable for the number of generated mutants than for the number of covered mutants.

In addition to calculating the reduction in the number of generated and covered mutants, we also determined the actual improvement in the runtime while exploiting two runtime optimizations. On the one hand, we do not analyze mutants that are not covered since they cannot be killed and on the other hand we do not further investigate a mutant once it has been killed. Moreover, some mutants lead to infinite loops, for instance the ones derived from mutating loop conditions. To prevent the mutation analysis process from getting stuck, these infinite loops have to be identified. Due to the fact that it is undecidable in general whether or not a mutant produces an infinite loop, we apply a timeout heuristic

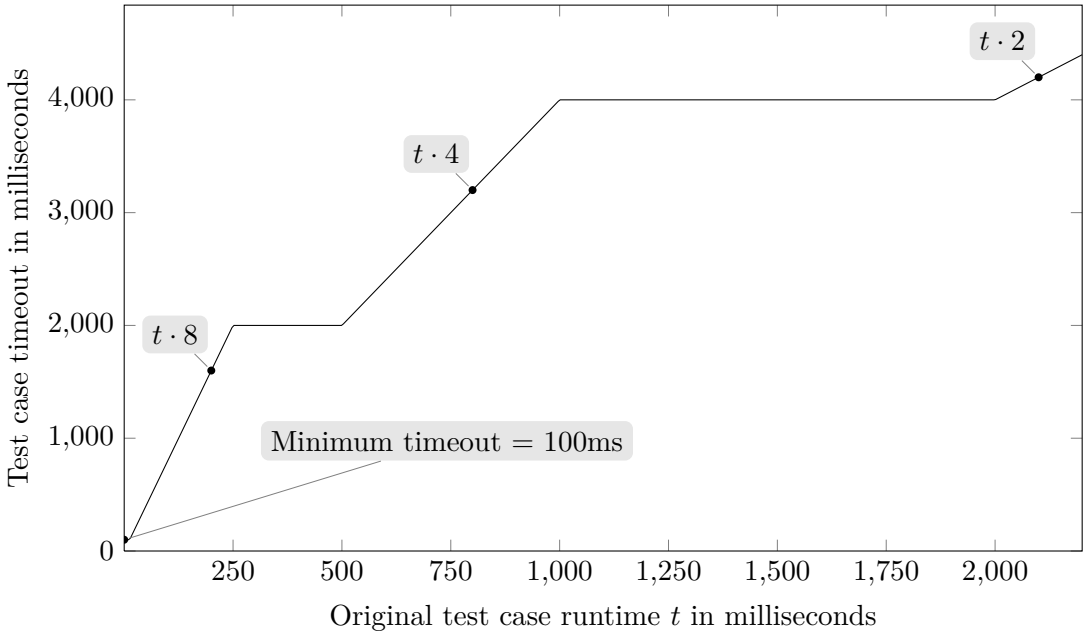


Figure 6.3: Function used to determine the timeout for mutants (base timeout factor=8).

that depends on the runtime of the unmutated version. Figure 6.3 visualizes the piecewise-defined function used to determine the timeout for a certain test case. This function uses a base timeout factor that decreases over time, down to a minimum of two. Because the runtime of a short test is more likely to be influenced by small delays inherent in the system, we use a considerably larger timeout factor for short-running tests. Moreover, the lower bound for the timeout is 100 milliseconds to compensate for any measuring inaccuracy when determining the timeout for test cases that exhibit an extremely small runtime. It is important to note that most of the unit tests provided with the investigated applications exhibit a small runtime below 50 milliseconds. Since mutants are usually covered by several unit tests, it is also crucial to state that the timeout does not depend on the mutant but rather on the test case that covers it.

In order to estimate the runtime of the original version on which the heuristic shown in Figure 6.3 is based on, we ascertain the maximum runtime of two independent runs of the original version. Reconsidering the characteristics of conditional mutation, which is described in Chapter 4, this runtime is indeed an adequate approximation since for every expression or statement that has been mutated with conditional mutation, the original version is always the innermost node within a nested expression or statement, respectively. Listing 6.2 provides an example for an expression and statement mutated with conditional mutation, where `M_NO` denotes the mutant identifier necessary to trigger a certain mutant. This example clearly shows that the execution of the original version is associated with the highest runtime overhead in terms of evaluating all the injected conditional expressions that enable conditional mutation. It is important to note that our heuristic, like all heuristics, for infinite loop detection may produce false-positive results. Nevertheless, the determined timeout only leads to an interruption of a mutant if its runtime is significantly

prolonged. In this case, interruption means that the execution of a test case that analyzes a certain mutant is stopped and the corresponding mutant is then marked as being killed. To the best of our knowledge, other mutation testing tools such as Javalanche [Schuler et al., 2009] or MuJava [Ma et al., 2006] use a timeout that is constant for all test cases. With regard to the individual test cases of which the runtimes differ by several orders of magnitude, we judge that a variable timeout is more sensitive.

```
1  public int eval(int x){
2      int a = 3, b = 1, y;
3
4      y = (M_NO==1)? a - x:
5          (M_NO==2)? a * x:
6                     a + x;
7
8      if(M_NO==3){
9          y -= b;
10     }else{
11         y += b;
12     }
13     return y;
14 }
```

Original version: All conditional expressions have to be executed to reach this point

Listing 6.2: Expressions and statements mutated with conditional mutation.

Table 6.7 furnishes the execution time of a mutation analysis process that uses Major to calculate a mutation score for each application's test suite. With a reduction in runtime of up to 37% for *lang* and a minimum of 11% for *jaxen*, the results demonstrate a significant speed-up for all of the applications. Yet, the observed improvement in the runtime depends on the distribution of the COR and ROR within the application and the runtimes of the tests that do not cover these mutants. For instance, the test suites for *math* and *jaxen* contain a few long-running tests that cover many mutants but only a few COR and ROR mutations. Since the runtime of these tests is a considerable proportion of the total runtime, the removal of the redundant COR and ROR mutants only yields a modest decrease in the cost of mutation analysis for these applications.

6.3.4 Increasing the precision of the mutation score

Since redundant mutants lead to an imprecision in the mutation score, we also calculate this score for both the generated and covered mutants for all of the investigated applications. Table 6.8 gives the mutation score for the generated mutants, with M_{all} and M_{red} again denoting the mutation sets described in Section 6.3.3. The reduced sets result in a decrease of the mutation score of more than 20% for programs like *chart* and an average

Table 6.7: Decrease in the runtime of the mutation analysis.

	Total runtime*		Covered mutants		Tests
	M_{all}	M_{red}	M_{all}	M_{red}	
itext	618.0	426.6 (-31.0%)	25,650	19,541 (-23.8%)	75
trove	47.7	37.7 (-21.0%)	9,494	6,137 (-35.4%)	544
chart	783.4	582.1 (-25.7%)	50,735	36,298 (-28.5%)	2,130
math	536.8	473.0 (-11.9%)	74,327	60,148 (-19.1%)	2,169
time	433.4	340.1 (-21.5%)	27,661	19,577 (-29.2%)	3,855
lang	38.9	24.5 (-37.0%)	32,034	20,196 (-37.0%)	2,039
jdom	179.3	135.6 (-24.4%)	14,998	10,266 (-31.6%)	1,723
jaxen	482.9	430.9 (-10.8%)	6,675	4,679 (-29.9%)	699
io	7.4	5.0 (-32.4%)	5,166	4,255 (-17.6%)	309
num4j	2.8	1.8 (-35.7%)	7,007	5,243 (-25.2%)	218
avg		-25.1%		-27.7%	

*Total runtime reported in minutes

decrease of almost 7%. Unless the redundant mutants are removed, the mutation score is overestimated for all applications except *io*. For this application, the corresponding test suite only covers 33% of the generated COR and ROR mutants. Thus, removing the redundant mutants affects the number of generated mutants more significantly than the number of killed mutants, leading to a 10% increase in the mutation score.

Table 6.8 also shows how the removal of the redundant mutants affects the mutation score that is calculated for the number of covered mutants. Once again, there is a notable 7% average decrease in the mutation score that ranges between 1% for *io* and 18% for *chart*. Overall, redundant mutants tend to overestimate the mutation score for the applications in this empirical study, thus causing this metric to become a less accurate assessment of test suite quality.

To better assess the improved mutation score, we relate it to mutation coverage and two code coverage criteria, namely line[2] and branch coverage. In addition to the mutation score, Table 6.9 also gives both the ratio of covered mutants (i.e., the number of mutants reached and executed) and the ratio of covered lines and branches. While the mutation and code coverage results are similar, the mutation score is considerably lower for all applications except for *io*. This clearly indicates that the test oracles used in *io* are suitable

[2]The term line coverage is used in Cobertura, but it is also known as statement coverage [Zhu et al., 1997].

Table 6.8: Divergence of the mutation score with regard to the generated and covered mutants.

	Generated mutants			Covered mutants		
	M_{all}	M_{red}		M_{all}	M_{red}	
itext	0.04	0.04	(-13.1%)	0.28	0.25	(-10.2%)
trove	0.05	0.05	(-8.5%)	0.66	0.58	(-11.7%)
chart	0.24	0.19	(-20.7%)	0.44	0.36	(-17.5%)
math	0.76	0.73	(-3.0%)	0.83	0.82	(-1.8%)
time	0.76	0.72	(-5.4%)	0.89	0.88	(-1.9%)
lang	0.77	0.71	(-7.4%)	0.79	0.74	(-6.3%)
jdom	0.80	0.76	(-4.1%)	0.83	0.80	(-3.1%)
jaxen	0.36	0.31	(-14.6%)	0.56	0.47	(-15.2%)
io	0.41	0.45	(10.5%)	0.78	0.78	(-0.8%)
num4j	0.66	0.64	(-3.1%)	0.69	0.67	(-2.7%)
avg			-6.9%			-7.1%

since they kill a substantial number of covered mutants. Yet, the corresponding test suite does not adequately cover the entire project. In contrast when considering a program such as *num4j*, the mutation coverage yields a satisfying result but the mutation score is significantly lower. This could be caused either by a large number of equivalent mutants or by a weakness of the applied test oracles. Since we primarily focus on redundant mutants in this chapter, we leave the investigation of equivalent mutants in the generated mutation sets open for future work. Overall, the results confirm that the mutation score has to be calculated as accurately as is possible to adequately reflect the quality of the test inputs as well as test oracles (cf. [Andrews et al., 2006]). An accurate mutation score is of particular importance if it is further used, for instance for test data generation [Fraser and Zeller, 2012]. The elimination of the redundant mutants therefore improves the expressiveness of the mutation score.

Besides calculating the scores of all three criteria, we also measure the necessary runtime to generate those values. With regard to the mutation coverage and score, we again apply both mutation sets M_{all} and M_{red} since the number of mutants may have an effect on the runtime. Because of the fact that the code coverage is mutant independent, there is only one runtime for gathering the code coverage. Table 6.10 shows the runtimes necessary to determine the individual scores. Intuitively, the runtime for the complete mutation

Table 6.9: Comparison of the mutation score with mutation coverage and code coverage.

	Mutation score		**Mutation coverage**		**Code coverage***	
	M_{all}	M_{red}	M_{all}	M_{red}	**line**	**branch**
itext	4.4%	3.8%	15.9%	15.4%	20.0%	11.0%
trove	5.3%	4.9%	8.1%	8.4%	7.0%	6.0%
chart	24.2%	19.2%	55.1%	53.0%	57.0%	46.0%
math	75.7%	73.5%	91.1%	90.1%	88.0%	85.0%
time	76.2%	72.1%	85.4%	82.3%	90.0%	80.0%
lang	76.6%	71.0%	97.0%	95.9%	93.0%	90.0%
jdom	79.5%	76.3%	96.0%	95.1%	95.0%	94.0%
jaxen	36.2%	30.9%	65.1%	65.6%	78.0%	55.0%
io	40.8%	45.1%	52.2%	58.1%	39.0%	29.0%
num4j	66.5%	64.4%	96.9%	96.4%	97.0%	96.0%

*As reported by Cobertura [Cobertura, 2010]

analysis process is several orders of magnitude larger than the runtime for the mutation and code coverage. This is in particular due to the iterative executions of the test suite when analyzing all (covered) mutants to determine the mutation score. In contrast, the mutation and code coverage can be ascertained with only one execution of the test suite. Moreover, it turns out that the mutation coverage, which is additionally an upper bound for the mutation score, can be calculated faster than the code coverage for nine out of ten applications, regardless of which mutant set is applied.

Since the mutation coverage provides a coverage criterion that is as effective as the used code coverage criteria, but at lower costs in terms of runtime, one could exclusively employ mutation analysis. This, however, may depend on the employed coverage tool as well as on the analyzed application. Therefore, additional studies are necessary to confirm our results – but we leave further investigations open for future work.

As for every empirical study, it is important to examine the threats to the validity of the reported results. The chosen subset of sufficient mutation operators could be a threat to internal validity. Different or additional operators may affect both the number and the ratio of the generated mutants. However, the operators employed in the presented study are frequently used in the literature and therefore provide comparable results [Namin et al., 2008; Offutt and Untch, 2000]. In addition to the mutation operators, the chosen function to determine the timeout for the mutants could be another threat to internal validity. A

Table 6.10: Comparison of the runtimes for calculating code coverage and for determining the mutation score and mutation coverage.

	Mutation score		**Mutation coverage**		**Code coverage**
	M_{all}	M_{red}	M_{all}	M_{red}	
itext	618.0	426.6	0.3	0.2	0.4
trove	47.7	37.7	0.3	0.3	0.2
chart	783.4	582.1	0.9	0.9	11.7
math	536.8	473.0	1.3	1.2	4.7
time	433.4	340.1	0.5	0.5	4.2
lang	38.9	24.5	0.3	0.2	0.8
jdom	179.3	135.6	0.5	0.4	1.0
jaxen	482.9	430.9	0.2	0.2	0.4
io	7.4	5.0	0.3	0.3	0.5
num4j	2.8	1.8	0.1	0.1	0.1

*Runtimes reported in minutes

timeout factor that is too small would result in a noticeable number of false-positives and therefore underestimate the runtime improvements. In contrast, a timeout factor that is too large slightly overestimates the runtime improvements since the set with all mutants leads to more timeouts than the reduced set. Nevertheless, the first aspect of the timeout factor is much more severe and we controlled this threat by manually analyzing samples of mutants that ran into a timeout and also by comparing the numbers of timeouts for different timeout factors. The analyzed mutants exhibited syntactic changes that caused the test suite to run exceptionally long or infinitely. The representativeness of the selected applications might be a potential threat to external validity. Thus, the presented results may be different for other programs. The analyzed applications, nevertheless, vary in terms of their size and operation purpose. Therefore, we judge that the reported results are meaningful.

6.4 Related Work

As previously mentioned in Chapter 3, employing all available mutation operators with all valid replacements results in a significant number of mutants, especially for large software systems. In addition to the runtime overhead caused by the many mutants, there are also

redundant and trivial mutants that prolong the runtime of the mutation analysis and, moreover, may misrepresent the mutation score. Within this section, we now discusses related work that has also focused on efficiency improvements for mutation analysis.

In an attempt to reduce the computational costs, different selective and sampling-based approaches have been proposed in the literature (e.g., [Jia and Harman, 2011], [Offutt and Untch, 2000]). These techniques reduce the quantity of mutants either by decreasing the number of operators or by selecting only a subset of the generated mutants. Yet, all of these approaches view the mutation operators, in their originally defined form, as atomic. Hence, there are still redundancies within the selected subset that affect both the runtime and the mutation score.

Kaminski et al. [2011] investigated the ROR operator in detail and showed that a subset of three out of seven valid replacements is sufficient for this operator. They additionally claimed, without further investigation or evidence, that this reduction would improve efficiency. Similar to our focus on conditional and relational operators, [Tai, 1996] developed a theory for testing the predicates in conditional logic statements.

In connection with our method that avoids redundant mutants and minimizes the impact of mutations, higher order mutation aims at generating fewer, but more subtle mutants [Jia and Harman, 2009]. Mutants created by means of the combination of two first order mutants are referred to as second order mutants. Accordingly, higher order mutation generally denotes the combination of two or more first order mutants. Jia and Harman [2009] showed the existence of higher order mutants that are harder to kill than the first order mutants out of which they were created. Nevertheless, the computational costs for higher order mutation are significantly greater because of the combinatorial explosion.

Apart from redundant mutants, the equivalent mutant problem is another crucial consideration in mutation testing. Equivalent mutants are harmful to the runtime of the mutation analysis process since they cannot be detected by any test. Additionally, employing a set of mutants that includes equivalent mutants results in an underestimation of the mutation score. Approaches that try to alleviate the equivalent mutant problem can be divided into two categories. On the one hand, there are techniques focusing on the detection of equivalent mutants (e.g., [Offutt and Craft, 1994; Hierons et al., 1999]). On the other hand, approaches exist for reducing the number of equivalent mutants during the mutant generation process (e.g., [Offutt, 1992]).

6.5 Summary

This chapter has investigated how redundant mutants affect the effectiveness and efficiency of mutation analysis. Focusing on three well-known mutation operators, namely the relational operator replacement (ROR), the conditional operator replacement (COR), and the unary operator insertion (UOI), this chapter makes several contributions. First, it develops a subsumption hierarchy for COR and UOI and reveals a sufficient set of replacements. Using this sufficient set, in conjunction with the reduced set of ROR mutants,

it reports on a study that empirically demonstrates how redundant mutants affect both the mutation score and the runtime of the mutation analysis process.

After determining how prevalent relational and conditional operators are in real-world applications, this chapter examines the ratio of the number of mutants generated by the COR and ROR mutation operators to the total number of mutants. With a mean value of 45.1% and a range from 30.9 to 63.6%, the high percentage of COR and ROR mutants clearly reveals the potential for improving mutation analysis by focusing on the mutants produced by these operators. The experiments also show that employing the sufficient replacements for the COR and ROR operators leads to a commensurate drop in the number of generated mutants that ranges from 18.1 to 37.6%.

Moreover, reducing the number of generated mutants leads to a decrease in the runtime of the mutation analysis process that is between 10.8 and 37%. Finally, the empirical results show that, depending on the application, improving the precision of the mutation score can lead to a value that is greater than or less than the score resulting from the use of the original set of mutants. For nine out of ten applications, using the reduced set of mutants yields a reduction in the mutation score ranging from 3.0 to 20.7%. Overall, by applying the mutant set that includes the redundant mutations, the mutation score is overestimated by 7% on average.

Furthermore with regard to assessing the quality of the investigated test suites, the mutation score is compared with the ratios derived from code coverage, much more meaningful but the costs of determining the mutation score are considerably higher. However, the study reveals that the mutation coverage, which is an upper bound for the mutation score, possesses a similar effectiveness as the code coverage. Since the necessary runtime to determine the mutation coverage ratio is lower than the runtime for the code coverage analysis, it is advisable to use mutation coverage instead.

Chapter 7

Test Suite Prioritization

The content of this chapter has been published in [Just et al., 2012b]

The previous chapters presented a new approach to mutant generation in conjunction with its corresponding implementation and basic research on non-redundant mutation operators. This chapter builds upon these results and makes several contributions to significantly improve the efficiency and scalability of mutation analysis. Since the isolated use of non-redundant mutation operators does not ensure that mutation analysis is efficient and scalable, especially with regard to large software systems, this chapter presents and empirically evaluates an optimized workflow for mutation analysis. This workflow exploits the redundancies and runtime differences of test cases to reorder and split the corresponding test suite.

7.1 Introduction

Compared with testing techniques that rely on various code coverage criteria, mutation analysis is rather expensive because of the fact that many mutated versions of the analyzed program have to be executed. With regard to the costs of mutation analysis, several approaches have been proposed that try to either reduce the number of generated mutants or to speed-up the analysis process (cf. [Offutt and Untch, 2000; Jia and Harman, 2011]). However, the runtime of mutation analysis on large and complex software systems is still quite long, sometimes even prohibitively so.

Addressing the challenge of applying mutation analysis to real-world programs, this chapter investigates the potential for efficiency improvements by employing test suite prioritization. For this purpose, this chapter first investigates existing test suites to identify redundancies, runtime differences, and room for improvements. Then, this chapter develops an optimized workflow for mutation analysis based on the given observations. Besides motivating and describing, this chapter also evaluates this workflow that notably reduces the runtime. Overall, this chapter makes the following contributions:

- An investigation of the test suite characteristics of real-world applications that enable runtime improvements.
- A presentation and visualization of an optimized mutation analysis workflow that is based on reordering and splitting test suites to exploit the identified redundancies and runtime differences of test cases.
- An empirical study that evaluates the presented approach on ten real-world applications. By utilizing non-redundant mutation operators, the optimized workflow reduces the total runtime by 30% on average for a set of 410,000 mutants in total.

In the remainder of this chapter, Section 7.2 briefly reviews the necessary background on non-redundant mutation operators to ensure that the chapter is self-contained. Next, Section 7.3 investigates characteristics of existing test suites for real-world applications and Section 7.4 presents an approach for significant efficiency improvements by exploiting mutation coverage and test runtime information. Thereafter, Section 7.5 empirically evaluates the approach and Section 7.6 describes related work. Finally, Section 7.7 summarizes this chapter.

7.2 Non-Redundant Mutation Operators

Subsumed mutants lead to redundancies in the generated set of mutants and thus affect the efficiency of mutation analysis and misrepresent the mutation score. Hence, removing such subsumed mutants yields immediate savings in terms of runtime. Chapter 6 has investigated redundancies in the mutation operators for replacing conditional operators and for the insertion of boolean unary operators. Besides showing that these operators introduce redundancies, it also provides a non-redundant-free definition for sufficient replacements. Moreover, Kaminski et al. [2011] investigated redundancies in the mutated relational operators and proposed a non-redundant version of replacements by means of a subsumption hierarchy. This section summarizes the essential results since this chapter employs the non-redundant versions of the mutation operators in the subsequent analyses.

Non-Redundant ROR Operator

The ROR mutation operator replaces all binary relational operators (i.e., `==,!=,<,<=,>,>=`) with both all valid alternatives and the special operators `true` and `false`. The following three replacements per operator are sufficient when mutating relational operators:

- `<` $\longmapsto$ `<=,!=,false`
- `>` $\longmapsto$ `>=,!=,false`
- `==` $\longmapsto$ `<=,>=,true`
- `<=` $\longmapsto$ `<,==,true`
- `>=` $\longmapsto$ `>,==,true`
- `!=` $\longmapsto$ `<,>,true`

As a consequence, the application of the sufficient set of three out of seven possible mutations yields a reduction of 57% for the ROR mutants.

Non-Redundant COR Operator

Generally, the COR mutation operator replaces an expression `a <op> b`, where `a` and `b` denote boolean expressions or literals and `<op>` is one of the logical connectors `&&` or `||`. With regard to binary conditional operators, valid mutations belong to one of the following three categories:

1. Apply conditional operator (COR)
    - Apply logical connector `AND`: `a && b`
    - Apply logical connector `OR`: `a || b`
    - Apply equivalence operator: `a == b`
    - Apply exclusive OR operator: `a != b`
2. Apply special operator (COR)
    - Evaluate to left hand side: `lhs`
    - Evaluate to right hand side: `rhs`
    - Always evaluate to true: `true`
    - Always evaluate to false: `false`
3. Insert unary boolean operator (UOI)
    - Negate left operand: `!a <op> b`
    - Negate right operand : `a <op> !b`
    - Negate expression: `!(a <op> b)`

By employing only the following sufficient four out of ten possible mutations, the reduction of the number of mutants generated for the conditional operators is 60%:

- `&&` $\longmapsto$ `lhs, rhs, ==, false`
- `||` $\longmapsto$ `lhs, rhs, !=, true`

Interestingly, the entire UOI operator is subsumed by the sufficient set of COR mutants, and hence also redundant.

Table 7.1: Summary of the applications investigated in the empirical study.

	Application	Version	LOC*	Mutants	Test LOC*
trove	GNU Trove	3.0.2	116,750	116,991	13,279
chart	jFreeChart	1.0.13	91,174	92,000	48,026
itext	iText	5.0.6	76,229	160,891	1,612
math	Commons Math	2.1	39,991	81,577	41,906
time	Joda-Time	2.0	27,139	32,380	51,901
lang	Commons Lang	3.0.1	19,495	33,065	32,699
jdom	JDOM	2beta4	15,163	15,616	22,194
jaxen	Jaxen	1.1.3	12,440	10,247	8,514
io	Commons IO	2.0.1	7,908	9,901	13,608
num4j	Numerics4j	1.3	3,647	7,234	5,273
total			409,936	559,902	239,012

*Physical lines of code as reported by sloccount (non-comment and non-blank lines)

7.3 Efficient and Scalable Mutation Analysis

Since redundant mutants are harmful to the efficiency of mutation analysis and they furthermore misrepresent the mutation score, as shown in Chapter 6, it is strongly advisable to apply only the non-redundant sets of the ROR and COR mutants. Therefore, the approach to efficient mutation analysis presented in this chapter always uses the non-redundant operators in order not to overestimate its efficiency improvements.

By focusing on the ten applications given in Table 7.1, this section makes several motivating observations concerning mutation coverage, the differences in test runtime, and the redundancies in a test suite. Ultimately, these insights lead to an optimized workflow that performs test suite prioritization and splitting. It is important to note that the depicted data about the test size in Table 7.1 reflects the characteristics of the existing JUnit test suites provided and released with the corresponding application.

By employing the non-redundant versions of the ROR and COR operators, we can decrease the number of generated mutants by almost 27% in total for the investigated applications, as shown in Table 7.2. However, the remaining number of mutants, which is 410,000 for

Table 7.2: Decrease in the number of generated mutants.

	All mutants	Reduced Set	Decrease
trove	116,991	72,959	-37.6%
chart	92,000	68,519	-25.5%
itext	160,891	126,781	-21.2%
math	81,577	66,787	-18.1%
time	32,380	23,781	-26.6%
lang	33,065	21,074	-36.3%
jdom	15,616	10,800	-30.8%
jaxen	10,247	7,132	-30.4%
io	9,901	7,319	-26.1%
num4j	7,234	5,437	-24.8%
total	559,902	410,589	-26.7%

all of the investigated applications, is still substantial. Therefore, we focus on further runtime improvements that do not rely on the reduction of mutants.

7.3.1 Mutation Coverage

Recalling the hierarchy of the three conditions that have to be fulfilled to ultimately to kill a mutant, as proposed by Voas [1992]:

1. Execution: The mutated code must be covered, meaning that it has to be reached and executed.
2. Infection: The execution of the faulty code segment has to change the internal state of the program.
3. Propagation: The infected internal state must be propagated to the output in order to be detectable.

While the first two conditions are necessary, the last one is also sufficient to kill a mutant. Besides, the last condition can be generalized to oracles that are not output-based. In this case, the infected internal state has to be propagated to a state that is observable by the test oracle (cf. [Fraser and Zeller, 2012; Harman et al., 2011]). An example

for such an observable state is the violation of invariants or contracts in general. Since the first condition is necessary, it implies that if a mutant is not covered it cannot be killed. As a consequence, mutants that are not covered can be excluded and marked alive without execution. Utilizing this implication can significantly reduce the number of executions, especially if a test suite exhibits poor mutation coverage. Nevertheless, it is important to note that the mutation coverage has to be determined at runtime. Employing a code coverage tool for this purpose and mapping the covered statements and branches to mutants is feasible, but rather laborious, since code coverage tools are not designed for this purpose.

More advanced mutation analysis systems that encode all mutants within the original program can provide the mutation coverage information at runtime by means of additional code instrumentation (e.g., Javalanche [Schuler and Zeller, 2009], EvoSuite [Fraser and Arcuri, 2011], and MAJOR). This chapter relies on MAJOR, the developed mutation system presented in Chapter 5, which gathers the mutation coverage information efficiently at runtime. For performance reasons, it only records the mutation coverage if and only if the original, which means the unmutated version of the SUT, is executed. Due to the overhead incurred by determining the mutation coverage, this feature is disabled during the execution of mutants.

For all of the investigated applications, Table 7.3 shows the number of generated and covered mutants. In addition to these numbers, it gives the corresponding ratios of covered-to-generated mutants. The results exhibit a notable divergence between the applications, ranging between 8.2% for *trove* and 94.7% for *num4j*, with a total mutation coverage of 43.5% for all analyzed applications. The reason for the extremely low mutation coverage for the *trove* application is that it contains a lot of generated source files, of which only a minor proportion is tested by the test suite. Overall, the coverage results clearly indicate the considerable potential for runtime improvements by excluding uncovered mutants from the mutation analysis (cf. [Schuler et al., 2009]).

Due to the necessary conditions to detect a fault, a test suite cannot kill a mutant that it does not reach and execute. Hence, a mutation analysis process that does not employ coverage information would have to execute the entire test suite for all of the uncovered mutants. In order to estimate the overhead originating from uncovered mutants, that is running the mutation analysis without coverage information, we use the total runtime of the existing JUnit test suites and the number of mutants that are not covered by the corresponding test suite. Table 7.4 shows the corresponding results for all applications. In this table, the overhead of the first three applications, which is more than ten days, is huge because of the fact that an enormous number of mutants is not covered. Even though the mutation coverage for the *math* application yields an acceptable ratio of 88.6%, the overhead of 160 hours caused by the long test runtime is still prohibitive. It is important to state that the total runtime of a mutation analysis process would include the estimated overhead and additionally the runtime necessary to analyze all covered mutants. Thus, mutation analysis for large real-world applications is not feasible without mutation coverage information. Therefore, we always exploit this mutation coverage information in the subsequent analyses.

Table 7.3: Ratio of covered to generated mutants.

	Generated mutants	**Covered mutants**	
trove	72,959	6,016	(8.2%)
chart	68,519	35,659	(52.0%)
itext	126,781	16,521	(13.0%)
math	66,787	59,195	(88.6%)
time	23,781	18,971	(79.8%)
lang	21,074	19,112	(90.7%)
jdom	10,800	9,519	(88.1%)
jaxen	7,132	4,419	(62.0%)
io	7,319	4,170	(57.0%)
num4j	5,437	5,149	(94.7%)
total	410,589	178,731	(43.5%)

Table 7.4: Estimated overhead in hours for evaluating uncovered mutants.

	Uncovered mutants	**Test runtime**	**Overhead**
trove	66,943	15.2 sec	282.6 h
chart	32,860	27.3 sec	249.2 h
itext	110,260	8.4 sec	257.3 h
math	7,592	76.2 sec	160.7 h
time	4,810	13.8 sec	18.4 h
lang	1,962	14.1 sec	7.7 h
jdom	1,281	30.4 sec	10.8 h
jaxen	2,713	12.1 sec	9.1 h
io	3,149	17.4 sec	15.2 h
num4j	288	1.8 sec	0.1 h

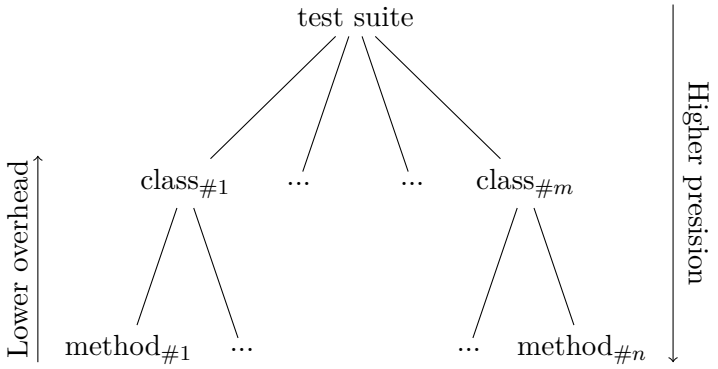


Figure 7.1: Different levels of granularity in JUnit test suites.

7.3.2 Precision of the Mutation Coverage

A test suite of JUnit tests is typically a hierarchical composition of test classes containing several test methods. Regarding such a composed test suite, as visualized in Figure 7.1, there are three different levels of granularity at which the mutation coverage can be measured. The highest one, with a coarse granularity, is the test suite level at which the mutation coverage determines which mutants are covered by the entire test suite. Considering the individual test classes, or even test methods, provides a finer level of granularity, leading to a higher precision in terms of the mutation coverage measure. However, executing test methods independently incurs a much higher overhead caused by additional class loading and, moreover, the instantiation and initialization of the corresponding test classes and the SUT.

Tables 7.5 and 7.6 show the differences in the total runtime and the number of covered mutants when executing the test suite at the class and method level. For all applications, the maximum number of mutants covered by a single test class is clearly lower than the number of mutants covered by the entire test suite, as indicated by the sixth column of Table 7.5. Thus, a mutation analysis process should always operate at minimum at the class level. The average numbers of covered mutants in the last column show that certain mutants have to be covered several times since the product of the average number and the number of tests is greater than the total number of covered mutants given in the first column of the table.

At the method level, the number of covered mutants is in turn lower than the one at the class level for almost all of the applications — and yet, the results are divergent. While *io* and *lang* exhibit a significant reduction, the maximum number of covered mutants for *itext* remains unchanged. Moreover, the average coverage per method is even higher for *itext* and *jaxen*, thus indicating that there are a lot of methods that cover numerous mutants. This result again implies a remarkable overlap in terms of the mutation coverage. Generally, the method level provides the most precise mutation coverage information. However, running the test methods independently leads to a higher overhead in terms of runtime,

Table 7.5: Precision of mutation coverage and total runtime at class level.

	Mutants	Tests	Runtime	Covered per class		
				Min	Max	Avg
trove	6,016	25	15.2 sec	41	2,150	954
chart	35,659	353	27.3 sec	1	4,702	665
itext	16,521	26	8.4 sec	94	9,537	3,906
math	59,195	234	76.2 sec	1	5,957	769
time	18,970	123	13.8 sec	37	6,032	3,011
lang	19,112	101	14.1 sec	1	2,437	310
jdom	9,519	78	30.4 sec	1	3,715	777
jaxen	4,419	78	12.1 sec	1	3,769	1,895
io	4,170	48	17.4 sec	1	2,474	134
num4j	5,149	63	1.8 sec	18	654	195

Table 7.6: Precision of mutation coverage and total runtime at method level

	Mutants	Tests	Runtime	Covered per method		
				Min	Max	Avg
trove	6,016	544	16.8 sec	2	1,053	516
chart	35,659	2,130	80.2 sec	1	3,599	293
itext	16,521	75	18.3 sec	70	9,537	4,861
math	59,195	2,169	138.8 sec	1	3,606	381
time	18,970	3,855	335.4 sec	1	4,939	1,636
lang	19,112	2,039	43.5 sec	1	780	87
jdom	9,519	1,723	127.1 sec	1	3,418	362
jaxen	4,419	699	60.9 sec	1	2,847	2,156
io	4,170	309	19.7 sec	1	598	68
num4j	5,149	218	3.2 sec	4	654	100

as previously stated. As shown in the fourth column of Table 7.6, some applications such as *trove*, *math*, *io*, and *num4j* exhibit a moderate overhead compared to the runtime at the class level in Table 7.5. In contrast, *time*, *jdom*, and *jaxen* incur a significant increase in runtime. Hence, the results clearly document the existing tradeoff between precision and runtime overhead.

7.3.3 Overlap of the Mutation Coverage

The previous section showed that the investigated applications exhibit an overlap in the coverage of mutants. Therefore, we now measure this overlap of the individual test classes. Due to the combinatorial explosion of pairwise comparisons between individual test classes, we focus on relating test classes to their encapsulating test suite and define the overlap $O(t_i, T)$ of a certain test class t_i with its corresponding test suite T as follows:

Definition 7.1 *Overlap* $O(t_i, T) \in [0, 1],\ t_i \in T$

$$O(t_i, T) := \begin{cases} 1, & |Cov(t_i)| = 0 \\ \frac{|Cov(t_i) \cap Cov(T \setminus t_i)|}{|Cov(t_i)|}, & |Cov(t_i)| > 0 \end{cases}$$

In this definition, the set T denotes a test suite containing all its test classes t_x, meaning that $T := \cup\{t_x\}$. Without loss of generality, the definition assumes that the test class t_i, of which the overlap is determined, is an element of the set T. Moreover, the operator $|s|$ represents the cardinality of the set s and the function Cov provides the set of mutants covered by the corresponding set of test classes. Intuitively, this overlap metric describes the similarity of a test class to all other test classes within the same test suite.

Figure 7.2 illustrates the distribution of the overlap for all analyzed applications using a box-and-whisker plot, where the thick line in the middle represents the median. The box itself shows the distribution of the data between the upper and lower quartile, thus including 50% of the data. By excluding the outlier values, the lower and upper whiskers denote the minimum and maximum value, respectively. The extreme values themselves are visualized by means of the circles beyond the whiskers. Within the plot, two exceptional patterns can be identified. On the one hand, there are applications such as *jaxen* and *time*, where even the minimum overlap of the outlier values is at least 50% and the median is almost 100%. On the other hand, the median of the overlap is only 75% for applications such as *num4j* and *lang*. Moreover, these programs contain test classes that have no overlap at all, indicated by an overlap value of 0%.

Recalling the three necessary conditions for killing a mutant, a high overlap of the mutation coverage does not imply that the test cases within the test suite are highly redundant since the mutation coverage only refers to the reachability condition. However, the probability of killing a mutant is much higher if the mutant is covered by several test classes.

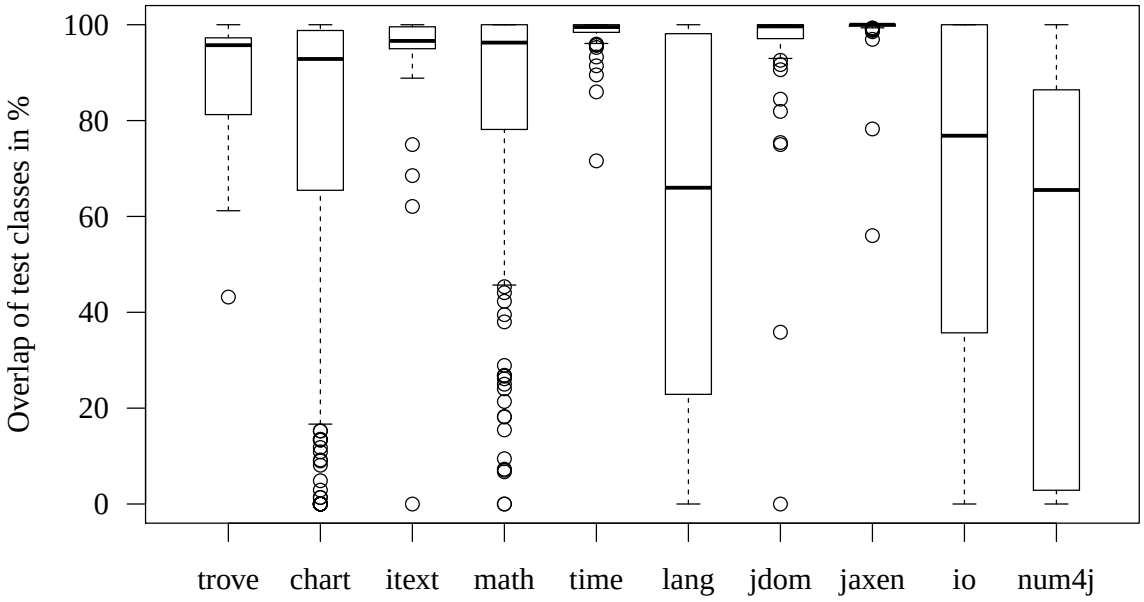


Figure 7.2: Coverage overlap distribution of the individual test classes related to the corresponding test suite for all investigated applications.

7.3.4 Runtime of Test Cases

Intuitively, the runtime of a test suite has an essential impact on the total time needed for the mutation analysis because every covered, and yet not killed, mutant has to be evaluated by executing the test suite. As previously mentioned, regarding the test suite at the class or method level leads to more precise mutation coverage information, which reduces the number of mutants that have to be evaluated for a certain test. Nevertheless, a very long-running test case can still result in a prolonged mutation analysis, even for a small number of covered mutants. Hence, the number of covered mutants and the runtime of the individual test cases are the determining factors for the total runtime.

Therefore, we investigate how the individual test classes collectively form the total runtime of the corresponding test suite. The runtime distribution of the individual test classes for all analyzed applications is again visualized by means of box-and-whisker plots in Figure 7.3. The extremely thin boxes and the short, if existing, whiskers clearly indicate that most of the test classes have a short runtime of less than 1 second. Even though the majority of the test classes have a rather low runtime, there are a few extreme values for which the runtime differs by an order of magnitude. Thus, for all applications, Table 7.7 additionally shows the number of test classes, along with both the cumulative runtime and the extremum of all test classes. This table demonstrates that the outlier values constitute a substantial proportion, sometimes even most, of the total runtime. For instance, in consideration of a total number of 353 test classes for the *chart* application, the test class with the longest runtime of 10.2 seconds forms more than 37% of the total runtime.

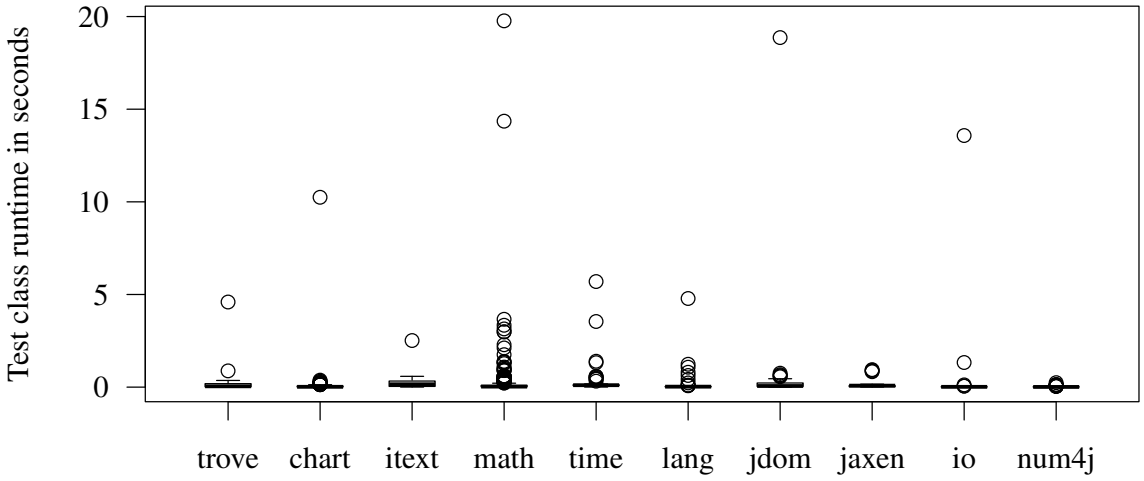


Figure 7.3: Runtime distribution of the individual test classes.

Table 7.7: Cumulative runtime and extremum of all test classes.

	Test classes	**Cumulative***	**Extremum***
trove	25	15.2	4.6 (30.3%)
chart	353	27.3	10.2 (37.4%)
itext	78	8.4	2.5 (29.8%)
math	234	76.2	19.8 (26.0%)
time	123	13.8	5.7 (41.3%)
lang	101	14.1	4.8 (34.0%)
jdom	78	30.4	18.9 (62.2%)
jaxen	78	12.1	0.9 (7.4%)
io	48	17.4	13.6 (78.2%)
num4j	63	1.8	0.2 (11.1%)

*Runtimes reported in seconds

7.3.5 Visualizing the Overlap and Runtime

In order to examine the correlation between the runtime and the mutation coverage overlap of the tests, we use scatter plots to visualize the overlap in conjunction with the individual test runtime. Due to the large number of analyzed applications, and to avoid a confusing set of diagrams, we focus on the two identified overlap patterns as representatives for all of the chosen applications. Figure 7.4 shows the plot for *time*, an application representative of those with high overlap, while Figure 7.5 gives the plot for *num4j* as a representative of those applications with a distinctive distribution in the coverage overlap.

Within the scatter plots, every data point indicates that the mutant on the vertical axis is covered by the test class or test classes on the horizontal axis. Intuitively, the plot visualizes a matrix representing a mutant-covered-by-test-class map. Since the introduced overlap metric is a measure for the similarity of each data line to all others, the plot for *time* clearly reveals that there is indeed a substantial overlap for almost all of the test classes. In contrast to this obvious overlap, the plot for *num4j* reveals that only a fourth of the mutants are overlapped by several test classes.

Besides the overlap, the plots visualize the runtime of the individual test classes. The test runtime in milliseconds is color coded by means of the color palette that is aligned to the right of the scatter plot. According to the chosen color gradient, the darker the color of a data line, the longer is the runtime of the corresponding test class. Both plots manifest that long-running tests, represented by the dark data lines, have a notable overlap with a lot of short-running tests. Moreover, when considering the original ordering of the tests for *num4j*, the test class with the longest runtime is placed before many of the overlapping short-running test classes. As a consequence, the long-running test has to be executed for all covered, and yet not killed, mutants – even though those mutants are also covered by test classes with a much shorter runtime. Since the results from the other applications conform to these observations, the characteristics of these plots suggest that a reordering according to the runtime could significantly improve the mutation analysis process.

7.4 Optimized Mutation Analysis Workflow

We now present an optimized mutation analysis workflow based on the observations and evidence given in the previous Section 7.3, which can be summarized as follows:

OBS 1 *The mutation coverage is 43.5% on average and ranging between 8.2% and 94.7%.*

OBS 2 *Analyzing the mutation coverage on the class or method level is much more precise. However, executing the test methods independently incurs a significant overhead.*

OBS 3 *Most tests within a test suite have a notable overlap with all remaining tests in terms of mutation coverage.*

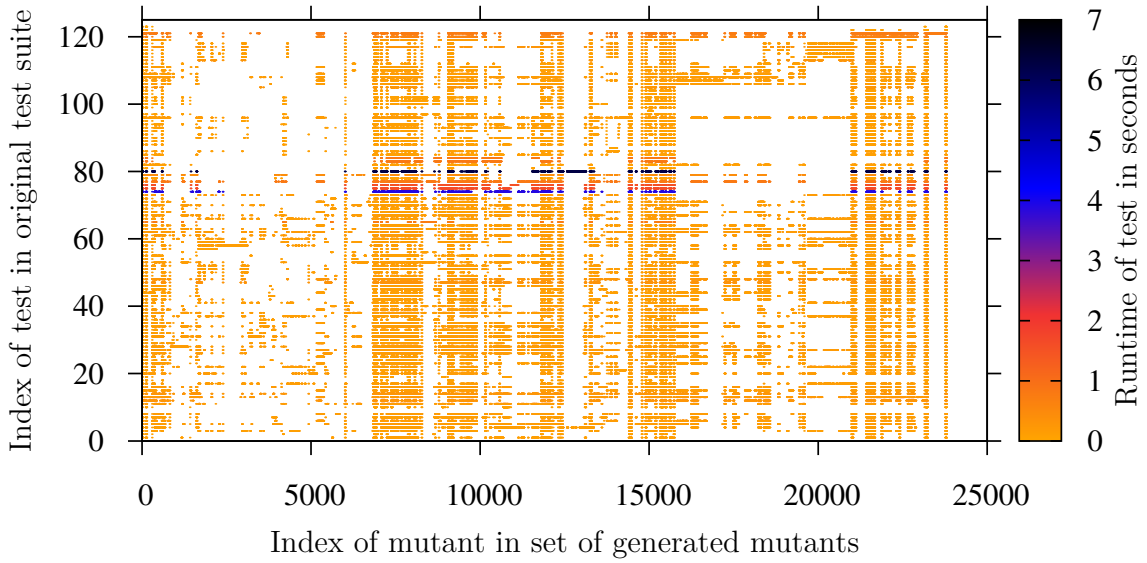


Figure 7.4: Mutation coverage with corresponding runtime for *time*.

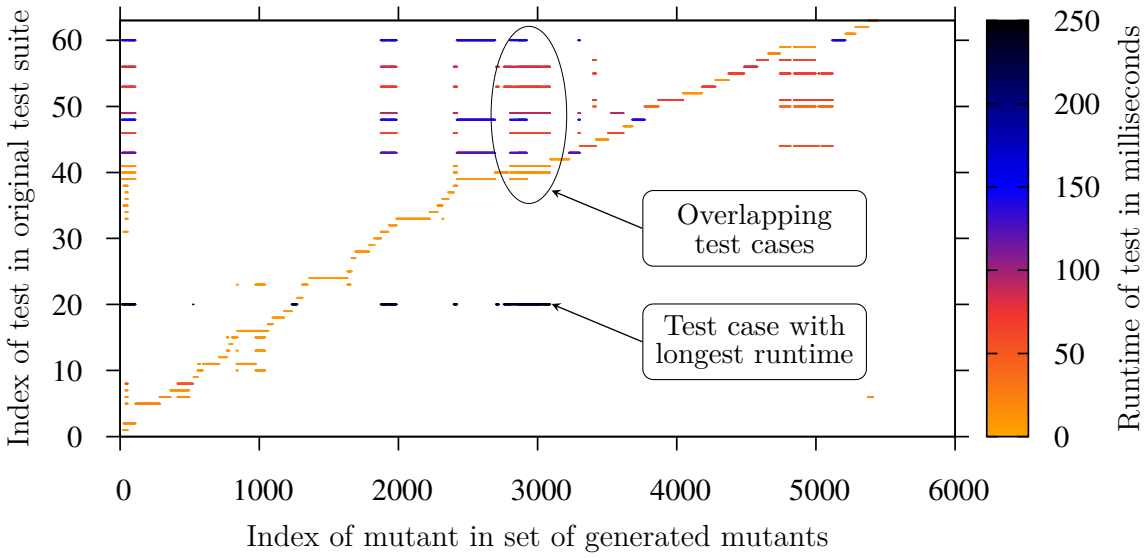


Figure 7.5: Mutation coverage with corresponding runtime for *num4j*.

OBS 4 *The runtime of individual test classes within a test suite differs, sometimes even by an order of magnitude.*

OBS 5 *Long-running tests have an essential overlap with many short-running tests but the investigated existing JUnit test suites are not ordered according to runtime.*

7.4.1 Gather Mutation Coverage Information

Due to the fact that a lot of mutants are not covered by the test suite, we exploit the mutation coverage information provided by Major's driver. A program instrumented by Major reports the coverage information to the driver if and only if the unmutated version is executed and the corresponding flag is enabled. This condition is crucial since gathering the coverage information involves method calls and incurs a notable overhead. Hence, determining the coverage information during the mutation analysis process would significantly increase the total runtime. Depending on the level of granularity, the coverage information is cached for each test class or test method.

7.4.2 Estimate Test Runtime and Prioritize Test Cases

Given the overlap of the individual tests and the runtime, which differs significantly, the runtime is estimated for every test method and entire class by executing the original version. Attempting to produce a runtime approximation that is as precise as possible, this step executes the original version of the program without enabling the mutation coverage. Section 6.3 within the previous chapter provides further details on this approximation and also discusses the used timeout-based heuristic for long-running mutants.

Next, based on the runtime results, the tests are sorted in descending order to ensure that the long-running tests will be executed last. This prioritization strategy is based on the assumption that unit tests have no dependencies, and hence the order is irrelevant. Even though this assumption is also specified for unit testing frameworks such as JUnit, we verify that reordering the tests does not break the test suite by executing the unmutated version with the prioritized test suite. In order to increase the confidence in the test's independence, a randomized order of the tests is also executed.

7.4.3 Threshold-based Splitting of Test Classes

In light of the tradeoff between precision of the mutation coverage on the one hand and runtime overhead on the other hand, we present two hybrid approaches. Extracting test methods with an exceptionally long runtime or splitting entire long-running test classes

seems to be the most promising. Therefore, we define the following two hybrid approaches that represent both kinds of splitting strategies.

Class-hybrid: Extract an individual test method from its corresponding test class if and only if the runtime of the pre-initialized test method is greater than a given threshold th_m.

Method-hybrid: Split a test class into all of its individual test methods if and only if the total runtime of the pre-initialized test class is greater than a given threshold th_c.

With regard to the threshold parameters that are used in both approaches, we determine an appropriate value by taking into account the average initialization time of a test class for th_m and the mean number of test classes within a test suite for th_c. Additionally, based on the observation that the test suites of all applications exhibit a significant mutation coverage overlap, the overhead of executing a test method separately should not exceed 100% of its runtime. For instance, the average initialization time of a test class ranges between 10 and 50 milliseconds for all applications. Thus, a threshold th_m of 50 milliseconds ensures that an individual test method is not extracted if its runtime is smaller than the initialization time of the enclosing test class.

7.4.4 Complete Mutation Analysis Workflow

Integrating the individual steps into a complete process leads to the optimized mutation analysis workflow that is illustrated in Figure 7.6. Generally, this process consists of three individual but consecutive phases:

1. Mutant generation phase that generates and compiles all mutants into the system under test.
2. Preprocessing step that gathers the mutation coverage and test case runtime information.
3. Mutation analysis with reordered and potentially split test suite employing the mutation coverage.

The dashed line within the diagram indicates that the threshold for the splitting strategy is estimated based on the mutation coverage overlap. It is important to note that we do not calculate the overlap of every test method with its test class but rather use the overlap of the test classes within a corresponding test suite. The splitting strategy may be more effective with an accurate overlap value for test methods within their encapsulating test classes. However, the investigation of this matter is left open for future research.

7.5 Empirical Evaluation

To empirically evaluate the workflow shown in Figure 7.6, we implemented it in Major's analysis component that extends the Apache ant build system. Therefore, all of the chosen

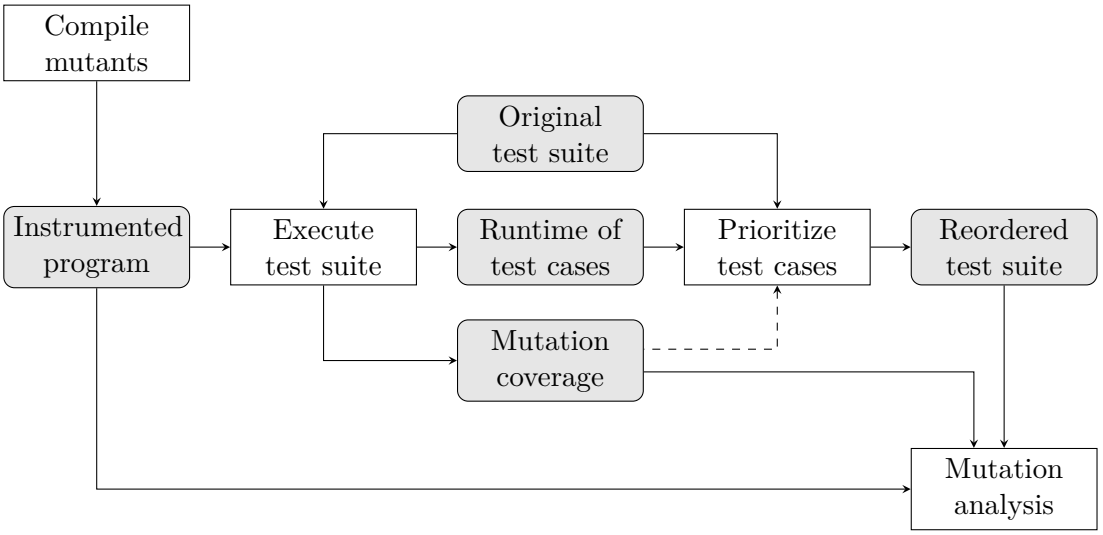


Figure 7.6: Optimized mutation analysis process that exploits mutation coverage and runtime information of test cases.

applications provide an ant-like build configuration. With regard to the performance evaluation, of particular interest are the runtime improvements due to reordering and splitting and the variation in efficiency due to differences in the coverage overlap and the effectiveness of the test suite. Table 7.8 reports the runtimes for the complete mutation analysis when employing the different reordering and splitting strategies for all applications. To better visualize the results, the fastest approach is highlighted for every application.

In order to minimize any potential side effects, all analyses were performed on a single machine[1] that did not take advantage of parallelization. Additionally, we measured the real runtime instead of CPU time due to the fact that most analyzed applications are not CPU-bound. Hence, the CPU time is much lower and does not adequately reflect the time needed to perform the entire mutation analysis.

Within the table, Original denotes the mutation analysis of the test suite without any prioritization or splitting. Method-level and Class-level describe the results for sorting and executing the test suite at the method level and class level, respectively. The runtime results for the two suggested approaches, namely Method-hybrid and Class-hybrid, are shown in the corresponding columns. The last column of Table 7.8 additionally shows the mutation score since the effectiveness of the investigated test suites is also a crucial factor. It is important to consider the mutation score because the prioritization technique is based on the assumption that a test suite kills a certain number of mutants, and hence the number of live mutants decreases over time. Furthermore, a mutant is always killed by the fastest test case that can detect it within the sorted test suite. Thus, if the mutation score is extremely low, reordering will only yield a marginal improvement since the entire test suite has to be executed for the majority of the mutants.

[1] Commodity GNU/Linux workstation with Intel Xeon CPU @2.4GHz, 16GB of RAM, and kernel version 2.6.32-5-amd64.

Table 7.8: Runtimes for different prioritization and splitting strategies.

	***Original**	***Method-level**	***Method-hybrid[1]**	***Class-level**	***Class-hybrid[2]**	**Mutation score**
trove	107.81	41.68 (-61.3%)	44.93 (-58.3%)	55.96 (-48.1%)	36.89 (-65.8%)	66.6%
chart	608.60	950.55 (56.2%)	564.14 (-7.3%)	270.40 (-55.6%)	309.88 (-49.1%)	36.3%
itext	644.43	1381.51 (114.4%)	1127.25 (74.9%)	627.89 (-2.6%)	674.18 (4.6%)	24.0%
math	793.19	394.60 (-50.3%)	388.39 (-51.0%)	674.73 (-14.9%)	381.10 (-52.0%)	79.1%
time	504.44	1182.61 (134.4%)	559.62 (10.9%)	470.03 (-6.8%)	410.59 (-18.6%)	85.7%
lang	42.75	27.58 (-35.5%)	23.31 (-45.5%)	29.93 (-30.0%)	19.11 (-55.3%)	74.2%
jdom	120.53	135.53 (12.4%)	189.08 (56.9%)	117.42 (-2.6%)	105.01 (-12.9%)	83.4%
jaxen	343.40	1773.15 (416.4%)	1521.79 (343.2%)	338.51 (-1.4%)	357.16 (4.0%)	43.6%
io	5.72	5.64 (-1.5%)	4.11 (-28.1%)	4.83 (-15.5%)	4.35 (-23.9%)	78.0%
num4j	2.54	2.12 (-16.5%)	1.95 (-23.1%)	1.99 (-21.6%)	1.94 (-23.5%)	68.1%
avg		56.9%	27.3%	-19.9%	-29.2%	63.9%

*Runtimes reported in minutes; [1]Threshold $th_c = 500$ milliseconds; [2]Threshold $th_m = 50$ milliseconds

When analyzing the entire test suite at the method level, meaning that every test method is executed independently, the two identified overlap patterns give distinction to the results. The runtime for the applications *itext*, *time*, and *jaxen*, which have a huge overlap, is increasing dramatically due to the incurred overhead. Yet, applications with a lower overhead such as *trove* and *math* yield a considerable runtime decrease of up to 61.3%.

The Method-hybrid approach reduces the number of individual test methods by only splitting long-running test classes. Due to the reduction, this approach improves the runtime for all applications but cannot compensate for the huge overhead of *itext*, *time*, and *jaxen*.

Sorting the test suite at the class level according to the runtime of the individual test classes yields an improvement for all of the applications due to the existing overlap between test classes and the divergent runtimes of the individual tests. Ranging between 1.4% for *jaxen* and 55.6% for *chart*, sorting at class level yields an average decrease of almost 20%. The improvement for *jaxen* is relatively low for two reasons. The runtime of the individual test classes is very homogeneous with a maximum of only 900 milliseconds and furthermore the mutation score is only 43.6%.

By additionally employing the splitting of the Class-hybrid approach, the runtime can be considerably reduced even further for most of the applications. With a speedup of 65.8% for *trove* and an average improvement of 29.2% for all applications, this approach yields the best results overall. However, the necessary runtime increases by approximately 4% for two applications, namely *itext* and *jaxen*. The reason is again the low mutation score in conjunction with the huge overlap of the individual test classes. For instance, some long-running test methods of *itext* cover exactly the same number of mutants as the entire, enclosing test class. Hence, the extraction of such methods introduces a higher overhead without increasing the precision of the mutation coverage.

For the *math* and *itext* applications, the diagrams in Figure 7.7 and 7.8 visualize the mutation analysis process using the original test suite and the Class-hybrid approach, which is the most efficient approach for all applications. Within these diagrams, the upper plot illustrates the runtime of the individual test cases and the lower plot depicts the ratio of analyzed-to-covered mutants. This lower-is-better ratio decreases if mutants covered by a certain test are already killed, and hence will not be executed again. Additionally, the width of the boxes exhibits the time needed to execute the corresponding test for all of the covered, and yet not killed, mutants.

With regard to the original test suite of the *math* application, the test class with the longest runtime is placed in the middle and the time necessary to execute this individual test class for all covered mutants is a considerable proportion of the total runtime of the complete process. Furthermore, the ratio of analyzed-to-covered mutants is still 60%. By means of the Class-hybrid approach, this long-running test class is split, so that the methods with a runtime larger than the threshold are extracted and the resulting set of individual tests is sorted. Accordingly, the last individual test possesses the longest runtime which is, however, with 16 seconds, smaller than the entire long-running test class from which it has been extracted. Given the coverage overlap of the tests and a mutation score of almost 80%, the ratio of analyzed-to-covered mutants is rapidly decreasing, and hence the

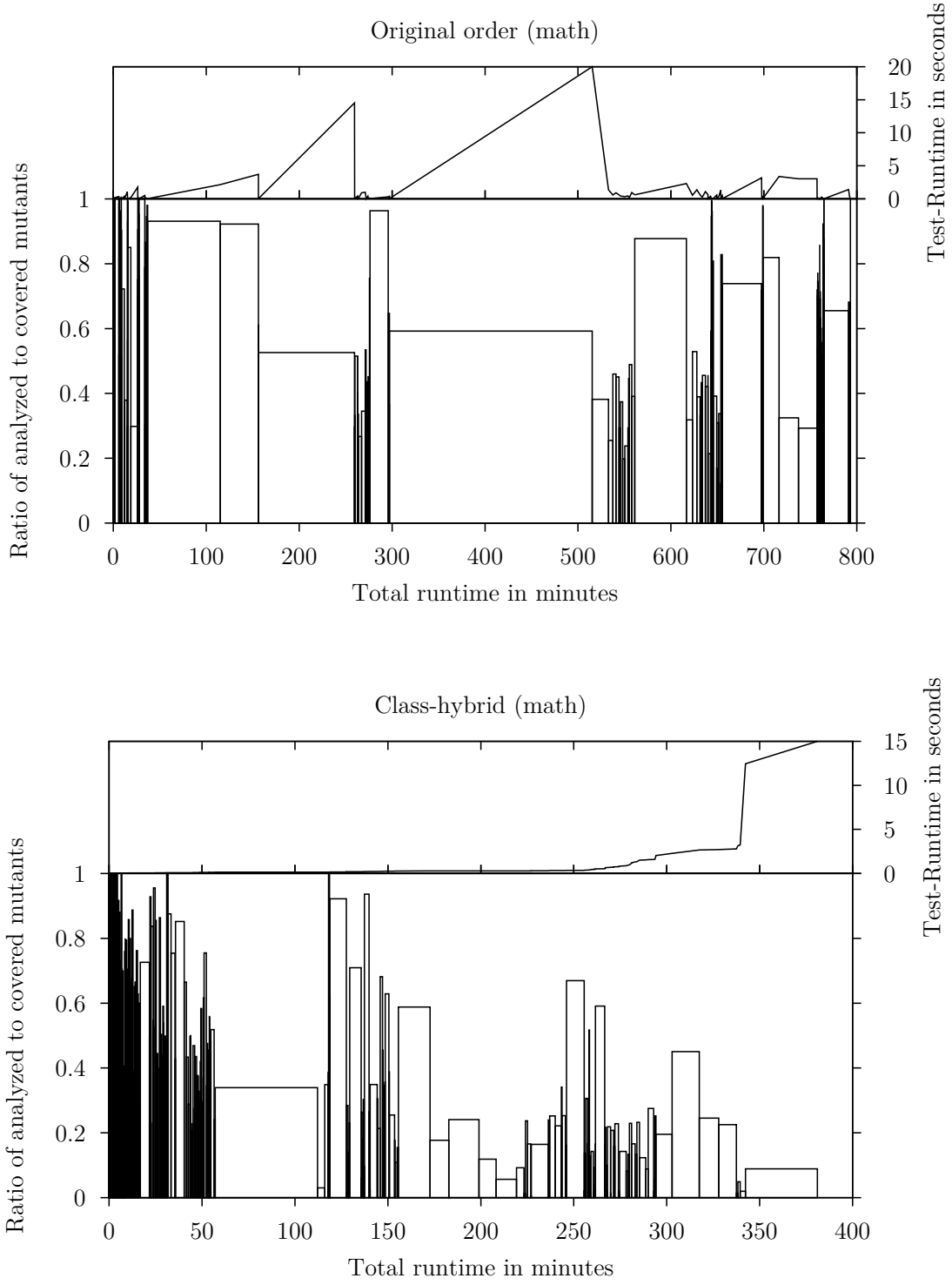


Figure 7.7: Visualization of the complete mutation analysis process for *math* using the original order and the class-hybrid approach.

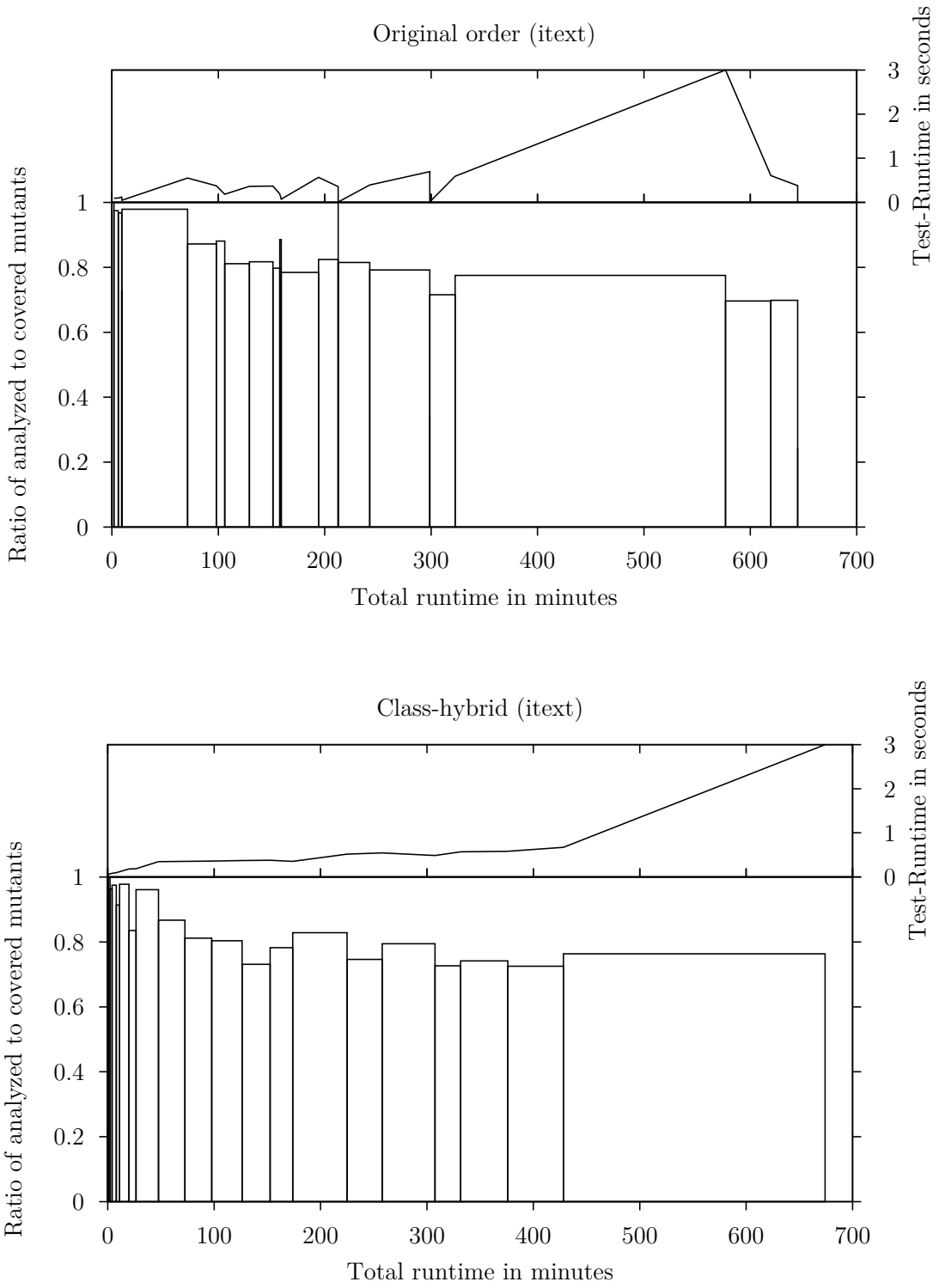


Figure 7.8: Visualization of the complete mutation analysis process for *itext* using the original order and the class-hybrid approach.

extracted long-running test method is only executed for approximately 10% of the covered mutants.

An example of an application with both an extremely low mutation score and a better-ordered original test suite is *itext*, for which Figure 7.8 illustrates the mutation analysis process. The runtime characteristics of the other applications conform to these two examples, and thus are not separately depicted.

As with every empirical study, it is crucial to discuss the threats to validity. The representativeness of the chosen applications might be again a potential threat to external validity. However, we controlled this threat by examining programs that differ significantly in their size, complexity, and operation purpose. Starting with an initial set of the five applications *math*, *time*, *lang*, *io*, and *num4j*, we improved the generalizability by successively adding new applications with varying operation purposes and originations from different developer communities. While adding more programs to the empirical study, we could identify the two exceptional overlap patterns to which all of the investigated applications can be related. Therefore, we judge that the reported results are meaningful and indeed transferable to other applications. Relying on the sufficient set of mutation operators could be a threat to internal validity. Different or additional operators may affect the improvement results due to differences in the mutation coverage overlap and the mutation score. However, the applied operators are frequently used in the literature, and thus provide comparable results [Offutt and Untch, 2000; Namin et al., 2008; Schuler and Zeller, 2009]. Moreover, the investigated test suites exhibit a significant divergence with regard to the mutation score and therefore the study also examines boundary cases.

7.6 Related Work

Considering the efficiency of mutation analysis, several approaches proposed in the literature belong to one of three categories: *do fewer*, *do smarter*, and *do faster* [Jia and Harman, 2011; Offutt and Untch, 2000]. *Do fewer* approaches employ selective or sampling strategies to reduce the number of generated mutants by either (randomly) selecting a subset of all generated mutants or by reducing the number of applied mutation operators. Namin et al. [2008], as well as Offutt et al. [1996], determined a sufficient set of mutation operators that can be applied without a major loss of information.

Concerning this relationship between reduction and accuracy, the utilization of non-redundant mutation operators is a special case of a *do fewer* approach since the exclusion of subsumed mutants even increases the accuracy of the mutation score. As described in detail in Chapter 6, using the non-redundant versions of the mutation operators significantly reduces the number of generated mutants, and hence the necessary runtime to run the mutation analysis. The presented workflow employs such non-redundant mutation operators and improves the efficiency without loss of information by reordering and splitting the analyzed test suites.

While *do smarter* methods exploit distributed or multi-core systems to parallelize the mutation analysis, *do faster* approaches aim at improving the efficiency without reduction

or parallelization. Even though the presented optimized workflow belongs to the group of *do faster* techniques, it can nevertheless be combined with sampling or selective approaches and it also can be easily parallelized to fully utilize the computational power of current machines.

To the best of our knowledge, this is the first approach that employs test suite prioritization to achieve efficient and scalable mutation analysis. Nevertheless, a wide variety of software testing techniques leverage mutation analysis for test suite prioritization. For instance, Elbaum et al. [2002] use mutation testing tools to support the reordering of a test suite according to the fault exposing potential of a test case. Furthermore, both Andrews et al. [2005] and Do and Rothermel [2006] use mutation analysis to empirically evaluate test suite prioritization techniques. It is, however, crucial to state that the described workflow does not aim at reducing or optimizing a given test suite but rather speed-up the mutation analysis process to assess its quality.

7.7 Summary

This chapter has addressed the challenges associated with the efficiency and scalability of mutation analysis and makes several contributions to this field. By leveraging existing definitions of non-redundant mutation operators, an empirical evaluation shows that the reduced sets result in a considerably decreased total number of generated mutants. However, the remaining number of mutants for large applications is still substantial. In order to further improve the runtime of mutation analysis for those applications, this chapter also ascertains several characteristics of existing unit test suites.

Drawing on the observations concerning these characteristics of real-world test suites, this chapter presents an optimized mutation analysis process that exploits mutation coverage and test runtime information. Empirically evaluated on ten open-source applications with 410,000 lines of code and 550,000 generated mutants in total, the suggested approach reduces the runtime by up to 65% and 30% on average.

Overall, the results given in this chapter convincingly demonstrate that mutation analysis is indeed applicable to large programs, and hence ready for a transfer and a wider integration into industry.

Part III

Assessing Partial Oracles with Mutation Analysis

Chapter 8

Background on the Oracle Problem and Partial Oracles

This chapter is based on [Just and Schweiggert, 2011].

As previously mentioned, a *testing strategy* is composed of two parts, namely a model for test data generation and an appropriate test oracle to verify the outputs. The model for generating the test data provides adequate input values, where the suitability is usually determined with regard to an adequacy criterion, for instance a certain code coverage criterion. Models for test data generation as well as several adequacy criteria have been discussed in Chapter 2.

The test oracle is responsible for the decision whether a given test case passes or fails. The oracle therefore consists of two entities, one to determine the expected output for a given input and one to compare this expected output to the actual output of the system to be tested. Hence, a simple comparator cannot be considered as test oracle due to the missing capability of predicting the correct output (cf. [Binder, 1999]). This section introduces the oracle problem and approaches to overcome this problem by focusing on partial oracles. After describing the basics of assertions, contracts, and invariants in the context of partial oracles, this section provides the necessary background on metamorphic testing.

8.1 The Oracle Problem

Finding or generating an adequate test oracle is not a straightforward task and may be associated with a significant effort. Now, the *oracle problem* describes the situation when such an oracle is unattainable or if the generation requires an excessive effort. Even though not well-defined, the effort to generate an oracle is excessive if it is equal to or greater than the effort to develop the system under test. For instance, this is clearly the case when the system under test is re-implemented as test oracle. Another example for systems that can also cause the oracle problem is the implementation of a randomized algorithm.

Furthermore, Davis and Weyuker [1981] and Weyuker [1982] refer to programs for which a test oracle is not available as non-testable programs. The term non-testable may be misleading because of the fact that such programs can be indeed tested to a certain extent with partial oracles, as further explained in Section 8.2.

Perfect Oracle

The perfect oracle denotes an oracle that can produce the correct output for every given input. This perfect oracle, however, cannot exist for non-trivial programs. If the perfect oracle was existing, there would not be a need for implementing a system that is functionally equivalent to this oracle – one could simply use the perfect oracle instead. Hence, the perfect oracle is more of theoretical importance.

Gold Standard Oracle

A gold standard oracle [Binder, 1999] is an existing pendant of the system to be implemented. Gold standard oracles are for instance trusted legacy systems or generally speaking equivalent, technically mature applications developed in a different programming language or build for a different operating system. It is crucial to state that a gold standard oracle is not claimed to be the perfect oracle since the legacy implementation, even though sophisticated, may still contain defects.

8.2 Partial Oracles

Concerning the oracle problem, various standard solutions exist which are, however, only employable in rare situations (cf. [Binder, 1999]). To alleviate the oracle problem for non-testable programs, Weyuker [1982] proposed the usage of *partial oracles*, a promising class of oracles that are considered to be easily automatable, more often applicable, and therefore employable for automating software tests [Bertolino, 2007].

These oracles are referred to as partial oracles because they cannot predict the correct output of a system for all possible inputs. In fact, most partial oracles do not determine an expected output but rather exploit constraints of the underlying function or algorithm in order to identify faults within the tested system. A trivial example for a partial oracle concerning the trigonometric sine function is for instance the equation:

$$\sin(x) = \sin(x + 2\pi) \tag{8.1}$$

Now, if equation 8.3 is not fulfilled by an implementation of the sine function, this implementation can be judged to be faulty even without the knowledge of the correct output of $\sin(x)$ and $\sin(x + 2\pi)$, respectively. In contrast, an implementation that does not violate this constraint may still contain a defect if it computes the same wrong output for $\sin(x)$ and $\sin(x + 2\pi)$. It is crucial to state that the machine accuracy has to be taken into account when implementing such equations in order to avoid false-positives.

In addition to this oracle that is based on well-defined equivalence relations, checking the range of the outputs is another example for a partial oracle. Concerning the sine function, a correct implementation has to fulfill the necessary condition that the outputs are within the range $[-1, 1]$. Such a partial oracle, however, accepts all outputs within this range, and hence might be less powerful. In contrast to testing equivalence relations or the output range, testing with special values provides a precise result for these special cases since the expected output is know in advance. With regard to the sine function various special cases such as $\sin(0)$, $\sin(\frac{\pi}{2})$, or $\sin(\pi)$, for which the output is well-known, can be tested with this method. It is important to note that the corresponding oracle of this testing technique is also a partial one since it can only predict the expected output for a few inputs.

8.2.1 Assertions, Contracts, and Invariants

Generally, assertions are used to ensure that certain constraints are fulfilled at a particular point in the program. Lots of programming languages have the concept of assertions already defined in their language specification, thus enabling an easy implementation of checks. The Java programming language [Gosling et al., 2005], for instance, provides assertions by means of the following statement:

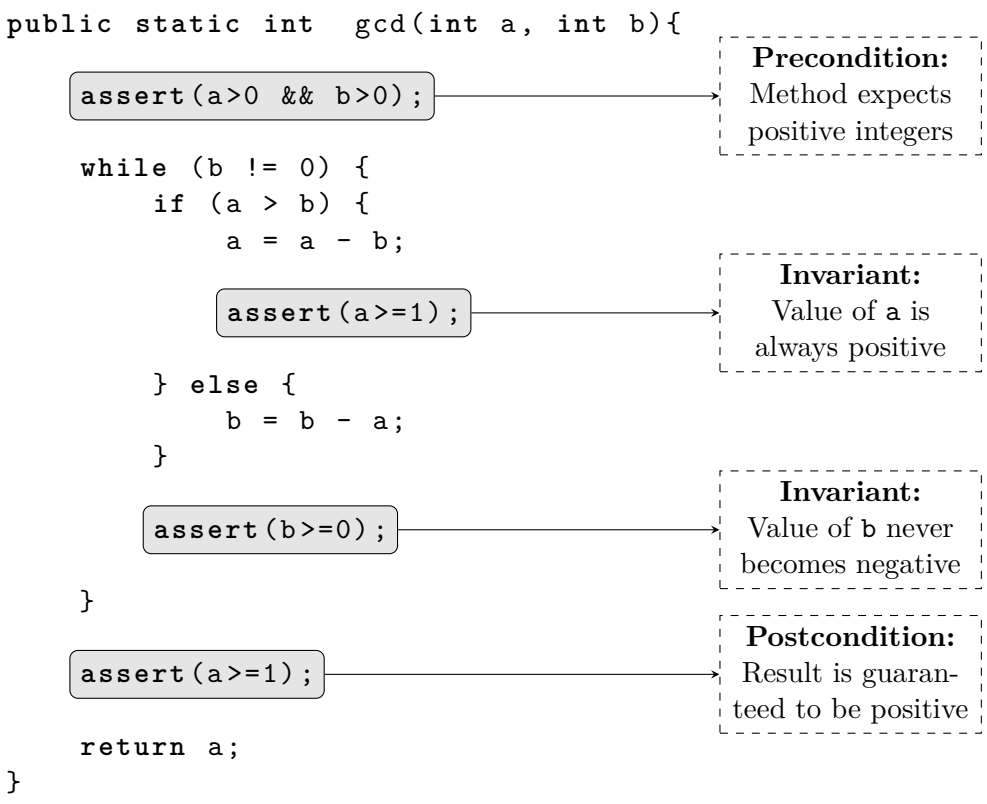
```
assert <boolean-expr>;                                    (8.2)
```

By means of such an assertion statement, every condition that evaluates to a boolean value can be verified. As defined in the Java programming language, an exception is thrown if the checked condition is violated, that is the boolean expression evaluates to false. While most conditions, which are checked with such an assertion statement, are necessary ones, assertions can generally be used to verify sufficient conditions for outputs too. For example, when applying special value testing, the actual output can be verified with an assertion that compares the actual to the expected output. Hence, assertions are not limited to particular oracles and can always be employed to verify conditions for certain values or states.

Contracts

Meyer [1997] suggested the class design technique Design-by-Contract that defines rules and constraints that have to be fulfilled by a client and a server in order to enable a proper execution. In this scenario the server is usually a certain class and the client is the caller of a particular method of this class. Moreover, the rules clearly define what the client has to provide and what the server has to deliver. Even though defined for class design with several methods involved, this principle can also be applied at the method level where the contract has to be fulfilled between the caller and the method itself. Generally, when considering a contract, the following three types of conditions can be distinguished:

- Preconditions
- Invariants
- Postconditions



Listing 8.1: Iterative implementation that determines the greatest common divisor of two positive integer numbers.

Listing 8.1 shows the iterative implementation of a method that determines the greatest common divisor of two positive integer values. Even for this trivial example, several necessary conditions can be defined, as highlighted within the code. These conditions are checked within the illustrated method by means of assertions.

Preconditions

All requirements and conditions that have to be fulfilled before the method can be properly executed are referred to as preconditions. With regard to the implementation of Listing 8.1, the algorithm is only defined for positive numbers and therefore the first `assert` statement verifies whether the provided inputs are indeed positive integers. Whether or not such a check is necessary depends on the type system of the corresponding programming language. Since Java does not support unsigned data types, this particular precondition has to be verified with a separate assertion, the required data type itself (i.e., integer numbers) can be enforced by the method signature, though.

Invariants

An invariant is a property (e.g., for a variable or state) that holds at a certain point within a program. Examples for invariants include the following:

1. `x > 3`
2. `ptr != null`
3. Array `array` is sorted
4. Tree `tree` is a binary tree

Invariants can also express properties of more complex data structures, as indicated by the third and fourth property. However, the necessary check and the implementation of corresponding assertion might be complex as well.

Ernst et al. [2001] developed a technique and a research tool to dynamically detect invariants by analyzing program traces. This is particularly useful for testing purposes since the detected invariants can be exploited as partial oracles, if they hold in general.

With regard to mutation testing, Schuler and Zeller [2010] investigated the adequacy of dynamic invariants for identifying mutants that are not semantically equivalent, and hence violate several invariants detected in the original program.

Postconditions

Conditions that are guaranteed to be fulfilled at the end of an execution are referred to as postconditions. The necessary postcondition of the implementation in Listing 8.1, again checked with an assertion, is that the greatest common divisor of two whole numbers is always a positive integer. It is crucial to state that this assumption is only valid if and only if the aforementioned preconditions hold. Therefore, both the pre- and postconditions form the contract of this method.

8.2.2 Metamorphic Relations

Testing with metamorphic relations is a generalization of testing with the aforementioned equivalence relations. Proposed by Chen et al. [2003], metamorphic relations also belong to the class of partial oracles and exploit properties of the SUT in order to evaluate the corresponding dependencies between inputs and outputs. These properties are used to form metamorphic relations [Chen et al., 2003] that consist of two relations R_I and R_O. Assume I to be the input domain, O the output range, and f a mapping $f : I \to O$. In addition, $R_I \subseteq I^n$ and $R_O \subseteq I^n \times O^n$. The pair (R_I, R_O) is called a metamorphic relation if and only if the following implication is fulfilled:

Definition 8.1 *Metamorphic relation* (R_I, R_O)

$$(i_1, \ldots, i_n) \in R_I \Rightarrow (i_1, \ldots, i_n, f(i_1), \ldots, f(i_n)) \in R_O$$

Compared to equivalence relations, this definition is indeed a generalization since it allows for arbitrary relations between inputs and outputs. It is also important to note that an actual match of the outputs for two related inputs is not required. With regard to the generation of test cases that fulfill these metamorphic relations, an additional test case, called follow-up test case, is usually generated based on a given relation R_I. The results of both inputs, computed by the system under test, then have to fulfill the corresponding relation R_O. Considering the properties of the sine function again, the following equation has to be fulfilled by a correct implementation:

$$\sin(x) = -\sin(-x) \tag{8.3}$$

Now in order to test for defects in a given implementation, this equation can be used as a metamorphic relation. Starting with an arbitrary input i_1, a follow-up test case i_2 is generated by means of the negation of i_1. The outputs of the two individual computations $sin(i_1)$ and $sin(i_2)$ are then verified with the corresponding output relation, which is again a negation. More complex metamorphic relations can be developed when considering, for instance, the relations between the sine and cosine functions.

8.3 Summary

This chapter has introduced the oracle problem and various partial solutions to alleviate it. With a main focus on metamorphic relations, this chapter describes several existing partial oracle approaches that have been proposed to overcome the oracle problem. Furthermore, it discusses the main characteristics of partial oracles, which can be summarized as follows:

- Partial oracles can only verify necessary conditions since they cannot predict the correct output in general
- Due to the partial evaluation of outputs they may produce false-negative results
- The quality of partial oracles has to be assessed to achieve reliable results

Chapter 9

Automating Unit and Integration Testing with Partial Oracles

The content of this chapter has been published in [Just and Schweiggert, 2010] and [Just and Schweiggert, 2011]

The oracle problem is a crucial part of current research on automating software tests. Partial oracles seem to be a viable solution, but their suitability for different testing steps and the general applicability for various systems remains still to be shown. As stated in Chapter 3, the evaluation of such testing strategies is a common application domain for mutation analysis. We therefore apply a bipartite approach that is based on mutation analysis to assesses the adequacy of partial oracles for both unit and integration testing.

This chapter investigates metamorphic relations, a certain type of partial oracles, and presents a study in which these partial oracles are applied in order to automatically test a jpeg2000 encoder as an example for a modular software system with several integrated units and components. Additionally, this chapter presents possibilities for improving the effectiveness as well as the efficiency of the employed partial oracles. It shows how the knowledge of certain characteristics of the system under test, such as linearity or time-invariance, may lead to a better choice of partial oracles and thus to an improved effectiveness and efficiency.

9.1 Introduction

Increasing the level of automation is essential with regard to cost effectiveness in all areas of software engineering and especially software testing. However, automating software tests is a non-trivial and complex task which concerns, besides the automated execution of test cases, the automated generation of appropriate input values and the evaluation of the corresponding outputs by means of an appropriate testing strategy. In order to achieve reliable results from testing, this testing strategy has to cover the structure as

well as the semantics of the implementation which is to be investigated. Moreover, the adequacy of a certain test data generation model and the corresponding test oracle may vary for different applications.

Regarding the test data generation, random or adaptive random testing is an established approach since it is suitable for most environments and an unbiased technique (cf. [Hamlet, 1994; Mayer, 2005]). However, the use of randomly generated inputs often results in the oracle problem if an appropriate test oracle is unattainable since its creation is associated with an excessive effort. With respect to integration testing, where several units are combined to form a subsystem, partial oracles have to be capable to verify necessary conditions of the complete subsystem in order to be applicable. In comparison with unit testing, these necessary conditions of the complete subsystem might be less restrictive than the conditions of the individual units since they must hold for the complete subsystem. If, for instance, a software unit implements a linear time-invariant system, we can exploit the linearity to define necessary conditions which have to be fulfilled by the implementation. Now, if a subsystem which contains this unit is no longer linear and time-invariant, these necessary conditions do not hold for the integrated system and cannot be used as partial oracles. As a consequence, partial oracles applicable for the complete (sub)system could be less effective than partial oracles constructed for an individual unit of this system. Therefore, the question arises whether partial oracles are in principle adequate in the field of integration testing and, if so, how suitable they are.

This chapter presents a case study of constructing, applying, and assessing partial oracles in order to automatically test several parts of an image processing application. The investigated program is a modular and object-oriented jpeg2000 encoder written in the Java programming language, representing a system with several integrated software units. The study relies on random test data generation and, furthermore, on mutation analysis to assess the effectiveness of the generated inputs as well as the applied partial oracles. It analyzes the adequacy of the chosen partial oracles for unit and integration testing and investigates, additionally, the complexity of the partial oracles as well as possible improvements of their effectiveness and efficiency.

The remainder of this chapter is organized as follows. First, Section 9.2 discusses preliminaries and provides the necessary background. Thereafter, Section 9.3 describes the study and the applied methodology in detail and discusses the corresponding results. This section also highlights the possibilities for effectiveness and efficiency improvements of the applied partial oracles. Finally, Section 9.4 concludes this chapter.

9.2 Preliminaries and Related Work

Concerning the input values, which have to be complete images in our case, it is obviously infeasible to cover all possibilities of the whole input range. Moreover, creating input values manually is time-consuming, particularly for image processing applications which operate on complex input values, and hence not convenient. Thus, we rely on random input generation to create the inputs efficiently since this technique is simple and versatilely applicable in most environments. Various models for generating gray level images exist,

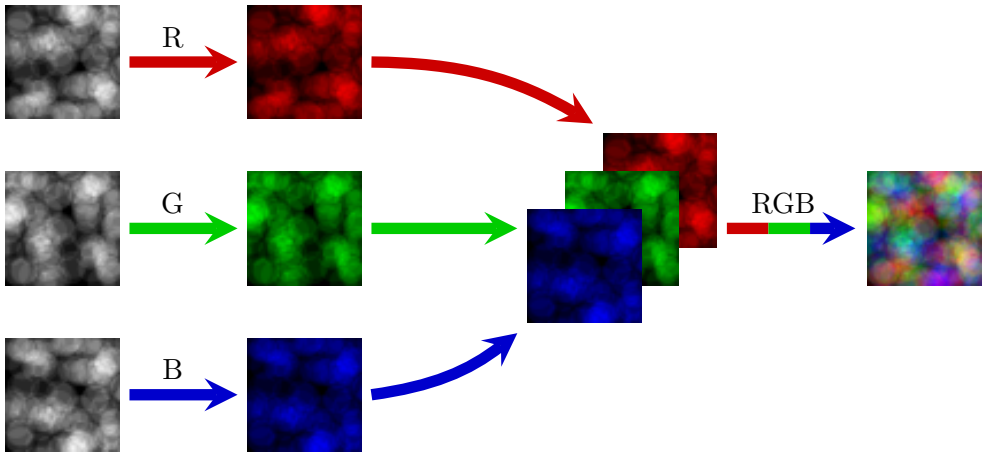


Figure 9.1: Input generation model for color images.

such as the random or boolean model [Guderlei and Mayer, 2007]. This study uses the random model, which determines the gray scale value of each pixel independently.

In order to obtain color images, this study uses and extends the random model for gray-level images, as shown in Figure 9.1. Each color component of the RGB color space is created independently by means of the random model and the complete image is achieved by merging all three components. That means that a gray-scale image is randomly generated for every color component red (R), green (G), and blue (B). Then these images are interpreted as color components, i.e., their gray-scale values represent the according color values of red, green, or blue. Finally, the union of the color components forms the resulting, randomly generated, color image. Generating a gray-scale image randomly is equivalent to computing a matrix in which every coefficient, which represents a pixel, is randomly generated. Since the resulting matrix shall represent a gray-scale image, the values of all coefficients have to be restricted to the interval $[0, 255]$. An example for a gray-scale image and the corresponding matrix that represents the color values of each pixel is illustrated in Figure 9.2.

$$\Longleftrightarrow \quad \begin{bmatrix} 128 & 128 & 128 & 128 & 128 & 128 \\ 128 & 128 & 128 & 128 & 128 & 128 \\ 70 & 70 & 70 & 70 & 70 & 70 \\ 70 & 70 & 70 & 70 & 70 & 70 \\ 170 & 170 & 170 & 170 & 170 & 170 \\ 170 & 170 & 170 & 170 & 170 & 170 \end{bmatrix}$$

Figure 9.2: Gray-scale image and its corresponding coefficient matrix.

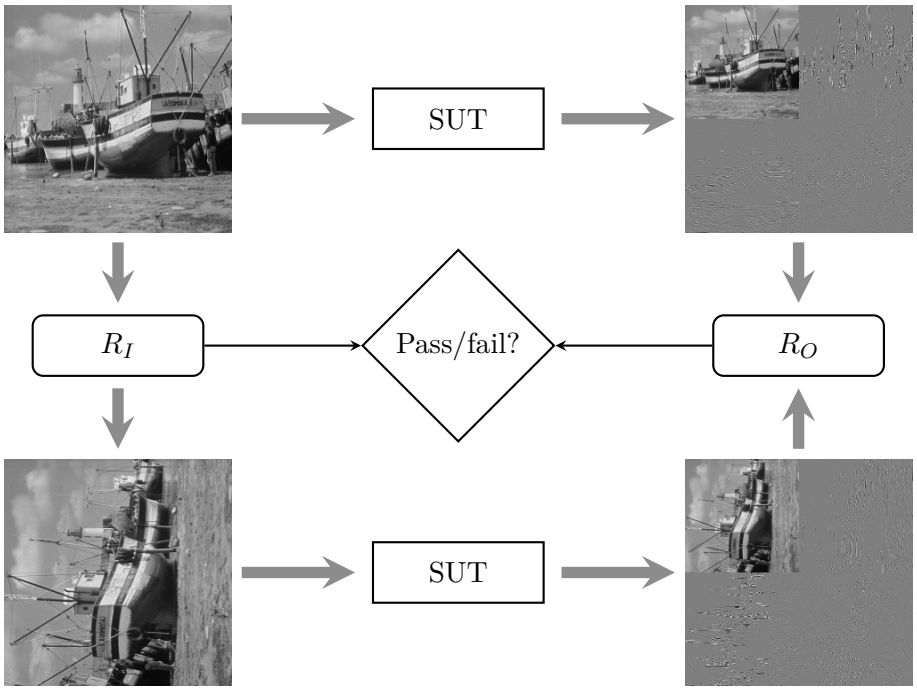


Figure 9.3: Exploiting the commutativity of the two-dimensional Wavelet Transformation as partial oracle by means of the matrix transposition.

With regard to the randomly generated input values, we face the oracle problem which is in our case to be alleviated by means of partial oracles. However, this class of oracles can only check necessary conditions but cannot verify sufficient conditions. Thus, one of the main characteristics of partial oracles is that their results may be false-negative, meaning that if the oracle judges the System Under Test (SUT) to be correct, it may still contain a fault since it only fulfills the necessary conditions. On the other hand, if such an oracle reveals a defect in the SUT, the system definitively contains a fault. It thus becomes clear that assessing the oracles is necessary since they may not detect every fault and the effectiveness of the applied testing strategy depends on the quality of the oracle.

Among others, metamorphic relations, which have been introduced in Chapter 8, are associated with the class of partial oracles. The corresponding workflow for metamorphic testing can be summarized as given below (cf. [Zhou et al., 2004]):

1. Generate a follow-up test case from an arbitrary input according to a relation R_I
2. Execute the SUT independently with both inputs
3. Verify whether the resulting outputs fulfill the corresponding relation R_O

An example for testing with such partial oracles is depicted in Figure 9.3 where the relation between the inputs R_I is the matrix transposition. The relation between the resulting

outputs R_O is also the matrix transposition because of the commutativity of the two-dimensional Wavelet Transformation which is the SUT in this case.

As previously stated, partial oracles may produce false-negatives, and hence reliable results from testing can only be achieved if the employed partial oracles are assessed. For this purpose, we apply mutation analysis to determine the adequacy of partial oracles, more precisely metamorphic relations, for unit and integration testing. In order to kill a mutant, three necessary conditions have to be fulfilled (cf. [Voas, 1992]):

1. The mutated code has to be reached and executed
2. The mutation has to change the state of the program
3. The change has to be propagated to the output

Obviously, the first condition (reachability) is solely related to the input values, apart from dead code fragments, and the latter can be reduced to the question of semantic equivalence. Since a testing strategy consists of a test data generation model and an oracle to evaluate the outputs, the adequacy of a strategy depends on both the quality of the input values and the capability of the oracle. However, the effectiveness of the oracle is correlated with the quality of the input values according to the reachability condition.

Mutation analysis can be employed to assess both parts of the strategy. First, the input values can be evaluated with the original implementation as a perfect oracle. The resulting mutation score S_i for the inputs provides an upper bound for the mutation score of the complete strategy since the perfect oracle (i.e., the best available oracle) is applied in this step. The (partial) oracle of the strategy can then be assessed with the input values which have been determined in the first step. Only the mutants killed by the perfect oracle are used in the second step because all other mutants cannot be killed. It is important to note that the perfect oracle is never applied in this second step because a mutant is said to be killed by the oracle which is to be assessed, if and only if it violates the constraints represented by the applied (partial) oracle. Hence, the mutation score S_o of the investigated oracle represents the number of mutants killed by this oracle related to the number of killable mutants (i.e., mutants killed by the perfect oracle). Now, in order to express the dependency of S_o and S_i, we have to define the overall mutation score S_s for a complete testing strategy as:

Definition 9.1 *Separated Mutation Score S_s*

$$S_s = S_i \cdot S_o$$

Let M_k^p be the number of mutants killed by the perfect oracle with respect to the applied test data generation model. Furthermore, M_t denotes again the total number of non-equivalent mutants of the SUT and M_k represents the number of mutants killed by the employed (partial) oracle. The mutation score S_s is equal to S:

$$S_s = S_i \cdot S_o = \frac{M_k^p}{M_t} \cdot \frac{M_k}{M_k^p} = \frac{M_k}{M_t} = S \quad \square \tag{9.1}$$

The separated evaluation, however, provides a better view on the effectiveness of the individual parts of the testing strategy. Let us consider the following example:

- Number of non-equivalent mutants: 900
- Mutants killed by the perfect oracle: 882
- Mutants killed by the (partial) oracle: 750

By employing Definition 9.1, we obtain the corresponding mutation scores:

$$S_s = \frac{882}{900} \cdot \frac{750}{882} = 0.98 \cdot 0.85 = 0.83 = \frac{750}{900} = S \tag{9.2}$$

In order to increase the effectiveness of the assessed testing strategy it would be advisable in this case to focus on the applied oracle because the test data generation model yields a satisfying result of 0.98. Aiming at evaluating the applied oracle exclusively, reliable results can only be achieved for almost perfect input values, i.e., $S_i \approx 1$.

The suitability of the generated inputs might be assessed with other metrics. The degree of certain code coverage criteria such as statement or branch coverage is, for example, also applicable for this purpose (cf. [Zhu et al., 1997]). Nevertheless, we rely exclusively on mutation analysis in this study. In order to fully automate the executions of the experiments in the case study, proper tool support for the mutation analysis is essential. Throughout the case study, we employ MuJava [Ma et al., 2005; MuJava, 2009], a mutation tool for the Java programming language due to the availability of a wide variety of mutation operators, especially class-based mutation operators.

9.3 Case Study

The selected SUT is a Java implementation of the jpeg2000 encoder which is part of the JJ2000 library [JJ2000, 2010; Jpeg, 2010]. This encoder is a system consisting of several concatenated subsystems which in turn consist of various combined transformations.

The system to be investigated, illustrated in Figure 9.4, is a complex and integrated subsystem which is responsible for preprocessing and decorrelation. The size, measured in Lines of Code (LOC), of the individual parts and the complete subsystem is given in Table 9.1. In addition it should be mentioned that the overall size of the jpeg2000 encoder and the jj2000 library is 14k LOC and 30k LOC, respectively.

Generally, the selected SUT takes, as shown in Figure 9.4, an uncompressed color image as an input value and executes the following workflow:

1. Split color image into color components red (R), green (G), and blue (B)
2. Shift the color values of each color component
3. Transform RGB color components into YCbCr components
4. Decompose components by applying the two-dimensional wavelet transformation

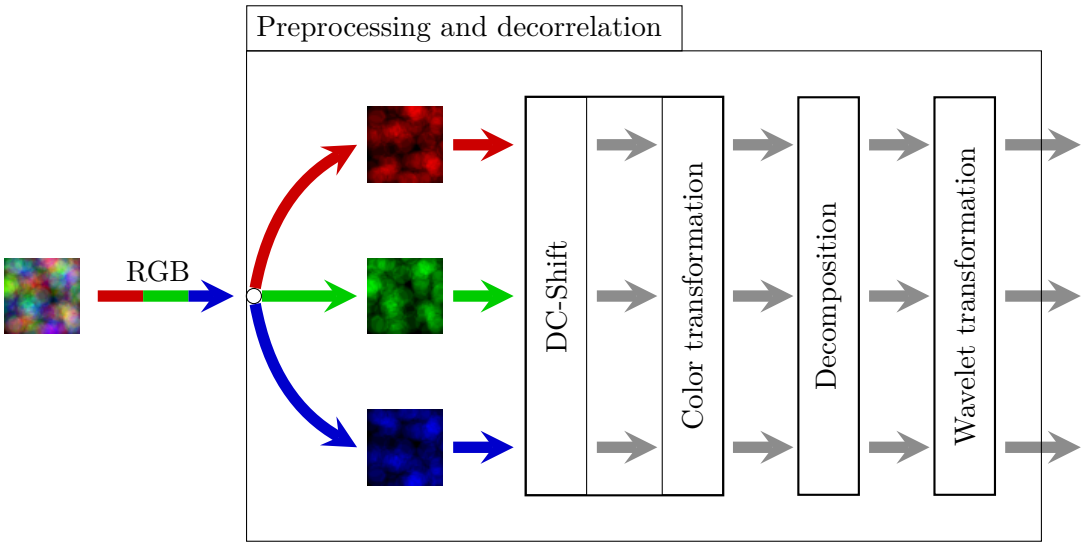


Figure 9.4: The investigated subsystem that is composed of several software units and responsible for preprocessing and decorrelation.

Table 9.1: Software packages and physical lines of code of the individual software units integrated in the investigated subsystem.

Softeware unit	**LOC***	**Package**
DC-Shift/ Color transformation	1,422	jj2000.j2k.image
Decomposition	964	jj2000.j2k.wavelet
Wavelet transformation	2,010	jj2000.j2k.wavelet
Complete subsystem	4,396	jj2000.j2k

*LOC as reported by sloccount (non-comment and non-blank lines)

As a consequence of this workflow, the outputs of the SUT are three individually decomposed color components where each of them contains a DC component (approximation of the color values) and three detail components (differences between pixels). Further background information on the encoder as well as the complete jpeg2000 standard is described in detail, e.g., in [Christopoulos et al., 2000; Skodras et al., 2001].

Generally, the use of randomly generated inputs, as applied in this study, results in the oracle problem for image processing applications. Therefore, handcrafted or well-known standard test images are usually employed for which the expected output can be defined in advance. However, the oracle problem is avoided in our study, as already mentioned in Section 9.1, by means of partial oracles. Aiming at automatically testing this integrated subsystem with partial oracles, the oracles have to be applicable to the complete subsystem. Therefore, the following partial oracles have been chosen.

Partial oracle I:

A constant offset is added to every color value of the input image. Because of this constant offset, the mean color value of each color component changes but the difference between two pixels remains the same. Therefore, only the DC component must be increased (or decreased if the offset is a negative value) by the constant offset.

Partial Oracle II:

The color values of the input image are multiplied by a constant factor. As a consequence, the mean color value as well as the differences between pixels are affected. Thus, the DC component and all detail components have to be changed with respect to the constant factor.

Partial Oracle III:

The input image is transposed by means of the standard matrix transposition. Because of the linearity of the SUT and the commutativity of the two-dimensional wavelet transformation, the resulting components (DC component and detail components) have to be transposed as well.

Partial Oracle IV:

The pixel values of each row within the input image can be regarded as a discrete signal and since the SUT is linear and time-invariant, these signals can be shifted. For this purpose, the image width is enlarged with a defined number (constant for all rows) of leading zeros. Consequently, the resulting components also have to be shifted with unchanged color values.

Partial Oracle V:

Applying an inverted image to the SUT has to result in inverted components due to the linearity of the SUT. Hence, the color values of the input image are inverted and all resulting components have to be affected.

The constraints of these described partial oracles can be defined as metamorphic relations by means of input and output relations. Table 9.2 summarizes all five metamorphic relations with a brief description of the input relation R_I and the corresponding output relation R_O.

Table 9.2: Description of the investigated metamorphic relations.

	Relation	Description
R1	R_I	Add an offset to the color values.
	R_O	Only the DC component must be affected.
R2	R_I	Multiply the color values by a coefficient.
	R_O	Every pixel has to be affected.
R3	R_I	Transpose the pixel array of the input image.
	R_O	The resulting components have to be transposed.
R4	R_I	Enlarge the input image with zero-padding.
	R_O	The resulting components have to be shifted.
R5	R_I	Invert the color values of the input image.
	R_O	The color values of the resulting components have to be inverted.

The metamorphic relations are implemented as matrix transformations. This means that on the one hand the color components of the randomly generated input values are mapped to follow-up matrices (according to R_I). On the other hand, the resulting outputs of the execution of the follow-up test cases are normalized (i.e., they are transformed according to the corresponding relation R_O) and then they are compared with the output of the SUT when executed with the randomly generated input.

Based on the bipartite approach already mentioned in Section 9.2, we investigate the adequacy of these partial oracles for testing purposes with respect to the integrated subsystem. In total, 1,977 non-equivalent traditional mutants can be generated for the complete subsystem. Additionally, 206 non-equivalent class-based mutants can be obtained (e.g., in interfaces between the transformations). The following examples of mutation operators illustrate the difference between traditional and class-based mutants:

Traditional mutants

- `int a = b + c;` $\longmapsto$ `int a = b - c;`
- `if( a && b ){...};` $\longmapsto$ `if( a || b ){...};`

Class-based mutants

- `component.setWidth(5);` ⟹ `component.setHeight(5);`
- `A obj = new A1();` ⟹ `A obj = new A2();`

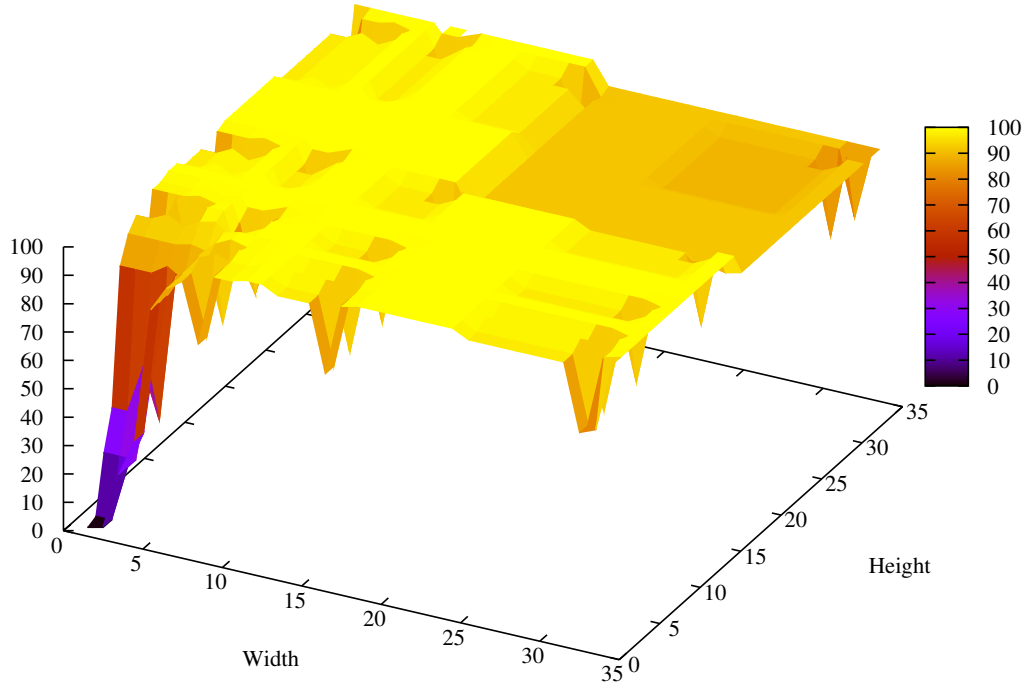
The class-based mutants represent structural defects in contrast to the traditional mutants, which are injected at the functional level. Thus, the class-based mutants can be regarded as faults that could have been introduced by a programmer during the integration of software units. It has to be pointed out that the class-based mutants are obligatory in this study since we aim at assessing the adequacy of partial oracles for integration testing. Instead of the developed mutation tool, presented in Chapter 5, MuJava was applied due to the existing support of class-based mutants. However, it has to be pointed out that the runtime of the tool was significant, which might be a limiting factor for further studies.

9.3.1 Evaluation of the Input Values

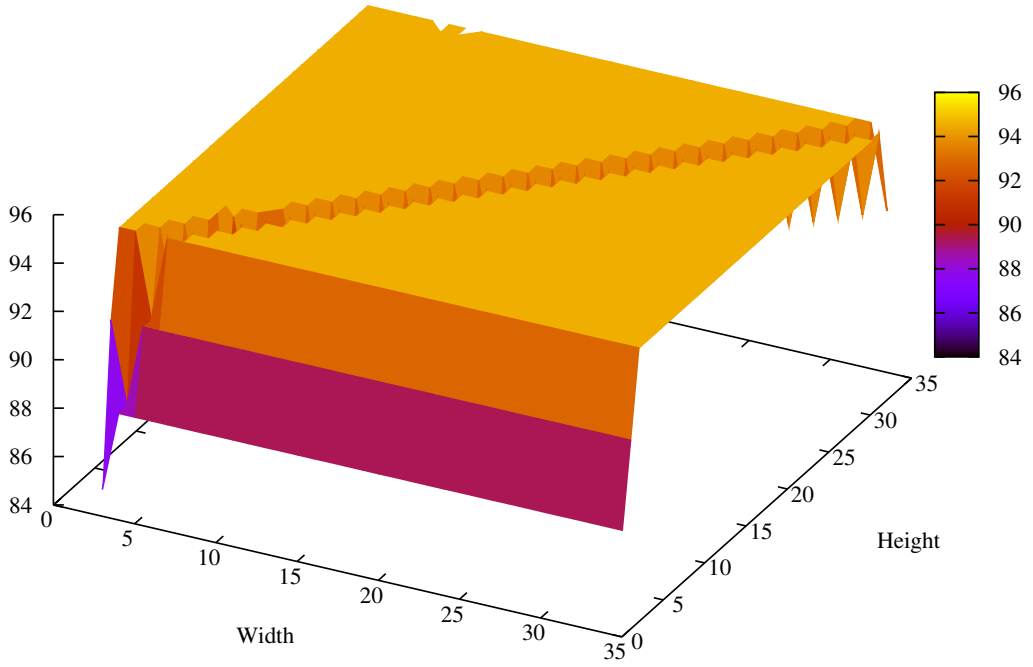
Corresponding to the reachability condition, the mutated code has to be covered in order to be detected. Hence, the effectiveness of the partial oracles is heavily dependent on the input values. Consequently, we assess the input values in a first step, as described in Section 9.2, to provide the most adequate inputs for evaluating the partial oracles. Considering the input values, different properties like the image dimension (i.e., width and height) or the color depth may affect the suitability. However, it turned out that only the image dimension has an impact. Thus, images with different width and height are applied to determine the most appropriate inputs.

In order to achieve the highest possible mutation score for the input values MS_I, we use an exhaustive search over a limited and reduced search space. Therefore, we apply randomly generated images, employ the original implementation as a gold standard oracle, that is a trusted oracle, and use the mutation score as fitness function. The remaining mutants not killed after the search are inspected to eliminate the equivalent mutants and to determine the correct mutation score. This task is done manually but approaches exist that could be used to automatically identify certain types of equivalent mutants (cf. [Offutt and Pan, 1997]). Two examples for resulting fitness landscapes are shown in Figure 9.5(a) and 9.5(b). The input values are classified according to their effectiveness, that is the mutation score. This means that all images within a certain equivalence class yield exactly the same mutation score when employing them as input value and applying the perfect oracle. More precisely, the same mutants are killed by all images of the same equivalence class. Considering, for example, the fitness landscape of the Decomposition in Figure 9.5(b), we could identify exactly three equivalence classes in this case:

1. width $<$ height
2. width $=$ height
3. width $>$ height

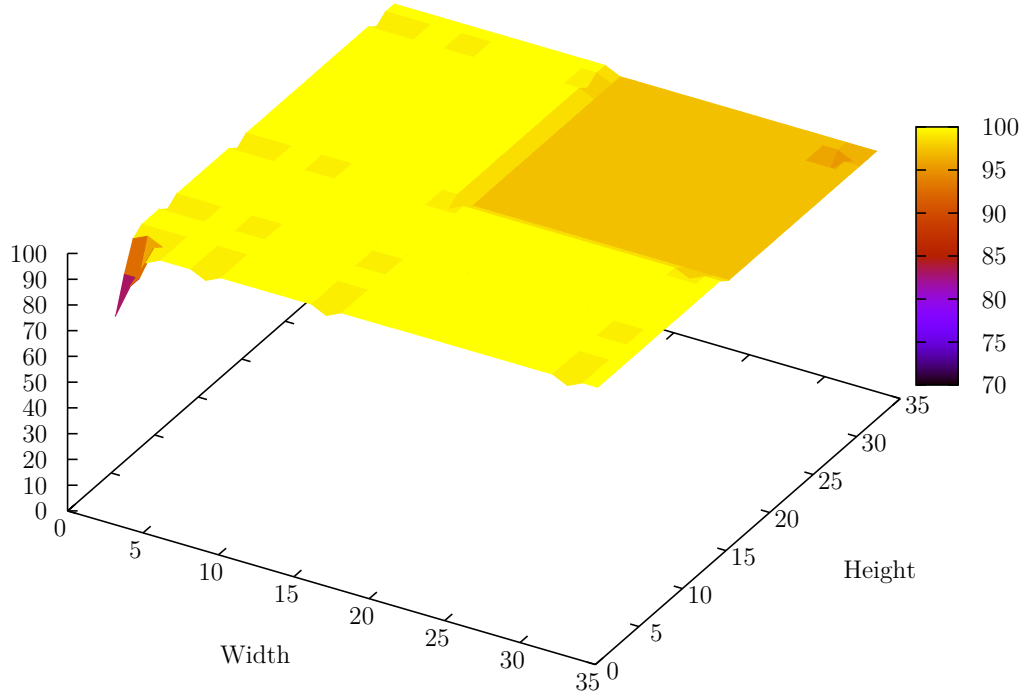


(a) Quadratic mutation score for the Wavelet transformation.

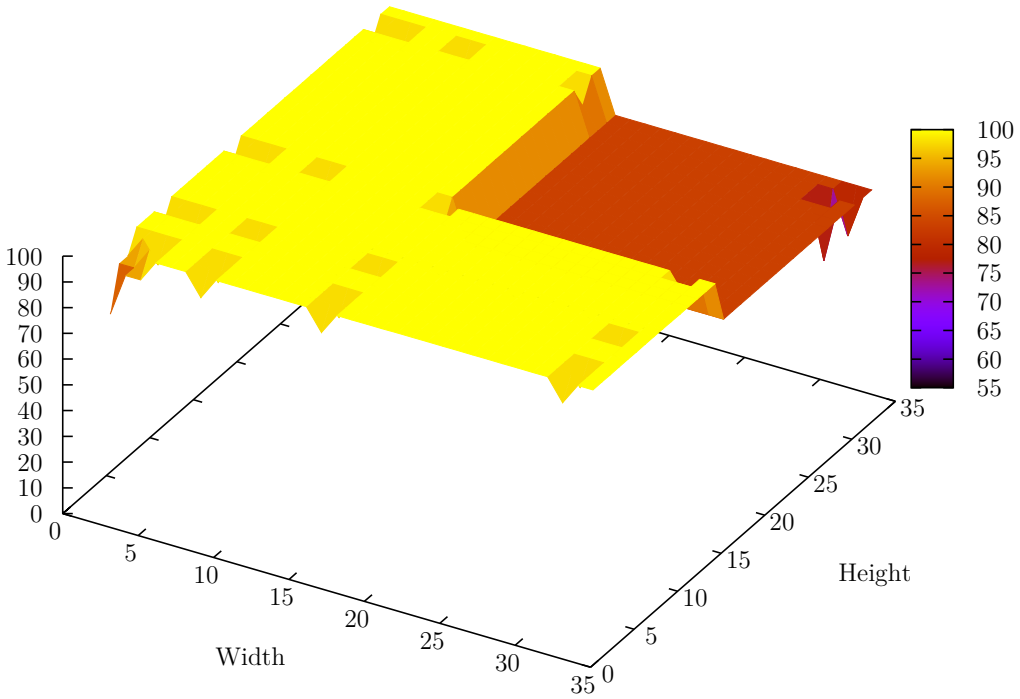


(b) Mutation score for the Decomposition.

Figure 9.5: Fitness landscapes of the mutation score related to the image dimensions for the Wavelet transformation and the Decomposition.



(a) Statement coverage.



(b) Branch coverage.

Figure 9.6: Fitness landscapes of statement and simple branch coverage for the Wavelet transformation, as reported by Cobertura [2010].

However, no class achieves a mutation score of 100%, indicating that several runs of the SUT are necessary to collectively reach the full mutation score. In general, if the input images related to the most effective class are sufficient to kill all non-equivalent mutants, then a conjunction of multiple classes, which implies several runs of the SUT, is not necessary. Otherwise the classes have to be combined to collectively kill all of the mutants. As shown in Figure 9.5(a), multiple executions are not necessary in this case since input images exist which achieve a mutation score of 100%.

As already mentioned in Section 9.2, another metric could be applied as a fitness function in this first step. For this reason, we investigate two code coverage metrics as fitness function in addition to the mutation analysis. We use statement and simple branch coverage within the same search space. An advantage of both code coverage metrics is that of their being less expensive than mutation analysis with regard to the necessary runtime to determine the level of coverage. For every input value the SUT has to be evaluated just once. However, it turns out that both metrics are weaker criteria than mutation analysis. Considering the three conditions to kill a mutant, the first one (reachability) is obviously a necessary condition for all three metrics. In contrast to the mutation analysis this condition is, however, also sufficient for the code coverage metrics. Since we apply all mutation operators, available within the mutation tool, every basic block and expression is mutated. In addition, almost every statement is mutated. Thus, mutation analysis implies both code coverage criteria in this case and is according to this fact the stronger criterion. Example results of the exhaustive searches are depicted in Figure 9.6(a) and 9.6(b). Both diagrams manifest that the investigated coverage criteria achieve quite often a level of coverage of 100%. By comparing these results to the mutation score, shown in Figure 9.5(a), it becomes clear that input values exist that cover all statements and branches but they are not able to kill all of the mutants.

It has to be pointed out that search-based techniques [McMinn, 2004] may be more efficient for larger search spaces. Additionally, the exhaustive approach is no longer feasible for huge input domains. Since the location of the mutation within the source code is exactly known, the problem of searching adequate inputs that cover the mutation can be transformed into a path problem (cf. [Visser et al., 2004; Fraser and Arcuri, 2011]).

9.3.2 Evaluation of the Partial Oracles

We now investigate the partial oracles with regard to both the capability to reveal faults in each transformation and the overall effectiveness for the complete subsystem. It is important to note that roughly 40% of the traditional and 20% of the class-based mutants throw a runtime exception, when the corresponding mutated code is covered, mostly because of invalid array indexes or references. Due to the restrictions of the Java virtual machine, which lead to a thrown exception in such cases [Lindholm and Yellin, 2000], these faults could thus be revealed without a particular oracle. However, all partial oracles catch these exceptions and mark the corresponding mutants as killed in this study. Consequently, the number of detected faults includes those mutants killed by an exception.

Table 9.3: Effectiveness of the applied partial oracles for the particular transformations and the complete subsystem concerning the traditional mutants.

	DC-shift/Color transformation	**Decomposition**	**Wavelet transformation**	**Complete subsystem**
	608	441	928	1977
R1	504 (82.9%)	402 (91.2%)	860 (92.7%)	1766 (89.3%)
R2	489 (80.4%)	381 (86.4%)	859 (92.6%)	1729 (87.5%)
R3	482 (79.3%)	413 (93.7%)	904 (97.4%)	1799 (91.0%)
R4	394 (64.8%)	398 (90.3%)	876 (94.4%)	1668 (84.4%)
R5	456 (75.0%)	372 (84.4%)	781 (84.2%)	1609 (81.4%)

The effectiveness of the partial oracles varies notably with respect to a specific transformation, as illustrated in Table 9.3. Concerning the DC-Shift in conjunction with the Color Transformation, the relation `R1`, which kills 504 mutants, is for instance much more effective than `R4`, which reveals only 394 defects. In reference to the total number of 608 non-equivalent mutants, this represents a considerable discrepancy in the mutation score between 83% and 65%. In addition, the effectiveness of the applied oracles differs with regard to the specific parts of the subsystem and thus, none of them is adequate for all transformations. Considering, for example, the DC-Shift and Color Transformation, the relation `R1` yields the highest mutation score for this part of the subsystem. However, in the field of the Wavelet Transformation, the effectiveness of `R1` is only average in comparison with the other relations.

Furthermore, the different kinds of mutants exhibit a substantial difference in effectiveness, as shown in Table 9.4. The relation `R3` is for instance the most effective oracle concerning the traditional mutants with an overall mutation score of 91% and ranging between 80% and 97% with regard to the individual transformations. It is, however, rather poor in killing the class-based mutants with a ratio of approximately 58%. Hence, it would be a fallacy to conclude that an oracle which is highly effective for testing a specific transformation, or more generally an individual unit of a subsystem, is as a consequence of this equally suitable for integration testing. Moreover, the effectiveness of the applied relations is insufficient concerning the class-based mutants, even though the results achieved by the relations are predominantly sufficient with respect to the traditional mutants.

In order to increase the effectiveness and achieve satisfying results, the partial oracles may be combined (cf. [Mayer and Guderlei, 2006]). Therefore, the two most effective oracles are combined pairwise and additionally the overall effectiveness of the partial oracles altogether is investigated. A combination in this case means that all of the combined partial

Table 9.4: Comparison of the effectiveness of the applied partial oracles for the complete subsystem with respect to the different kind of mutants.

	Traditional mutants	**Class-based mutants**
	1,977	206
R1	1,766 (89.3%)	151 (73.3%)
R2	1,729 (87.5%)	146 (70.9%)
R3	1,799 (91.0%)	119 (57.8%)
R4	1,668 (84.4%)	84 (40.8%)
R5	1,609 (81.4%)	87 (42.2%)

Table 9.5: Increase in effectiveness of the applied partial oracles by means of combination of the two most effective oracles (R_x and R_y) and all oracles.

(a) Increase in effectiveness for the traditional mutants.

	Total	$\mathbf{R_x}$	$\mathbf{R_y}$	$\mathbf{R_x}$ **&** $\mathbf{R_y}$	**All Oracles**
DC-shift/Color transformation	608	R1	R2	544 (89.5%)	553 (91.0%)
Decomposition	441	R3	R1	422 (95.7%)	430 (97.5%)
Wavelet transformation	928	R3	R4	923 (99.5%)	923 (99.5%)
Complete subsystem	1,977	R3	R1	1,889 (95.6%)	1,906 (96.4%)

(b) Increase in effectiveness for the class-based mutants.

	Total	$\mathbf{R_x}$	$\mathbf{R_y}$	$\mathbf{R_x}$ **&** $\mathbf{R_y}$	**All Oracles**
Complete subsystem	206	R1	R2	193 (93.7%)	200 (97.1%)

oracles are applied individually and the mutation score collectively achieved equates to the effectiveness of the combined oracles. The results for each transformation as well as for the complete subsystem are depicted in Table 9.5 where R_x and R_y represent the most effective and the second most effective oracle with respect to the corresponding transformation. Regarding, for instance, the Wavelet transformation, `R3` is the most effective oracle and `R4` is the second most effective one. Thus, the column "R_x & R_y" denotes the mutation score collectively achieved by the combination of these both oracles.

The results clearly show that the combination of the partial oracles can significantly increase their effectiveness, especially with regard to the class-based mutants. The conjunction of the two most effective relations leads to a mutation score of 94% for the class-based mutants and an overall mutation score of 95% for all traditional mutants. In addition, the variance is reduced significantly since all ratios are at least 90%.

On the other hand, the additional benefit of combining all partial oracles is rather small compared with the pairwise combination of the two most effective ones. The necessary effort of applying two more oracles is disproportionate compared to the further increase of at most 2% for the traditional and 3% for the class-based mutants.

9.3.3 Efficiency and Effectiveness Improvements

Considering the complexity of the partial oracles, we can distinguish between code and model complexity. Metrics for the code complexity are, for instance, physical lines of code or McCabe's cyclomatic complexity [McCabe, 1976], which is a measure for the structural complexity. Based on the control flow graph G, let n be the number of nodes, e the number of edges, and p the number of connected components. McCabe's cyclomatic complexity $C(G)$ is then defined as:

$$C(G) = e - n + 2 \cdot p \tag{9.3}$$

The physical lines of code as well as McCabe's cyclomatic complexity are indicators for the effort to understand and implement the corresponding partial oracle.

On the other hand we use the term model complexity to describe the costs of applying the partial oracle. The model complexity is predominantly defined by the number of parameters and additionally by the number of necessary inputs, which is equal to the number of required runs of the SUT. Obviously, the number of parameters is more severe since the partial oracle has to be applied by executing the SUT for every parameter value which is to be investigated. Considering for example the oracle `R2` which can be described by the following equation:

$$\underbrace{SUT(c \cdot I)}_{run\#1} = c \cdot \underbrace{SUT(I)}_{run\#2} \tag{9.4}$$

The only parameter of this oracle is the factor c which has to be chosen. In addition, the SUT has to be executed twice in order to apply this oracle. Thus, the partial oracle expects the parameter c and two inputs, namely I and $c \cdot I$. It is crucial to state that the difference between parameters and inputs is of particular importance. The oracle needs to

Table 9.6: Complexity and effectiveness of the applied partial oracles (Params denotes the number of parameters of the corresponding partial oracle and Inputs represents the number of necessary runs of the SUT).

	LOC*	McCabe		Params	Inputs	Mutation Score	
		R_I	R_O			traditional	class-based
R1	363	17	11	1	2	89.3%	73.3%
R2	352	17	6	1	2	87.5%	70.9%
R3	327	10	5	0	2	91.0%	57.8%
R4	398	20	11	1	2	84.4%	40.8%
R5	331	17	6	0	2	81.4%	42.2%

*LOC as reported by sloccount (non-comment and non-blank lines)

be calibrated if it contains a parameter and the choice of the parameter value may have an impact on the effectiveness. Table 9.6 gives the code and model complexity, as well as the overall mutation score of the investigated oracles.

Given the complexity and effectiveness in terms of the mutation score of the employed partial oracles, we focus on the oracles `R1` and `R3` for further improvements of effectiveness and efficiency since they achieve the highest mutation score for the class-based and traditional mutants, respectively. In order to avoid the additional parameter of `R1`, the constant offset, we can generalize this partial oracle by adding a randomly generated offset to each coefficient. For this purpose, the new partial oracle `R6` generates another random image I' with the same dimension as the input I and adds both inputs together with the standard matrix addition. Because of the linearity of the SUT, the necessary condition which has to be fulfilled by the SUT can be described by the following equation:

$$\textbf{R6:}\quad \underbrace{SUT(I + I')}_{run\#1} = \underbrace{SUT(I)}_{run\#2} + \underbrace{SUT(I')}_{run\#3} \tag{9.5}$$

Therefore, the additional parameter has been replaced by another input and thus an extra run of the SUT. Moreover, the implementation of R_O for `R6` is even simpler compared with `R1` because we do not have to locate the DC component in the outputs.

According to the increase in effectiveness by combining partial oracles, a combination of the oracles `R3` and `R6` seems to be promising for further improvements. With respect to efficiency, especially execution time, we can again exploit the linearity and furthermore the commutativity of the SUT. To combine both necessary conditions of `R3` and `R6` within one oracle, we define a new partial oracle `R7`:

$$\textbf{R7:}\quad \underbrace{SUT((I + I')^T)}_{run\#1} = (\underbrace{SUT(I)}_{run\#2} + \underbrace{SUT(I')}_{run\#3})^T \tag{9.6}$$

Table 9.7: Complexity and effectiveness of enhanced partial oracles (Params denotes the number of parameters of the corresponding partial oracle and Inputs represents the number of necessary runs of the SUT).

	LOC*	McCabe		Params	Inputs	Mutation Score	
		R_I	RO			traditional	class-based
R6	368	18	9	0	3	93.2%	88.8%
R7	391	24	13	0	3	96.3%	96.6%

*LOC as reported by sloccount (non-comment and non-blank lines)

The model complexity of this partial oracle is equal to that of `R6` because we do not have additional parameters and the required number of executions of the SUT is still 3. Hence, this oracle leads to an efficiency improvement by reducing the runtime by 40% in comparison with single executions of `R3` and `R6`. Table 9.7 shows the complexity and the mutation score of the enhanced partial oracles. Overall, both oracles manifest a higher code and model complexity due to the additional input and the more comprehensive calculation. They achieve, however, a significant increase in effectiveness for the traditional and the class-based mutants. Concerning the traditional mutants, the variance of the mutation score, which ranges between 91.4% and 98.8% for the individual software units, is also considerably reduced. Hence, these partial oracles can be regarded as being suitable for the SUT.

9.3.4 Discussion

Regarding the results, it seems that partial oracles are indeed applicable for integration testing, but a few aspects have to be considered. First of all, the partial oracles derived from the characteristics of the integrated (sub)system may be less effective than partial oracles for the individual software units of this system. In addition, the effectiveness of oracles for testing at the functional level of individual system parts most likely differs from the effectiveness for testing the integration of these parts. In order to compensate such variations and to increase the effectiveness, it is advisable to combine partial oracles.

As shown in the case study, combining the most effective oracles is nearly as powerful as joining all oracles. Thus, the testing effort can be reduced without a major loss of effectiveness by prioritizing the partial oracles with regard to their fault-finding capability and joining just the most effective ones. Since the SUT has to be executed for every partial oracle and every input value, the time needed to run all of the tests is proportional to the number of partial oracles. Concerning the investigated subsystem, processing the input values and executing the SUT is a time-consuming task and thus applying only two out of four relations notably reduces the time totally needed. This decrease of the execution

time, and hence testing effort, may be of particular importance in the field of regression testing.

According to our results of the efficiency and effectiveness improvements, some suggestions on the selection and construction of partial oracles can be given. First of all, it is advisable to exploit constraints like equivalence relations in conjunction with properties such as commutativity, distributivity, or associativity. For efficiency reasons the combination of necessary conditions should be implemented within one partial oracle even though the model and code complexity of the corresponding oracle is increasing. Additional parameters should be avoided or kept to a minimum since they extend the search space. Moreover, with respect to automated (adaptive) random testing, a partial oracle like `R7` would be preferable (e.g., via prioritization) because it generates a follow-up value with different properties (cf.[Chen et al., 2004]). As a consequence, the input values are better distributed in the search space and the convergence rate of the mutation score is most likely higher.

Since the investigated SUT is a Java implementation, there are many mutants that result in an exception due to violating restrictions. As previously mentioned in Section 9.3.2, these mutants can be killed without a specific oracle, for instance with smoke tests. Considering the properties of partial oracles, such smoke tests can be regarded as the simplest partial oracle, which checks for the necessary condition that the system under test should not crash during execution. As a consequence, partial oracles can be implicitly combined with such smoke tests, for instance, by means of an adequate exception handling, or smoke tests can be used in a first step to reveal invalid indexes and references. However, this is language dependent and may be less suitable for other languages.

With regard to the discussed results, some threats to validity have to be considered. The chosen mutation operators could be a threat to internal validity. Different operators or hand seeded faults may affect the mutation score of the investigated partial oracles. However, we applied all possible operators, provided by the mutation tool, in order to cover a wide variety of defects. Furthermore, most of the applied operators are frequently used in the literature and therefore provide comparable results [Andrews et al., 2005]. A potential threat to external validity might be the representativeness of the selected application. There is no guarantee that the depicted results and the achieved improvements of effectiveness and efficiency of the partial oracles will be similar for other systems. The investigated subsystem, nevertheless, represents a modular object-oriented application with several integrated units, and hence is comparable to other software systems. So, the reported results might be transferable but a replication of this study, especially for other programming languages and larger software systems, is necessary. This matter is left open for future research and further discussed in Chapter 10. However, it is important to note that the presented case study clearly reveals that in general the effectiveness of partial oracles for integration testing cannot be derived from their adequacy with regard to unit testing. Defects in the mutation tool or in our testing framework could be a threat to construct validity, but we controlled this threat by analyzing the generated mutants and by testing our implementation. Every partial oracle was applied to the original implementation and executed with all input values to ensure that the implemented constraints

are fulfilled by the investigated system. Thus, we judge that the mutants were properly generated and that our implementation worked correctly.

9.4 Summary

This chapter has studied the adequacy of partial oracles for integration testing. In order to evaluate the applicability of partial oracles for this purpose, an integrated subsystem of an object oriented image processing application is investigated by means of mutation analysis. The applied partial oracles are assessed with regard to their capability to reveal faults in the individual parts of the subsystem and their suitability for the integrated subsystem in its entirety. It turns out that the effectiveness of the investigated partial oracles varies concerning the different parts of the subsystem and none of the oracles is sufficient for the complete subsystem. Additionally, the adequacy of the partial oracles for integration testing cannot be inferred from the effectiveness for testing the particular parts of the subsystem. However, the combination of the partial oracles yields satisfying results for both unit and integration testing. Moreover, exploiting certain characteristics of the system under test provides partial oracles which lead to a significant increase in effectiveness and efficiency. Hence, this kind of oracles seems to be suitable for testing purposes and especially test automation with respect to the oracle problem.

In summary, the results suggest that partial oracles are suitable for automating various parts of the software testing process but further research is necessary to confirm and generalize the findings. In addition, the results should be transferable to other linear and time-invariant systems as well as transformations which meet the exploited constraints. However, examining the transferability to other partial oracles as well as different software systems is left open for future work.

Part IV

Conclusions and Appendix

Chapter 10

Conclusion and Future Work

This chapter concludes the thesis by summarizing the main contributions. Additionally, it discusses possible extensions and areas for future research.

10.1 Conclusions

This thesis makes several contributions to the ongoing research in the field of mutation testing and testing with partial oracles.

Effective and Efficient Mutation Analysis

With regard to the improvements of mutation analysis, this thesis first presented conditional mutation, an approach to mutant generation that transforms the abstract syntax tree and embeds mutants for expressions and statements within the scope of the original instruction. Due to the embedded mutants and several optimizations, conditional mutation enables efficient mutation analysis.

The corresponding implementation Major enhances the Java Standard Edition compiler and is, thus, applicable in every Java-based environment. The runtime overhead of this enhanced compiler is negligible in consideration of the total number of generated mutants. Moreover, Major provides its own domain specific language Mml that makes it highly configurable and extensible.

Focusing on the definition of mutation operators, we could identify redundancies in two frequently used mutation operators and we also presented a non-redundant version. Employing Major for several evaluations, this thesis reveals that applying the non-redundant mutation operators considerably improves the runtime of mutation analysis. Moreover, the non-redundant set of mutants yields a more precise mutation score, and hence improves the expressiveness of this metric.

Due to the fact that the non-redundant mutation operators still generate a substantial number of mutants, this thesis further analyzed runtime improvements that do not rely on the reduction of mutants. By investigating existing test suites of several real-world projects, we ascertained some essential characteristics such as test case overlap and runtime and ultimately suggested an optimized workflow for the mutation analysis of comprehensive test suites for large software systems.

To sum up, the combination of the suggested approaches and the corresponding implementation Major has resulted in an efficient, scalable, and thoroughly-studied framework for the mutation analysis of Java programs.

The presented and implemented techniques within this thesis, enabled us to perform mutation analysis on several large real-world projects. To the best of our knowledge, these were the largest studies on mutation analysis to date. Given the presented efficiency and effectiveness improvements, the results suggest that mutation testing is indeed ready for being transferred and wider integrated into industry.

Testing with Partial Oracles

Concerning the oracle problem, partial oracles seem to be an applicable solution for unit and integration testing. By employing mutation analysis, this thesis has conducted a study that evaluates the adequacy of partial oracles for both testing levels. Yet, the presented results reveal that the suitability of partial oracles for integration testing cannot be derived from their adequacy for unit testing. Based on these findings, this thesis, furthermore, investigated possible efficiency and effectiveness improvements, which finally led to partial oracles that exhibit comparable adequacy for unit and integration testing.

10.2 Future Work

This section suggests several possible extensions of the work described in this thesis. While some of the aspects have already been considered in this thesis but were not fully evaluated, others represent areas for further research. By reconsidering the contributions of this work, we suggest extensions of the presented approaches and implementations. Furthermore, we discuss possible empirical research to further evaluate and generalize the results.

Conditional Mutation

The conditional mutation approach could be further enhanced with respect to the runtime by balancing the abstract syntax tree with further conditional expressions and statements. This would decrease the necessary evaluations and thus the runtime overhead – the space overhead would increase, though. Moreover, further investigations and optimizations of higher order conditional mutation is another direction for future research.

With regard to the necessary conditions to kill a mutant, side-effect free expression-based mutations do not have to be analyzed when the corresponding expression value does not differ from the original version. Conditional mutation can be enhanced to support the

verification of this necessary condition by extending the mutation coverage implementation. This extension might be in particular useful for mutants with a minimal impact, like those discussed in Chapter 6.

Mutation Analysis Framework

Major, the mutation tool presented in Chapter 5 currently supports a commonly used set of traditional mutation operators, which are configurable by means of the domain specific language Mml. Implementing new mutation operators, especially class-based ones would enable further experiments and the replication and extension of previous, possibly limited, studies. This is, for instance, essential when aiming at an extension of the study on partial oracles that was presented in Chapter 9. Besides the implementation of the existing class-based mutation operators, Mml could be enhanced and generalized to support the customization and definition of operators based on arbitrary transformations of the abstract syntax tree.

Redundancies in Mutation Operators

Because of the considerable improvements reported in Chapter 6, a determination of sufficient sets of replacements for other mutation operators is desirable. For instance, the mutation operator that replaces arithmetic operators forms a substantial proportion of the total number of generated mutants. Hence, eliminating redundancies in this mutation operator would further improve the efficiency and also increase the precision of the mutation score. With regard to composed conditional expressions, the establishment of a subsumption hierarchy for expressions with more than one logical connector is another area for extensions.

Equivalent Mutant Problem

Concerning the equivalent mutant problem, further investigations are necessary to determine whether mutants with a minimized impact, like the ones studied in Chapter 6, are more or less likely to be equivalent. Furthermore, due to the minimal impact of those mutants, a certain combination of the input values is necessary to detect them. Therefore, static analysis techniques might be applicable to determine whether this combination is feasible. The crucial part of this extension is the fact that the necessary (path) constraints need to be efficiently extracted and solved.

Test Suite Prioritization

The splitting approaches suggested in Chapter 7 are parameterized with a threshold that is currently determined with a heuristic based on the test class initialization time. This technique could benefit from taking the overlap at the method level into account to only split methods with a sufficiently small overlap. Moreover, to better assess the quality of the heuristics, the optimally sorted test suite could be calculated. This, however, implies a tremendous computational expense since every test case has to be executed for every covered mutant to identify the fastest test case that not only covers but also kills the mutant.

Empirical Research

As already indicated in the empirical studies, we cannot generally claim that the results are transferable to all types of software systems, even though the investigated applications differ in terms of size, complexity, and operational purpose. Most of the applications employed to evaluate the proposed techniques in this thesis have been used in other empirical studies, and hence provide meaningful and comparable results. Nevertheless, the applications were not randomly selected and could therefore miss boundary cases. More recently, Fraser and Arcuri [2012a] randomly selected 100 open-source applications to enable sound empirical studies. Hence, the techniques presented in this thesis as well as the developed mutation tool could be applied to this set of applications in order to obtain more generalizable results and, moreover, a more precise mean value of the improvements in terms of effectiveness and efficiency.

Concerning the testing with partial oracles, an extension of the case study presented in Chapter 9 is desirable to confirm, extend, and generalize the results. This is, however, not feasible with the current setup due to the limitations of the applied mutation testing tool. Yet, implementing the class-based mutation operators into Major will enable further research and considerably larger empirical studies in this field.

Appendix A

Major mutation language Grammar

This appendix provides the grammar for the domain spcific language Mml. It is important to note that the implementation details, such as symbol table or error handling, have been omited in order not to confuse the visualization of the lexer and parser rules. For the sake of clarity, the grammar is devided into the following three categories:

- Terminal symbols for keywords, operators, and special characters
- Lexer rules of Mml
- Parser rules of Mml

```
1  BIN          = 'BIN'    ;
2  UNR          = 'UNR'    ;
3
4  ORU          = 'ORU'    ;
5  LVR          = 'LVR'    ;
6  AOR          = 'AOR'    ;
7  COR          = 'COR'    ;
8  ROR          = 'ROR'    ;
9  SOR          = 'SOR'    ;
10 LOR          = 'LOR'    ;
11
12 LHS          = 'LHS'    ;
13 RHS          = 'RHS'    ;
14 FALSE        = 'FALSE'  ;
15 TRUE         = 'TRUE'   ;
```

Listing A.1: Mutation keywords of Mml.

```
1  AND                    =  '&&'   ;
2  ARROW                  =  '->'   ;
3  ASSIGN                 =  '='    ;
4  AT                     =  '@'    ;
5  BIT_AND                =  '&'    ;
6  BIT_OR                 =  '|'    ;
7  BIT_SHIFT_R            =  '>>>'  ;
8  COLON                  =  ':'    ;
9  COMMA                  =  ','    ;
10 DEC                    =  '--'   ;
11 DELIM                  =  '"'    ;
12 DIV                    =  '/'    ;
13 DOLLAR                 =  '$'    ;
14 DOT                    =  '.'    ;
15 EQUAL                  =  '=='   ;
16 GE                     =  '>='   ;
17 GT                     =  '>'    ;
18 INC                    =  '++'   ;
19 LBRACK                 =  '['    ;
20 LCURLY                 =  '{'    ;
21 LE                     =  '<='   ;
22 LT                     =  '<'    ;
23 LOGICAL_NOT            =  '!'    ;
24 LPAREN                 =  '('    ;
25 MINUS                  =  '-'    ;
26 MOD                    =  '%'    ;
27 MUL                    =  '*'    ;
28 NOT                    =  '~'    ;
29 NOT_EQUAL              =  '!='   ;
30 OR                     =  '||'   ;
31 PLUS                   =  '+'    ;
32 RBRACK                 =  ']'    ;
33 RCURLY                 =  '}'    ;
34 RPAREN                 =  ')'    ;
35 SEMI                   =  ';'    ;
36 SHIFT_L                =  '<<'   ;
37 SHIFT_R                =  '>>'   ;
38 UNDERSCORE             =  '_'    ;
39 XOR                    =  '^'    ;
```

Listing A.2: Operators and special characters used by Mml.

```
IDENT_START : ('a'..'z' | 'A'..'Z' | UNDERSCORE)
            ;

IDENT_REST  : ('0'..'9')
            ;

IDENT       : IDENT_START (IDENT_START | IDENT_REST)*
            ;

INT_LIT     : '0'
            | '1'..'9' ('0'..'9')*
            ;

FLOAT_LIT   : '1'..'9'+ DOT '0'..'9'*
            | '0' DOT '0'..'9'*
            | DOT '0'..'9'+
            ;

COMMENT     : '//' ~('\n'|'\r')* ('\r\n' | '\r' | '\n')?
            ;

WS          : (' ' | '\r' | '\t' | '\u000C' | '\n' )
            ;

INIT_NAME   : LT ('init' | 'clinit') GT
            ;
```

Listing A.3: Lexer rules of Mml.

```
1  script     : stmt* ;
2
3  stmt       : ( | vardef | call | repl_stmt) SEMI
4             | opdef
5             | COMMENT ;
6
7  repl_stmt  : prefix mut_kind LPAREN op RPAREN
8               stmt_scope ARROW (op_list | IDENT) ;
9
10 prefix     : | PLUS | LOGICAL_NOT ;
11
12 stmt_scope : | LT (IDENT | DELIM flatname DELIM) GT ;
13
14 flatname   : IDENT (DOT IDENT)*
15              (DOLANDR (IDENT | INT_LIT))*
16              (AT (IDENT | INIT_NAME))? ;
17
18 mut_kind   : BIN | UNR ;
19
20 mut_op     : (PLUS | MINUS)?
21              (AOR | LOR | COR | ROR | SOR | ORU | LVR) ;
22
23 op_list    : LCURLY op (COMMA op)* RCURLY ;
24
25 op         : MUL | DIV | MOD | PLUS | MINUS    // AOR/ORU
26            | LT | LE | GT | GE | EQUAL | NOT_EQUAL // ROR
27            | AND | OR                          // COR
28            | XOR | BIT_AND | BIT_OR            // LOR
29            | SHIFT_L | SHIFT_R | BIT_SHIFT_R   // SOR
30            | NOT | LOGICAL_NOT                 // ORU
31            | LHS | RHS | FALSE | TRUE ;        // ROR/COR
32
33 vardef     : IDENT ASSIGN (DELIM flatname DELIM | op_list);
34
35 opdef      : IDENT LCURLY (op_block)* RCURLY ;
36
37 op_block   : prefix mut_kind LPAREN op RPAREN ARROW
38              ( op_list | IDENT ) SEMI
39            | ( | mut_op | IDENT) SEMI | COMMENT ;
40
41 call       : (mut_op | IDENT) stmt_scope ;
```

Listing A.4: Parser rules of Mml.

Bibliography

[Acree et al., 1979]
Acree, A. T., Budd, T. A., DeMillo, R. A., Lipton, R. J., and Sayward, F. G. Mutation analysis. techreport GIT-ICS-79/08, Georgia Institute of Technology, Atlanta, Georgia, 1979.

[Ammann and Offutt, 2008]
Ammann, P. and Offutt, J. *Introduction to Software Testing.* Cambridge University Press, 1 edition, 2008.

[Andrews et al., 2005]
Andrews, J. H., Briand, L. C., and Labiche, Y. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 402–411, 2005.

[Andrews et al., 2006]
Andrews, J. H., Briand, L. C., Labiche, Y., and Namin, A. S. Using mutation analysis for assessing and comparing testing coverage criteria. IEEE Transactions on Software Engineering, volume 32(8):608–624, 2006.

[Bertolino, 2007]
Bertolino, A. Software testing research: Achievements, challenges, dreams. In *Future of Software Engineering*, FOSE '07, pages 85–103, 2007.

[Binder, 1999]
Binder, R. *Testing Object-Oriented Systems: Models, Patterns, and Tools.* Addison-Wesley, 1999.

[Budd, 1980]
Budd, T. A. *Mutation Analysis of Program Test Data.* Ph.D. thesis, Yale University, 1980.

[Budd et al., 1980]
Budd, T. A., DeMillo, R. A., Lipton, R. J., and Sayward, F. G. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In *Proceedings of the 7th ACM Symposium on Principles of Programming Languages*, POPL '80, pages 220–233. Las Vegas, Nevada, 1980.

[Chen et al., 2004]
Chen, T. Y., Huang, D. H., Tse, T. H., and Zhou, Z. Q. Case studies on the selection of useful relations in metamorphic testing. In *Proceedings of the 4th Ibero-American Symposium on Software Engineering and Knowledge Engineering*, JIISIC '04, pages 569–583, 2004.

[Chen et al., 2003]
Chen, T. Y., Tse, T. H., and Zhou, Z. Q. Fault-based testing without the need of oracles. Information and Software Technology, volume 45(1):1–9, 2003.

[Choi et al., 1989]
Choi, B., Mathur, A., and Pattison, B. PMothra: scheduling mutants for execution on a hypercube. Software Engineering Notes, volume 14:58–65, 1989.

[Christopoulos et al., 2000]
Christopoulos, C., Skodras, A., and Ebrahimi, T. The jpeg2000 still image coding system: An overview. IEEE Transactions on Consumer Electronics, volume 46(4):1103–1127, 2000.

[Cobertura, 2010]
Cobertura. The official web site of the Cobertura project, 2010. [accessed July 2010]. URL `http://cobertura.sourceforge.net`

[Davis and Weyuker, 1981]
Davis, M. D. and Weyuker, E. J. Pseudo-oracles for non-testable programs. In *Proceedings of the Association for Computing Machinery '81 conference*, ACM '81, pages 254–257. ACM, 1981.

[DeMillo et al., 1991]
DeMillo, R. A., Krauser, E. W., and Mathur, A. P. Compiler-integrated program mutation. In *Proceedings of the 5th Annual Computer Software and Applications Conference*, COMPSAC '91, pages 351–356, 1991.

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

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

[Dijkstra, 1970]
Dijkstra, E. W. Notes on Structured Programming. Technical Report 70-WSK-03, Technological University Eindhoven, 1970. http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF.

[Do and Rothermel, 2006]
Do, H. and Rothermel, G. On the use of mutation faults in empirical assessments of test case prioritization techniques. IEEE Transactions on Software Engineering, volume 32(9):733–752, 2006.

[Elbaum et al., 2002]
Elbaum, S., Malishevsky, A. G., and Rothermel, G. Test case prioritization: A family of empirical studies. IEEE Transactions on Software Engineering, volume 28(2):159–182, 2002.

[Ernst et al., 2001]
Ernst, M. D., Cockrell, J., Griswold, W. G., and Notkin, D. Dynamically discovering likely program invariants to support program evolution. IEEE Transactions on Software Engineering, volume 27(2):99–123, 2001.

[Fraser and Arcuri, 2011]
Fraser, G. and Arcuri, A. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 416–419. ACM, 2011.

[Fraser and Arcuri, 2012a]
Fraser, G. and Arcuri, A. Sound empirical evidence in software testing. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 178–188. IEEE Press, 2012.

[Fraser and Arcuri, 2012b]
Fraser, G. and Arcuri, A. Whole test suite generation. IEEE Transactions on Software Engineering, volume 99(PrePrints), 2012.

[Fraser and Zeller, 2012]
Fraser, G. and Zeller, A. Mutation-driven generation of unit tests and oracles. IEEE Transactions on Software Engineering, volume 38(2):278–292, 2012.

[Gamma et al., 1995]
Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1995.

[Gosling et al., 2005]
Gosling, J., Joy, B., Steele, G., and Bracha, G. *The Java Language Specification.* Addison-Wesley Professional, 3rd edition, 2005.

[Guderlei and Mayer, 2007]
Guderlei, R. and Mayer, J. Towards automatic testing of imaging software by means of random and metamorphic testing. International Journal of Software Engineering and Knowledge Engineering, volume 17(06):757–781, 2007.

[Hamlet, 1994]
Hamlet, R. Random testing. In *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.

[Harman et al., 2011]
Harman, M., Jia, Y., and Langdon, W. B. Strong higher order mutation-based test data generation. In *Proceedings of the 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC/FSE '11, pages 212–222, 2011.

[Hierons et al., 1999]
HIERONS, R., HARMAN, M., and DANICIC, S. Using program slicing to assist in the detection of equivalent mutants. Software Testing, Verification and Reliability, volume 9:233–262, 1999.

[Howden, 1982]
HOWDEN, W. E. Weak mutation testing and completeness of test sets. IEEE Transactions on Software Engineering, volume 8:371–379, 1982.

[Irvine et al., 2007]
IRVINE, S. A., PAVLINIC, T., TRIGG, L., CLEARY, J. G., INGLIS, S., and UTTING, M. Jumble Java byte code to measure the effectiveness of unit tests. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pages 169–175, 2007.

[Jia and Harman, 2009]
JIA, Y. and HARMAN, M. Higher order mutation testing. Information and Software Technology, volume 51:1379–1393, 2009.

[Jia and Harman, 2011]
JIA, Y. and HARMAN, M. An analysis and survey of the development of mutation testing. IEEE Transactions on Software Engineering, volume 37(5):649–678, 2011.

[JJ2000, 2010]
JJ2000. Official web site of the jj2000 project, 2010. [accessed January 2010].
URL `http://jj2000.epfl.ch`

[Jpeg, 2010]
JPEG. Official web site of the joint photographic experts group, 2010. [accessed January 2010].
URL `http://www.jpeg.org`

[JUnit, 2012]
JUNIT. The official web site of the JUnit project, 2012. [accessed August 2012].
URL `http://www.junit.org`

[Just et al., 2011a]
JUST, R., KAPFHAMMER, G. M., and SCHWEIGGERT, F. Using conditional mutation to increase the efficiency of mutation analysis. In *Proceedings of the 6th ACM/IEEE International Workshop on Automation of Software Test*, AST '11, pages 50–56. ACM Press, 2011.

[Just et al., 2012a]
JUST, R., KAPFHAMMER, G. M., and SCHWEIGGERT, F. Do redundant mutants affect the effectiveness and efficiency of mutation analysis? In *Proceedings of the 7th IEEE International Workshop on Mutation Analysis*, Mutation '12, pages 720–725. IEEE Computer Society, 2012.

[Just et al., 2012b]
JUST, R., KAPFHAMMER, G. M., and SCHWEIGGERT, F. Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation

analysis. In *Proceedings of the 23rd IEEE International Symposium on Software Reliability Engineering*, ISSRE '12. IEEE Computer Society, 2012. To appear.

[Just et al., 2012c]
JUST, R., KAPFHAMMER, G. M., and SCHWEIGGERT, F. Using non-redundant mutation operators to improve the efficiency and effectiveness of mutation analysis, 2012. Working paper (submitted, under review).

[Just and Schweiggert, 2010]
JUST, R. and SCHWEIGGERT, F. Automating software tests with partial oracles in integrated environments. In *Proceedings of the 5th ACM/IEEE International Workshop on Automation of Software Test*, AST '10, pages 91–94. ACM Press, 2010.

[Just and Schweiggert, 2011]
JUST, R. and SCHWEIGGERT, F. Automating unit and integration testing with partial oracles. Software Quality Journal, volume 19:753–769, Springer, 2011.

[Just et al., 2011b]
JUST, R., SCHWEIGGERT, F., and KAPFHAMMER, G. M. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 612–615. IEEE Computer Society, 2011.

[Kaminski et al., 2011]
KAMINSKI, G., AMMANN, P., and OFFUTT, J. Better predicate testing. In *Proceedings of the 6th International Workshop on Automation of Software Test*, AST '11, pages 57–63. ACM Press, 2011.

[King, 1976]
KING, J. C. Symbolic execution and program testing. Communications of the ACM, volume 19(7):385–394, 1976.

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

[Lindholm and Yellin, 2000]
LINDHOLM, T. and YELLIN, F. *The Java Virtual Machine Specification*. Addison-Wesley, 2 edition, 2000.

[Lipton and Sayward, 1978]
LIPTON, R. J. and SAYWARD, F. G. The status of research on program mutation. In *Proceedings of the Workshop on Software Testing and Test Documentation*, pages 355–373, 1978.

[Ma et al., 2002]
MA, Y.-S., KWON, Y.-R., and OFFUTT, J. Inter-class mutation operators for java. In *Proceedings of the 13th International Symposium on Software Reliability Engineering*, ISSRE '02, pages 352–363, 2002.

[Ma et al., 2005]
MA, Y.-S., OFFUTT, J., and KWON, Y. R. MuJava: an automated class mutation system. Software Testing, Verification, and Reliability, volume 15(2):97–133, 2005.

[Ma et al., 2006]
MA, Y.-S., OFFUTT, J., and KWON, Y.-R. MuJava: A mutation system for Java. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 827–830, 2006.

[Mayer, 2005]
MAYER, J. Lattice-based adaptive random testing. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 333–336. ACM, 2005.

[Mayer and Guderlei, 2006]
MAYER, J. and GUDERLEI, R. An empirical study on the selection of good metamorphic relations. In *Proceedings of the 30th Annual International Computer Software and Applications Conference*, COMPSAC '06, pages 475–484. IEEE Computer Society, 2006.

[McCabe, 1976]
MCCABE, T. J. A complexity measure. IEEE Transactions on Software Engineering, volume 2(4):308–320, 1976.

[McMinn, 2004]
MCMINN, P. Search-based software test data generation: a survey. Software Testing, Verification, and Reliability, volume 14(2):105–156, 2004.

[Meyer, 1997]
MEYER, B. *Object-oriented software construction (2nd ed.)*. Prentice-Hall, Inc., 2nd edition, 1997.

[Mills, 1972]
MILLS, H. On the Statistical Validation of Computer Programs. Technical report, Federal Systems Division, IBM, 1972.

[Morell, 1984]
MORELL, L. J. *A Theory of Error-Based Testing*. Ph.D. thesis, University of Maryland at College Park, 1984.

[Morell, 1990]
MORELL, L. J. A theory of fault-based testing. IEEE Transactions on Software Engineering, volume 16(8):844–857, 1990.

[MuJava, 2009]
MUJAVA. The official web site of the MuJava project, 2009. [accessed November 2009]. URL `http://www.cs.gmu.edu/~offutt/mujava`

[Myers, 1979]
MYERS, G. J. *The Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1979.

[Myers et al., 2011]
Myers, G. J., Sandler, C., and Badgett, T. *The Art of Software Testing.* Wiley Publishing, 3rd edition, 2011.

[Namin et al., 2008]
Namin, A. S., Andrews, J. H., and Murdoch, D. J. Sufficient mutation operators for measuring test effectiveness. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 351–360, 2008.

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

[Offutt, 1992]
Offutt, A. J. Investigations of the software testing coupling effect. ACM Transactions on Software Engineering Methodology, volume 1(1):5–20, 1992.

[Offutt and Craft, 1994]
Offutt, A. J. and Craft, W. M. Using compiler optimization techniques to detect equivalent mutants. The Journal of Software Testing, Verification, and Reliability, volume 4:131–154, 1994.

[Offutt et al., 1996]
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, volume 5(2):99–118, 1996.

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

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

[OpenJDK, 2012]
OpenJDK. The official web site of the OpenJDK project, 2012. [accessed November 2011].
URL `http://openjdk.java.net`

[Pacheco et al., 2007]
Pacheco, C., Lahiri, S. K., Ernst, M. D., and Ball, T. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 75–84. IEEE Computer Society, 2007.

[Parr, 2007]
Parr, T. *The Definitive ANTLR Reference: Building Domain-Specific Languages.* Pragmatic Bookshelf, 2007.

[Parr and Fisher, 2011]
PARR, T. and FISHER, K. LL(*): The foundation of the antlr parser generator. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 425–436. ACM, 2011.

[Radio Technical Commission for Aeronautics (RTCA), 2011]
RADIO TECHNICAL COMMISSION FOR AERONAUTICS (RTCA). Software considerations in airborne systems and equipment certification, 2011. RTCA/DO-178C.

[Schuler et al., 2009]
SCHULER, D., DALLMEIER, V., and ZELLER, A. Efficient mutation testing by checking invariant violations. In *Proceedings of the 18th International Symposium on Software Testing and Analysis*, ISSTA '09, pages 69–80. ACM, 2009.

[Schuler and Zeller, 2009]
SCHULER, D. and ZELLER, A. Javalanche: efficient mutation testing for java. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 297–298. ACM, 2009.

[Schuler and Zeller, 2010]
SCHULER, D. and ZELLER, A. (Un-)covering equivalent mutants. In *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation*, ICST '10, pages 45–54, 2010.

[Skodras et al., 2001]
SKODRAS, A., CHRISTOPOULOS, C., and EBRAHIMI, T. The jpeg 2000 still image compression standard. IEEE Signal Processing Magazine, volume 18(5):36–58, 2001.

[Spillner et al., 2007]
SPILLNER, A., LINZ, T., and SCHAEFER, H. *Software Testing Foundations: A Study Guide for the Certified Tester Exam.* Rocky Nook, 2007.

[Tai, 1996]
TAI, K.-C. Theory of fault-based predicate testing for computer programs. IEEE Transactions on Software Engineering, volume 22(8), 1996.

[Tillmann and De Halleux, 2008]
TILLMANN, N. and DE HALLEUX, J. Pex: White box test generation for .net. In *Proceedings of the 2nd International Conference on Tests and Proofs*, TAP '08, pages 134–153. Springer-Verlag, 2008.

[Untch et al., 1993]
UNTCH, R. H., OFFUTT, A. J., and HARROLD, M. J. Mutation analysis using mutant schemata. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '93, pages 139–148, 1993.

[Visser et al., 2004]
VISSER, W., PĂSĂREANU, C. S., and KHURSHID, S. Test input generation with Java PathFinder. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04, pages 97–107, 2004.

[Voas, 1992]
Voas, J. M. PIE: a dynamic failure-based technique. IEEE Transactions on Software Engineering, volume 18:717–727, 1992.

[Weyuker, 1982]
Weyuker, E. J. On Testing Non-Testable Programs. The Computer Journal, volume 25(4):465–470, 1982.

[Zhou et al., 2004]
Zhou, Z. Q., Huang, D. H., Tse, T. H., Yang, Z., Huang, H., and Chen, T. Y. Metamorphic testing and its applications. In *Proceedings of the 8th International Symposium on Future Software Technology*, ISFST '04, 2004.

[Zhu et al., 1997]
Zhu, H., Hall, P. A. V., and May, J. H. R. Software unit test coverage and adequacy. ACM Computing Surveys, volume 29(4):366–427, 1997.