Comparing Mutation Coverage Against Branch Coverage in an Industrial Setting

Ali Parsai and Serge Demeyer

{ali.parsai, serge.demeyer}@uantwerpen.be
Universiteit Antwerpen and Flanders Make

Abstract

The state-of-the-practice in software development is driven by constant change fueled by continuous integration servers. Such constant change demands for frequent and fully automated tests capable to detect faults immediately upon project build. As the fault detection capability of the test suite becomes so important, modern software development teams continuously monitor the quality of the test suite as well. However, it appears that the state-of-the-practice is reluctant to adopt strong coverage metrics (namely mutation coverage), instead relying on weaker kinds of coverage (namely branch coverage). In this paper, we investigate three reasons that prohibit the adoption of mutation coverage in a continuous integration setting: (1) the difficulty of its integration into the build system, (2) the perception that branch coverage is "good enough", and (3) the performance overhead during the build. Our investigation is based on a case study involving four open source systems and one industrial system. We demonstrate that mutation coverage reveals additional weaknesses in the test suite compared to branch coverage, and that it is able to do so with an acceptable performance overhead during project build.

1 Introduction

With the popularity of agile methods, especially test-driven development [14] and continuous integration [29], developers shifted their focus to test the changes to their software early and often [65]. This strategy demands a high standard of quality for test suite to avoid potential faults as much as possible in the later stages of software development [15]. Developers assess the quality of test suite according to its capability of detecting yet unknown faults. For a faulty statement to be revealed, the program must pass through four stages: (i) the faulty statement must be executed *(Reachability)*, (ii) the faulty statement needs to affect the program state *(Infection)*, (iii) the effect of the faulty statement on program state need to propagate to the program output *(Propagation)*, and the test needs to observe the failure in the program output *(Reveal)* [8, 56]. Several coverage metrics exist that aim to quantify the quality of a test suite. Among them, mutation coverage is generally acknowledged as the state-of-the-art coverage metric since it checks whether a test covers all four aforementioned stages [9, 10, 38, 48, 74]. Indeed, mutation testing is the process of deliberately injecting faults into a software system, and then verifying whether the tests actually fail. This faulty version of the software is called a *mutant*. The faults injected by each mutant are modeled after the common mistakes often made by developers, hence mutation testing provides a repeatable and scientific approach to measure the fault detection capability of a test suite [47, 74]. Using the results of mutation testing, it is possible to improve test suite quality or prioritize tests [12, 27, 86, 87]. Comparative stud-

ies demonstrated that in terms of fault detection, mutation testing is more effective than several other coverage criteria [9, 30, 57]. If the fault model used for mutation operators is close to reality, mutation testing produces more accurate results than simple coverage metrics [10]. Finally, mutation testing has been shown to subsume statement and branch coverage [71]. This is due to the fact that branch and statement coverage only check for reachability [38].

Despite these promising results, the state-of-the-art is not yet adopted into the state-of-the-practice. For instance, Gopinath et al. put forward that mutation testing *"is generally not used by real-world developers with any frequency"* [36]. Instead, the state-of-the-practice relies on simple coverage metrics such as statement and branch coverage. This is despite their inadequacy for assessing the fault detection capability of a test suite [91, 41]. Among these metrics, branch coverage is commonly used in industry [88]. However, even with 100% of branch coverage there is still the potential for faults to go unnoticed, because branch coverage only checks whether the test code executes each branch and does not assess whether the program state is infected or the fault is propagated to the output [62, 36, 90]. Because of this, such coverage metrics cannot reveal anything regarding the quality of the test oracle.

A possible explanation for the preference towards simple coverage metrics is they are relatively easy and fast to collect. Today, there are plenty of test coverage tools available that instrument the code, execute the tests, and report the parts of the code not covered by the tests (i.e., statements, branches, paths). Also, while the performance overhead of these tools on the overall test execution are not negligible, it can often be tolerated in development environments [16, 22]. Moreover, Gligoric et al. demonstrate that branch coverage—among several coverage criteria—is the best one to predict the mutation coverage of a test suite [35]. From a practitioner's point of view, branch coverage is a reasonable quality criterion because of the trade-off between time (the performance overhead induced by collecting the measurements) and quality (a "good enough" fault detection capability). This means that the successful adoption of mutation testing in industry re-

quires convincing practitioners that there are tangible and worthwhile differences between the two methods when it comes to analyzing industrial software.

Literature blames the lack of adoption of mutation testing in industry mainly on the performance overhead, leading to the adagium —*do fewer, do smarter, and do faster* [70]. Examples in that sense are mutant sampling, weak mutation testing, and performing the mutation testing on byte code rather than the source code [47, 74]. However, the performance overhead is only one part of the equation. We argue that there are two other issues when adopting mutation testing in an industrial strength continuous integration setting: (i) the complexity of continuous build environments and (ii) the optimism regarding the added value of mutation testing. We expand on each of those issues below.

(i) First, the research on mutation testing has ignored the novel trend that modern test infrastructure is part of a continuous integration environment. In such a setting, the build steps follow one another in lockstep. In fact, fully automatic test infrastructure is now common place in many companies, e.g. Google performs roughly 800 thousand builds and 150 million test runs in an average day automatically [66]. Integrating additional steps (such as calculating mutation coverage) easily interferes with such automated systems, and requires up-front considerations in the design to ease the integration.

(ii) Second, there is little empirical evidence that mutation testing reveals additional weaknesses in industrial strength test suites. Typically, studies that promote mutation testing are based on open source cases with components designed to be open and accessible [31, 44, 35, 57]. The same is not true for the industrial systems, where brown field development is common and legacy code is integrated as black-box components and tested accordingly [42, 33].

We derived these issues from a pilot study, integrating mutation testing in an industrial strength continuous integration setting. In particular, we were asked to explore the advantages and drawbacks of mutation testing in the context of the Segmentation component of the Impax ES medical imaging soft-

2

ware used by Agfa HealthCare. The Segmentation component provides imaging algorithms to perform segmentation on 3D modeled volumes. As can be expected from a software system in healthcare, it must adhere to strict safety standards including advanced monitoring of test quality. Yet, the segmentation component interfaced with some legacy code, hence black-box testing was an inherent part of the testing strategy. Also, the Impax ES system is implemented as a small product-line, hence the Maven build system was configured to resolve dependencies with libraries depending on the product line variant to be built.

In our work, we first explore the feasibility of mutation testing in an industrial project which relies on a continuous integration system (RQ1). Then, we investigate the pros and cons that arise by its adoption. We attempt to verify that compared to branch coverage, mutation testing has the pros of revealing additional weaknesses in the test suite (RQ2), and it has the cons of introducing overhead (RQ3). This leads us to pursue the following research questions:

RQ1: *Is it feasible to integrate mutation testing in a continuous integration system?*

$\Rightarrow$ To answer this question we follow a proof by construction. We first integrate an existing mutation coverage tool (namely PITest) into the build system (namely Maven) We report the challenges we encountered and the workarounds we performed, all to no avail. Consequently, we adapted and used a mutation testing tool named *LittleDarwin* specifically designed to integrate well within a continuous integration environment.

RQ2: *Does mutation testing reveal additional weaknesses in the test suite compared to branch coverage?*

$\Rightarrow$ To answer this question we analyze branch coverage along with the mutation coverage. We investigate parts of the code where these two metrics differ to see whether mutation coverage indeed exposes additional weaknesses. To increase the generalizability of our findings, we perform the same comparison on four open source systems.

RQ3: *Can we reduce the performance overhead induced by mutation testing to an acceptable level?*

$\Rightarrow$ To answer this question, we measure the performance overhead induced by a full mutation analysis, injecting 12K mutants for 38K lines of code.

We compare this against the performance overhead after *mutant sampling* (Section 2.3), effectively reducing the number of mutants to 34.7%. We verify the results of the full mutation coverage against the sampled mutation coverage to see whether the reduced number of mutants still reveals weaknesses in the test suite.

The rest of paper is structured as follows. In Section 2, we provide some background information on test coverage in general and mutation testing in particular. We describe the main tools used in this study in Section 3. We then proceed with a discussion of the case study design, including a description of the cases under investigation as well as a detailed explanation of the setup for the open source and industrial cases in Section 4. The results of our case study are then discussed in Section 5, followed by a discussion of the threats to the validity of this study in Section 6. An overview of related work is presented in Section 7. Finally, in Section 8, we present our final conclusions.

2 Background

In this section we present an overview of the background information necessary to understand the rest of the paper. We briefly introduce continuous integration environments and discuss the importance of test suite quality therein (Section 2.1), give a brief explanation of code coverage in general and branch coverage in particular (Section 2.2), and discuss mutation testing and its related concepts in detail (Section 2.3).

2.1 Testing in Continuous Integration Environment

Continuous integration is defined as the practice of merging the developed code with a central source code repository as often as possible. The concept of continuous integration was first proposed by Booch as a way to avoid integration problems [17]. This method has been in the center of attention in the past decade since it is the basis for today's agile development techniques. Continuous integration allows

the software to be continuously tested. Simple tasks (e.g. unit tests) can be triggered upon each commit; whereas, the time-consuming tasks (e.g. integration test) can be postponed to the nightly build. Providing continuous feedback, the continuous integration environment takes care that the code-base remains stable during development, and reduces the risk of arriving in integration hell (the point in production when members on a delivery team integrate their individual code) [29].

The introduction of agile development techniques has resulted in an increased interest in the fault detection capability of the test suite. This is typically monitored by means of *code coverage metrics*. In this context, a weakness in a test suite is a lack of capacity of a test suite to detect faults in a particular part of software, either by a lack of tests to cover that part, or incomplete testing of the covered part.

2.2 Code Coverage Metrics

Code coverage is defined as the proportion of code that is tested by the test suite. There are several ways to calculate code coverage. The most often used metrics in industry are statement coverage and branch coverage [36]. Statement coverage is the number of statements in the program that are executed at least once by the test suite divided by the total number of statements. Similarly, branch coverage is the number of branches executed at least once by the test suite divided by the total number of branches (Equation 1). Branch coverage subsumes statement coverage, because if all branches are examined, all statements contained in the branches are examined in the process [98, 8]. Branch coverage is often used in popular industrial tools to evaluate the quality of a test suite. More specifically, a high value of branch coverage is assumed to imply a "good enough" test suite [94].

$$Branch\ Coverage = \frac{Number\ of\ branches\ executed\ at\ least\ once}{Number\ of\ all\ branches} \quad (1)$$

A test suite that achieves 100% coverage according to a certain coverage criterion is called an *adequate test suite*. For example, a test suite is branch-adequate when its branch coverage is 100%. However, it is known that statement-adequate or branch-adequate test suites are ineffective for assessing the fault detection capability of a test suite [43, 91, 44, 41]. An alternative test coverage metric that is used mostly in safety-critical context is decision coverage, namely the proportion of decision points triggered by tests. The avionics standard RTCA/DO-178C and the automotive standard ISO26262 enforce complete modified condition/decision coverage (MC/DC) [20, 45]. Writing tests that achieve complete MC/DC is very difficult, and needs a high level of expertise in the system under test [40]. Yet, even 100% MC/DC does not guarantee the absence of faults [34, 50].

2.3 Mutation Testing

Mutation testing is the process of injecting faults into a software system and then verifying whether the test suite indeed fails (i.e. detects the injected fault). The idea of mutation testing was first mentioned in a class paper by Lipton (as reported by Offutt et al. in [70]), and later developed by DeMillo, Lipton and Sayward [25]. The first implementation of a mutation testing tool was done by Timothy Budd in 1980 [21].

Mutation testing induces the following steps on the test process. It starts with a *green* test suite — a test suite in which all the tests pass. First, a faulty version of the software is created by introducing faults into the system *(Mutation)*. This is done by applying a known transformation *(Mutation Operator)* on a certain part of the code. After generating the faulty version of the software *(Mutant)*, it is passed on to the test suite. If there is an error or failure during the execution of the test suite the mutant is marked as killed *(Killed Mutant)*. If all tests pass, it means that the test suite could not catch the fault and the mutant has survived *(Survived Mutant)*.

Invalid Mutants.

In the process of generation of mutants, sometimes a mutant is not compilable. Such mutants are called *invalid mutants*. Given the fact that typical mutation testing tools do not attempt to compile the code

4

entirely, it is possible that mutants are created that adhere to the syntax of a language, but cannot be compiled. For example, in case of concatenation of two string variables using "+" operator, changing this operator to "-" leads to generation of an invalid mutant. While most invalid mutants can be avoided at mutant generation time, some are difficult to filter out without having the facilities of a compiler.

Equivalent Mutants.

If the output of a mutant for all possible inputs is the same as the original program, it is called an *equivalent mutant*. It is not possible to create a test case that passes for the original program and fails for an equivalent mutant, because the equivalent mutant has the same semantics as the original program. This makes the creation of equivalent mutants undesirable, since the time that the developer wastes on an equivalent mutant does not result in the improvement of the test suite. Equivalent mutants have a significant impact on the accuracy of the mutation coverage [37]. Unfortunately, equivalent mutants are not easy to detect because they depend on the context of the program itself [61]. For example, in Figure 1, $--$ replacing $++$ in *proc1* changes the output for any input other than 0, while the same mutant in `proc2` does not. Indeed, the preceding line $i++$ ensures that the condition $i > 0$ is always met for $i \geq 1$. The mutant can be killed in `proc1`, because $i = 0$, however in `proc2` the mutant is undetectable by any test because of $i = 2$.

As for filtering the equivalent mutants, there are no tools available that automatically detect and remove all equivalent mutants. In general, detection of equivalent mutants is an undecidable problem [68]. Manual inspection of all mutants is the only way of filtering all equivalent mutants, which is impractical due to the amount of work it needs. Therefore, the common practice within today's state-of-the-art is to take precautions to remove as many equivalent mutants as possible (e.g. using Trivial Compiler Equivalence [73]), and accept equivalent mutants as a threat to validity.
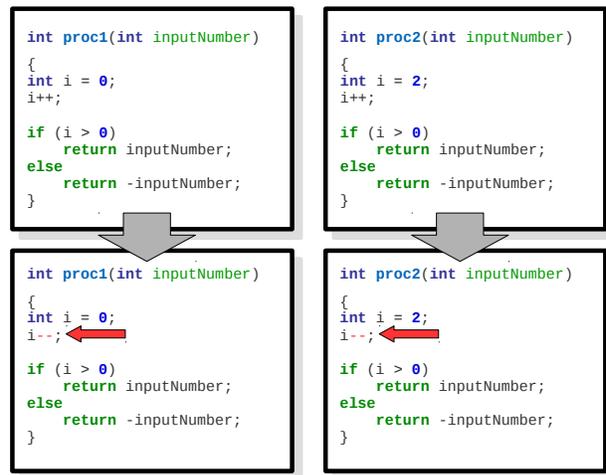

Figure 1: Example of an equivalent mutant in `proc2`

Mutation Coverage.

Mutation testing allows software engineers to monitor the fault detection capability of a test suite by means of *Mutation Coverage* (see Equation 2). A test suite is said to achieve *full mutation test adequacy* whenever it can kill all of the non-equivalent mutants, thus reaches a mutation coverage of 100%. Such test suites are called *mutation-adequate test suites*.

$$Mutation\ Coverage = \frac{Number\ of\ killed\ mutants}{Number\ of\ all\ non\text{-}equivalent\ mutants} \quad (2)$$

Mutation coverage is often declared as a *stopping criterion* for writing (unit) tests — the next level of testing can only start when mutation coverage exceeds a given threshold [69, 98]. This is especially useful when tests are generated automatically [24, 32].

Mutation Operators.

A mutation operator is a known transformation which creates a faulty version by introducing a single change. The first set of the mutation operators designed were reported in King et al. [51]. These operators which work on very basic entities were introduced in the tool Mothra which was designed to mutate FORTRAN77 programming language. In 1996,

5

Offutt et al. determined that a selection of few mutation operators are enough to produce similarly capable test suites with a four-fold reduction of the number of mutants [67]. This reduced set of operators shown in Table 1 remained more or less intact in all subsequent research papers.

Table 1: Reduced-set mutation operators (adapted from [60] ©ACM 2006)

Operator	Description
AOR	Arithmetic Operator Replacement
AOD	Arithmetic Operator Deletion
AOI	Arithmetic Operator Insertion
ROR	Relational Operator Replacement
COR	Conditional Operator Replacement
COD	Conditional Operator Deletion
COI	Conditional Operator Insertion
SOR	Shift Operator Replacement
LOR	Logical Operator Replacement
LOD	Logical Operator Deletion
LOI	Logical Operator Insertion
ASR	Assignment Operator Replacement

With the popularity of the object-oriented programming paradigm, there was a need to design new mutation operators to simulate the faults that occur in this kind of programs. Several studies proposed new mutation operators [59, 23], and some of them were designed to prove the usefulness of object-oriented operators [54, 26]. Ahmed et al. did a complete survey on this subject [6].

During the past decade, the academic focus was on creating new mutation operators for special purposes such as targeting certain security problems [84, 95] or language specific mutation operators [3, 18, 85, 76, 75]. These mutation operators, even though important in their own context, do not relegate into the general concept of mutation testing. The traditional mutation operators are by far the most often implemented [74]. One reason for this is that using more mutation operators produces more mutants; which makes the procedure longer to finish, and as a result, less practical. The reduced set of operators mentioned in Table 1 provides a smaller set which produces results with enough detail for any practical purpose, even though the confidence in such results are slightly less than those retrieved by using additional mutation operators.

Mutant Sampling.

To make mutation testing practically applicable, it is important to reduce the time needed — do fewer, do smarter, and do faster [70]. "Do fewer" is achieved by *mutant sampling*: randomly selecting a sample set of mutants instead of processing all of them. This idea was first proposed by Acree [4] and Budd [21] in their PhD theses. Since then, there were many studies confirming the effectiveness of this approach: the performance gain is significant yet reveals the same weaknesses [51, 92, 64, 97]. The random mutant selection can be performed uniformly, meaning that each mutant has the same chance of being selected. Otherwise, the random mutant selection can be enhanced by using heuristics based on the source code.

The percentage of mutants that are selected determines the *sampling rate* for random mutant selection. Using a fixed sampling rate is common in literature [93, 97, 96]. However, it is possible to use a weight factor to optimize the sampling rate according to various parameters such as the number of mutants per class. This is called *weighted* mutant sampling [77]. It is also possible to determine the sampling rate dynamically while performing mutation testing. A method resembling the latter was proposed by Sahinoglu and Spafford to randomly select the mutants until the sample size becomes statistically appropriate [82]. They concluded that their model achieves better results due to its self-adjusting nature [47].

There is one other factor besides the sampling rate that needs to be considered when sampling; the total amount of time that is practically viable. Unfortunately, in the current literature we did not find any concrete targets. Therefore, we set our own target based on a hypothetical scenario of an agile team running the whole mutation testing once every week during the weekend. In this scenario, the team works from Monday at 8am till Friday at 6pm, which leaves the whole weekend (thus 62 hours) to perform the analysis.

6

3 Tools Used in This Study

In this section we present the test coverage tools used to investigate the trade-offs between branch coverage and mutation coverage in an industrial setting. These tools are JaCoCo, a tool to compute statement and branch coverage for Java software systems (Section 3.1); PITest a state-of-the-art mutation testing tool designed for easy integration with current test and build tools (Section 3.2) and LittleDarwin, the tool we adapted and used to perform mutation testing on Java software systems with complicated build systems (Section 3.3).

3.1 JaCoCo.

JaCoCo [http://www.eclemma.org/jacoco] is a lightweight, flexible, and well documented tool to provide statement and branch coverage for Java programs. JaCoCo is compatible with most Java build systems, hence is easy to deploy in a continuous integration environment. JaCoCo is the de facto standard for measuring test coverage for Java projects, and is used as baseline in research concerning test coverage (e.g. [55, 46, 13, 52]).

JaCoCo uses a set of different probes to calculate coverage metrics. All these probes are instrumented into Java class files which are Java byte code instructions and debug information optionally embedded therein. Consequently, JaCoCo uses dynamic analysis to compute coverage over byte code which allows it to work even without the source code available. The link to source files are then generated using the debug information that accompanies Java byte code.

However, byte code instrumentation has known disadvantages [55, 89]: By performing a study on JaCoCo, Tengeri et al. identify 6 reasons why using byte code instrumentation in Java language might not be as accurate as source code instrumentation [89]. In particular, (i) the act of instrumentation itself can affect the behavior of the tests, (ii) cross-coverage among submodules is not taken into account, (iii) untested submodules are excluded from the analysis, (iv) method signatures are different in byte code and source code for methods that contain compiler-injected parameters, (v) exceptions leading to inter-

ruption of the control flow result in loss of information, and (vi) generated code produces obstacles in collecting coverage information. Consequently, the results from JaCoCo—especially when statements or branches are reported as *not* being covered by a test—need to be double-checked for accuracy.

JaCoCo is used throughout this study to compute branch coverage for industrial and open source cases.

3.2 PITest

PITest [http://pitest.org/] is a state-of-the-art mutation testing system for Java, designed to be fast and scalable. PITest seamlessly integrates with today's test and build tools (i.e. Ant, Gradle and Maven). PITest is the de facto standard for mutation testing within Java, and it is used as baseline in research concerning mutation testing (e.g. [49, 44, 79, 83, 11]).

PITest has a wide range of mutation operators. The default setting is practically a subset of the reduced-set of mutation operators (Table 2), however because they are applied to byte code, they are grouped and named differently. There are several other mutation operators that can be enabled as well.

PITest comes with a lot of internal optimizations to tackle the "do faster" part of the maxim: *do fewer, do smarter, and do faster* [70]. Most importantly, mutations are performed at the level of byte code to avoid recompilation. However, byte code level mutation presents other obstacles to overcome. For example, PITest needs to find and execute tests by itself for each mutant, which causes incompatibility with complicated build structures. This is more apparent when test code is located in separate packages, and therefore, not easily discoverable by PITest. In addition, PITest incorporates some heuristics to choose which tests to run, which implies that the accuracy of these heuristics affects the results of PITest as well. In particular, PITest uses the same mechanism as JaCoCo to determine statement coverage, and skip the evaluation of mutants in uncovered statements. This, in turn, raises similar issues as described in Section 3.1.

PITest is used in this study in an attempt to demonstrate the feasibility of mutation testing in an industrial environment (RQ1).

7

Table 2: PITest mutation operators at the time of this study

Operator	Description	Example	
		Before	**After**
CBM	Mutates the boundry conditions	$a > b$	$a >= b$
IM	Mutates increment operators	$a + +$	$a - -$
INM	Inverts negation operator	$-a$	a
MM	Mutates arithmetic & logical operators	$a \& b$	$a \,|\, b$
NCM	Negates a conditional operator	$a == b$	$a! = b$
RVM	Mutates the return value of a function	return true;	return false;
VMCM	Removes a void method call	voidCall(x);	—

3.3 LittleDarwin.

LittleDarwin [http://littledarwin.parsai.net/] is a mutation testing tool created by Ali Parsai (first author of this paper) to provide mutation testing within a continuous integration environment. It is designed to have a loose coupling with the test infrastructure, instead relying on the build system to run the test suite. LittleDarwin imposes two restrictions only: (a) the build system must be able to run the test suite; (b) the build system must return non-zero if any tests fail, and zero if it succeeds. For a detailed description of LittleDarwin, please refer to Parsai et al. [78].

For the purposes of this study, there are 9 mutation operators implemented in LittleDarwin listed Table 3. These operators are a subset of the reduced-set of mutation operators (Table 1), and similar to the default setting of PITest, but differently grouped and named. Since the number of mutation operators of LittleDarwin is limited, it is possible that no mutants are generated for a class. In practice, we observed that usually only very small compilation units (e.g. interfaces, and abstract classes) are subject to this condition.

Table 3: LittleDarwin mutation operators

Operator	Description	Example	
		Before	**After**
AOR-B	Replaces a binary arithmetic operator	$a + b$	$a - b$
AOR-S	Replaces a shortcut arithmetic operator	$+ + a$	$- - a$
AOR-U	Replaces a unary arithmetic operator	$-a$	$+a$
LOR	Replaces a logical operator	$a \& b$	$a \,|\, b$
SOR	Replaces a shift operator	$a >> b$	$a << b$
ROR	Replaces a relational operator	$a >= b$	$a < b$
COR	Replaces a binary conditional operator	$a \&\& b$	$a \,||\, b$
COD	Removes a unary conditional operator	$! a$	a
SAOR	Replaces a shortcut assignment operator	$a * = b$	$a \,/ = b$

For the moment, LittleDarwin is not optimized for speed. For each mutant injected, LittleDarwin demands a complete rebuild and test cycle on the build system. This easily leads to several hours of analysis time. Currently the only way to speed-up LittleDarwin is to use mutant sampling, which is covered under RQ3.

LittleDarwin is used throughout this study to compute mutation coverage for industrial and open source cases.

4 Case Study Design

In this section, we explain the details about the setup of our case study. We start describing the industrial case (Section 4.1) and open source cases (Section 4.2). Then, we report the setup of the tools (in Section 4.3). Finally, we provide an overview of the comparison criteria (in Section 4.4).

4.1 Industrial Case

There are three main components which create the core of Impax ES Clinical Applications. One of these three components is the Segmentation component. The main use of this component is to provide imaging algorithms to segment 3D volumes. This component is average in size compared to the other components of the system, and it includes a test suite which was under active development at the time of the case study. The component is entirely written in Java.

The team is geographically dispersed across four cities around the globe, thus making coordination a critical part of the software development. This also increases the importance of the test suite during build, because faults discovered downstream require long distance communication over different time zones. The team uses the SCRUM model of development, by holding a weekly sprint meeting to coordinate their efforts [80]. They use a typical continuous integration with separate build servers and source code repository servers, centered around Maven. The build configuration includes several plugins to compute code coverage, generate reports, and obfuscate the target classes.
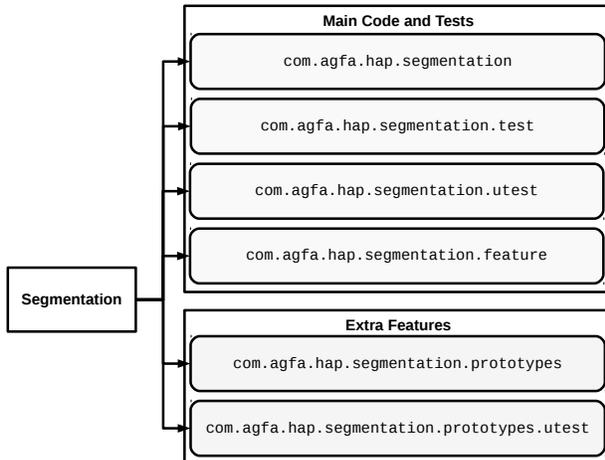
8

Figure 2: Agfa HealthCare Segmentation build components

The Impax ES system is released in two main variants (production or prototype). There are a few minor variants as well, depending on the target hardware platform the system is deployed upon. Therefore, the Maven build system was configured to resolve dependencies with libraries depending on the product line variant to be built. The system architecture itself relies on the OSGI (Open Service Gateway Initiative) dynamic component model, to load and unload components dynamically without rebooting the system. The extensive use of (dynamic) OSGI headers implies a complicated build process where the Maven plug-in Tycho [https://eclipse.org/tycho/] is used to fetch dependencies, compile source files, and run the test suite.

The Segmentation component itself is divided into 6 subcomponents as shown in Figure 2. The main code is located in *com.agfa.hap.segmentation* while the test suite code, some resources used by unit tests and a feature manifest for eclipse are located in the *com.-agfa.hap.segmentation.{utest, test, feature}* subcomponents respectively. The other two subcomponents provide early prototype features and their unit tests which should be excluded from the final build. Each subcomponent has a separate *pom.xml* file and can be built on its own. There is also a parent *pom.xml* file

which selects and builds these subcomponents based on the profile of the product-line variant to be built (production or prototype).

For our case study, we focus on unit test suite of Segmentation component, and we do not include the acceptance tests. Removing the acceptance tests decreases the total time for compilation and testing of the Segmentation component from a few hours to less than a minute in each build.

4.2 Cases Under Investigation

To increase the generalizability of our findings we analyzed four open source systems for addressing RQ2. The descriptive statistics of all cases under investigation (the industrial one + the four open source ones) are reported in Table 4. Because the non-disclosure agreement does not allow us to reveal too many details, we only list approximate statistics for the industrial case.

4.3 Tool Setup

JaCoCo was already in use as part of the Maven build configuration in the industrial case and some of the open source cases, therefore we used JaCoCo to calculate the branch coverage for the rest as well. For this case, JaCoCo was being used with its default parameters. PITest was run using the default suite of mutation operators, and it was run in parallel mode, which detects the number of available CPU cores and uses all of them for the analysis. Finally, LittleDarwin was run with two sets of commands: the first to compile the mutated source code and install the final result into the local Maven repository; the second to execute the test suite on the compiled version. This was necessary because the production code and the test code have separate build systems.

To make the performance comparison (RQ3) valid, we ran the analysis of all open source cases on the same machine, operating system, and python interpreter. However, since the industrial case was analyzed 5 months prior to the open source cases on premise, the analysis was done on different hardware and operating system, hence the absolute numbers of the execution times cannot be compared. For the

Table 4: Descriptive statistics for cases under investigation

| Case | URL | Version | Size (LoC) | | Ratio |
			Main	Test	
Industrial Case (The numbers are approximated for confidentiality reasons.)					
Agfa Segmentation	http://www.agfahealthcare.com/ global/en/main/resources/product_ images/impax_6_0.jsp	3.7-snapshot	38K	50K	~1.3
Open Source Cases					
Joda Time	http://www.joda.org/joda-time/	2.8	28479	54645	1.92
Apache Commons Codec	http://commons.apache.org/proper/ commons-codec/	1.7	5773	9917	1.72
jOpt Simple	http://pholser.github.io/ jopt-simple/	5.0	1958	6072	3.10
AddThis Codec	http://github.com/addthis/codec	3.2.1	3614	1318	0.36

industrial case, the machine used has two Intel Xeon 2.80 GHz processors with 16 GB (4x4 GB) of DDR3-1333 memory running Windows 7 Enterprise Edition. Since LittleDarwin only uses a single thread to perform its analysis, there are no gains from the multi-core architecture of the hardware it runs on. The open source cases were analyzed on a custom made PC with AMD 1090T 3.2 GHz processor and 8 GB (2x4 GB) memory running Linux Mint 17. The execution time for a build is extracted from the output of the build system (which was Maven in all our cases). The execution time for JaCoCo was extracted in the same manner, since JaCoCo acts as a Maven plugin. The execution time for LittleDarwin was calculated by aggregating the time spent generating the mutants and the time spent for gathering the results of the execution of the tests for all mutants.

4.4 Comparison Criteria

In this paper we analyze branch and mutation coverage from a conceptual point of view. For both metrics, a higher value is assumed to suggest a good test suite quality to the developer. While branch coverage is widely used in industry, mutation coverage is known to be a better indication of fault detection capability of a test suite. For this reason, we are interested to explore the situation where branch and mutation coverage present different values.

Considering m as mutation coverage percentage[1], b

---

[1]Mutation coverage is often expressed as a ratio rather

as the branch coverage percentage, and t as a threshold, we define 5 categories (Table 5). Note that we consider coverage at class level, hence the unit of analysis is a *class*.

- The Category SimCov corresponds to the category in which the difference between m and b is less than a given threshold ($t\%$). For these classes mutation coverage does not provide extra information with respect to branch coverage.
- For the category LoB-HiM the mutation coverage is larger than the branch coverage. There mutation testing provides extra confidence concerning the test suite; despite the low branch coverage the test suite has a high fault detection capability. *This category represents those classes where branch coverage is "good enough".*
- In contrast, the category HiB-LoM marks classes where the mutation coverage is smaller than the branch coverage. This is the most interesting category for our investigation. Indeed, mutation testing reveals weaknesses in the test suite, namely where test suite lacks of detectability of a potential fault (Section 2.1). From another point view, *this category shows where high branch coverage gives a sense of false confidence regarding the fault detection capability of the test suite.*
- Finally, the categories NoB and NoM mark the special cases where the corresponding coverage metric is zero. If both the branch and mutation

than a percentage, but for the sake of consistency, we use it as a percentage here.

10

Table 5: Categorization of differences between branch and mutation coverage

Category	Definition	Description
SimCov	$(\lvert m - b \rvert <= t \wedge m, b > 0) \vee m = b = 0$	Similar branch and mutation Coverage
LoB-HiM	$m - b > t \wedge m, b > 0$	Low Branch coverage and High Mutation coverage (confirms the fault detection capability of the test suite)
HiB-LoM	$b - m > t \wedge m, b > 0$	High Branch coverage and Low Mutation coverage (false confidence regarding the fault detection capability of the test suite)
NoB	$m > 0 \wedge b = 0$	No Branch coverage
NoM	$b > 0 \wedge m = 0$	No Mutation coverage

coverage are zero, it most likely corresponds to a class which is never tested. However, for NoB, this may also be due to anomalies in the byte code level instrumentation of JaCoCo (see Section 3.1). When NoM is zero this is most likely caused by lack of mutable statements in the code (see Section 3.3).

The value of t determines the threshold that the two coverage scores are considered close enough so that the difference between them does not make any practical difference. For example, for a threshold of 10%, if for a particular class the branch coverage is 55% and mutation coverage is 63%, having either of these scores does not change the perceived test quality of that class, and the coverage can be interpreted as "around 60%". Therefore, by adjusting t, we can model the sensitivity of developers to the coverage score. This threshold does not have any effect on the number of classes in Categories NoB and NoM, and only changes the number of classes in Categories SimCov, LoB-HiM and HiB-LoM. To determine t we examine the results of both metrics, and choose the minimum value of t where neighboring t values would not change the number of classes in each category. This way, the threshold is selected based on the specifics of each case, minimizing the analysis bias. For our investigation, we tried all ordinal values of t between 1 and 25 and ultimately derived $t = 11$ and $t = 7$ respectively for the industrial and the open source cases. The result of this derivation for the industrial case is shown in Figure 3. The number of classes in different categories remain fairly constant after $t = 11$, hence is chosen as the threshold.

Another way to compare mutation and branch cov-erage is to compute the correlation of these two metrics. Specifically, we analyze whether the order of a set of classes by one metric can predict the order by another set using Kendall's τ_b coefficient as described in statistic handbooks [2, 5]. This has been used previously in a similar study by Gligoric et al. where they argue that this coefficient is more appropriate than a linear correlation coefficient because it does not assume a linear correlation between the metrics [35].

5 Results and Discussion

In this section, we discuss the results of our study. For each research question, we briefly describe our motivation, then our approach, and conclude with our observations.

RQ1. Is it feasible to integrate mutation testing in a continuous integration system?

Motivation. Mutation testing adds the following steps to the build process: (a) inject a mutant into the software system, (b) build the mutated system, (c) execute the test suite, (d) record the results, (e) restore the system to its original state, and (f) repeat the procedure until there is no more mutants. These steps easily interfere with other parts of the build process (i.e. selection of product line variants, dependency resolution, and dynamic component loading) hence the reason to address the feasibility issue.
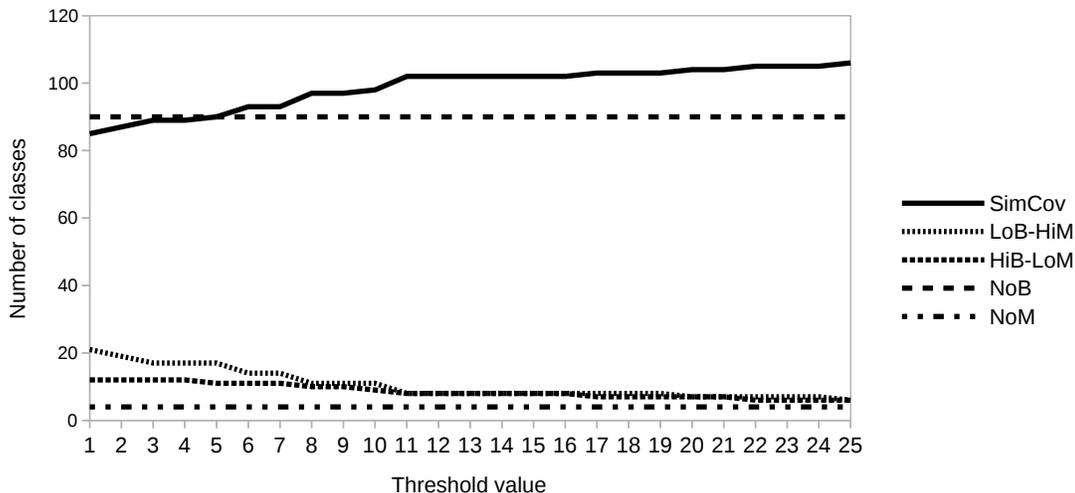
Figure 3: Number of classes in each category for t values between 1 and 25 in the industrial case

Approach. To answer this question we follow a proof by construction. We integrate a state-of-the-art mutation coverage tool (namely PITest) into the build environment of the industrial case (namely Maven). The industrial case is representative for many other industrial systems: it has legacy components (where unit tests are missing), it has two major variants (incorporated in a product-line architecture), and has a complicated build structure (where components are dynamically loaded into the build environment by means of OSGI). We report the challenges we encountered and the workarounds we performed, all to no avail. Consequently, we adapted and used a mutation testing tool named LittleDarwin specifically designed to integrate well within a continuous integration environment.

Findings. We considered a series of tools during our feasibility study: PITest, Cheshire, and MuUnit. PITest is a widely-used mutation testing tool aimed at industrial projects; Cheshire is a tool to convert OSGI interfaces and MuUnit is a mutation testing tool designed to perform its analysis on OSGI projects. The challenges we faced performing the proof by construction are summarized as follows:

- **OSGI.** As explained in Section 4.1, the industrial case heavily relies on the OSGI (Open Service Gateway Initiative) dynamic component model for dynamic loading and unloading of components. We first considered to refactor the system and remove the OSGI headers. However, we learned that these OSGI headers are deeply embedded in all of the source code. Removing the OSGI headers would alter the code beyond recognition hence was not an option.

- **PITest.** PITest does not refer to OSGI in its documentation, nor did we find any other information sources. We tried it out ourselves, and quickly discovered that PITest could not run the OSGI-dependent code by itself, and the Tycho plug-in for Maven was incompatible as well. We posted a few questions in the PITest forums and there it was confirmed that OSGI could cause problems (see https://groups.google.com/d/topic/pitusers/IH21Q4jJaco/discussion).

  In particular, there were two blocking issues that we encountered during our attempt. First, PITest cannot handle a test suite in a completely separated package, loaded dynamically via OSGI and/or Tycho. Second (and related to the

12

first issue), the PITest test selection heuristics deciding which tests should be run first (see Section 3.2) could not find the tests due to dynamic loading of components.

- **Cheshire.** [http://github.com/AlFranzis/cheshire] As the next option, we explored the possibility to automatically convert the project into a non-OSGI one during the build. Cheshire is a prototype tool that provides an interface for OSGI-compliant software to resolve and retrieve dependencies during compile time. After e-mail communication with the developer of Cheshire, it was clear that many extra recipes should be written for the various kinds of dependencies used within the industrial case. According to the developer's estimation, it would take weeks for someone not familiar with the details of the component. Even then, the lack of previous experience with a combination of PITest and Cheshire made the final result unpredictable. Therefore, this solution was dismissed.

- **MuUnit.** [https://code.google.com/p/muunit] Finally, we tried to incorporate an OSGI-compliant tool to perform the mutation testing. The only prototype suitable for this task was MuUnit [39]. After a quick try-out it became clear that at the time of our analysis (which was September 2014) MuUnit was an early prototype able to run its analysis on simple projects only. Since then, there has been no development on this project, and as it stands, it can be considered an abandoned project. For this reason, this solution was dismissed.

- **Others.** We considered two other options, Jumble [http://jumble.sourceforge.net/] and Javalanche [http://www.st.cs.uni-saarland.de/mutation/]. A cursory analysis of these tools and their documentation revealed that the OSGI components would cause similar problems as we had with PITest, hence these options were dismissed as well.

Lessons Learned. This feasibility study demonstrated that—contrary to common wisdom—it is not that easy to integrate mutation testing into a complicated build process. This is caused by the inter-ference between the mutation testing (deeply coupled with the test infrastructure in order to speed up the process) and the product line configuration (with dynamic loading of test components). As often within software engineering, it is an accidental problem not an essential one [19]. Indeed, if the development team of the tools under investigation would choose to do so, they could probably engineer a solution. Yet, at the time of analysis the OSGI headers were too deeply embedded in the case under investigation to be handled by the tools available.

Due to aforementioned problems, Ali Parsai (the first author of the paper) developed and adapted a proof-of-concept tool called LittleDarwin, described in Section 3. LittleDarwin is explicitly designed to be loosely coupled to the test infrastructure, completely relying on the build system to run the tests. However, in doing so LittleDarwin forsakes the speed-up enabled by deep analysis of the test infrastructure and fast mutation injection via byte-code manipulation. We successfully applied LittleDarwin to perform the mutation testing on Segmentation. The problems we encountered using other tools were alleviated by the fact that LittleDarwin itself does not run the tests, but rather, the build system does. Therefore, the build system fetches the OSGI dependencies, and creates and configures the test harness. By using LittleDarwin to analyze Segmentation successfully, we demonstrated the feasibility of mutation testing within an industrial continuous integration environment.

> **RQ1 Summary**
> *Due to the accidental complexity, byte-level mutation tools such as PITest cannot easily be integrated into a complicated build process. Yet, if one decouples the mutation tool from the test infrastructure (thus relies on the build system to execute the tests) it is feasible to integrate mutation testing in a continuous integration setting. However, one does so at the expense of performance, i.e. mutation testing cannot be done as smartly and efficiently as the tightly coupled counterpart.*

RQ2. Does mutation testing reveal additional weaknesses in the test suite compared to branch coverage?

Motivation. In a development environment where software changes frequently and in an incremental manner, having an accurate view on the quality of the test suite is a necessity. Mutation coverage is a strong contender for this role, whereas branch coverage is more widely used in industry. On the one hand, mutation coverage has been demonstrated to subsume branch coverage [71, 57]. On the other hand, Gligoric et al. reports that among several coverage criteria, branch coverage is the best one to predict the mutation coverage of a test suite [35]. In this context, analyzing them together helps to determine whether or not mutation testing is capable of highlighting where branch coverage offers false confidence on the quality of the test suite.

Approach. Just as with RQ1, the unit of analysis is a class. We collect branch coverage via JaCoCo and mutation coverage via LittleDarwin for one industrial system and four open source systems listed in Table 4. For each class in the systems under study, we analyze the branch and mutation coverage, classifying them in the five categories shown in Table 5. By focusing on code sections characterized by the difference between branch coverage and mutation coverage, we show how mutation coverage exposes additional weaknesses. The similarity threshold we derived was $t = 11$ for the industrial case and $t = 7$ for the open source cases. We also calculate the Kendall correlation coefficient (τ_b) and the p-value.

Findings for the Industrial Case. The industrial case has 212 classes. During the analysis, 4,955 of the 12,825 generated mutants were killed by the test suite, resulting in 38.6% overall mutation coverage. Figure 4 shows the mutation coverage and branch coverage for each class. First of all, a large number of classes (48%) have zero branch coverage and mutation coverage, illustrating the inadequacy of the test suite. This is true considering that we focus on the unit tests only; indeed, there was a suite of

acceptance tests which did exercise most of the code but takes hours to execute. Secondly, there are several classes where the branch coverage and mutation coverage vary by a large margin.

The same data, organized according to the classification described in Section 4.4, is reported in Table 6. The table lists the number of classes for each of the five categories, as well as Kendall correlation coefficient (τ_b) and the p-value. First of all, we see that for 102 out of 212 classes (thus less than half) the branch and mutation coverage are the same for all practical purposes. For these classes, mutation testing does not provide additional value. Secondly, there are 8 classes in the category LoB-HiM, where the mutation coverage is larger than the branch coverage. Given the fact that LittleDarwin includes ROR mutation operator, at first sight it is expected that wherever there is mutation coverage, it should guarantee branch coverage. However, with deeper analysis we found this not to be true because there are lots of multi-branched methods that include many arithmetic operations only in a few of branches. The tests often target only these branches due to their perceived importance. Consequently, many of the large number of arithmetic mutants generated for these branches are killed, resulting in a high mutation coverage despite covering only few branches. Here, mutation testing provides extra confidence concerning the test suite; despite the low branch coverage, the test suite has a high coverage over fault-prone areas of the code. For these classes, mutation testing provides additional value. Most interestingly, there are 8 classes in the category HiB-LoM, where the mutation coverage is smaller than the branch coverage. There the mutation testing reveals weaknesses in the test suite; the high branch coverage gives a sense of false confidence regarding the test suite, even though the mutation coverage shows that some covered branches are indeed not adequately tested.

The most extreme case in this respect is the class *Discrete3DContour*, which has 88% branch coverage yet only 41% mutation coverage. For this class additional tests are needed, since the current tests cannot reveal injected faults. Most surprisingly (also apparent in Figure 4), there are 90 classes with zero branch coverage, yet some mutation coverage
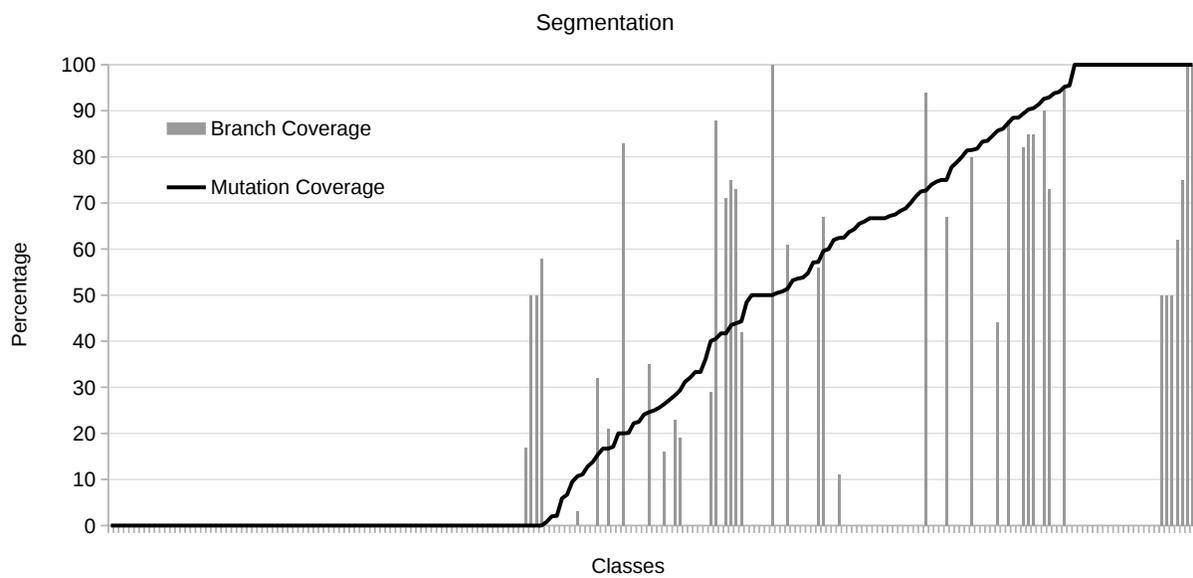
14

[The horizontal axis represents the classes sorted by mutation coverage; the vertical axis is the coverage percentage. Each bar represents the branch coverage for a class, and each point on the line represents the mutation coverage for a class.]

Figure 4: Visual comparison of branch coverage (JaCoCo) versus mutation coverage (LittleDarwin) on the industrial case
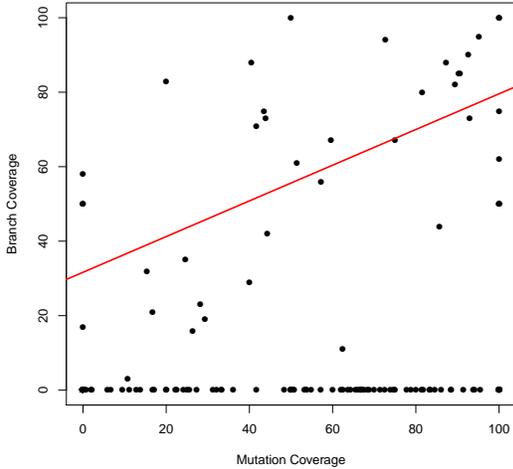
[The horizontal axis represents mutation coverage; the vertical axis represents branch coverage. The red line is the linear regression line.]

Figure 5: Weak correlation between branch coverage (JaCoCo) and mutation coverage (LittleDarwin) on the industrial case

(Category NoB). One representative example is the class *RegionGrowerNeighbours*; which has 0% branch coverage yet all 22 generated mutants are killed by the *VolumeGrowerTest* and *RegionGrowerNeighboursTest*. Both tests indirectly verify the algorithms provided by *RegionGrowerNeighbours*, and for this reason it is unlikely that the branch coverage is 0%. Manual inspection confirmed that here as well it was the dynamic loading of components by means of OSGI which leads to the loss of execution traces and thus results in the miscalculation of the branch coverage in JaCoCo (see Section 3.1). Finally, there were four classes with zero mutation coverage yet significant branch coverage (58%, 50%, 50% and 17% respectively). There the branch coverage creates an even higher sense of false confidence: there is branch coverage, yet the tests fail to reveal any faults.

Figure 5 shows the weak correlation between branch and mutation coverage values of all classes in the industrial case. While correlation cannot be ruled out (Kendall correlation of 0.25, p-value < 0.01), it shows that branch and mutation coverage correlate weakly at best. This may be partially attributed to the 90 classes with zero branch coverage (the horizontal line of dots at the bottom of the chart), which is an anomaly caused by the dynamic loading of OSGI components.

Findings for the Open Source Cases. Figure 6 shows the mutation and branch coverage values for all classes in the open source cases. Most strikingly, the results of the analysis for open source cases are quite different than from the industrial case. This is quite apparent in case of Joda Time, where branch coverage is providing more information than mutation coverage. Moreover, the low number of classes with no coverage at all shows that the open source cases are more adequately tested compared to the industrial case. The same data, organized according to the classification described in Section 4.4, is listed in Table 6. All four cases show some degree of branch coverage (NoB = 0); three of the four open source cases (Apache Commons Codec, jOpt Simple and AddThis Codec) also have some degree of mutation coverage (NoM <= 1). Here as well, Joda Time is the outlier (NoM = 70). Manual inspection revealed that a lot of the classes in Joda Time were implemented without any statements that could be mutated by the mutation operators of LittleDarwin (see Section 3.3). For example, some data classes include only variables and getter/setter methods, and therefore no mutant is generated for them. This illustrates that it might be worthwhile to expand the mutation operators beyond the reduced set listed in Table 1.

Looking at the column SimCov, we see that for most of the classes (ranging from 30% to 88%) the branch and mutation coverage are the same for all practical purposes. Thus for more than half of the classes, mutation testing does not provide additional value. This is quite different from the industrial case, where less than half of the classes had similar coverage values. Nevertheless, there are a significant number of classes in the category LoB-HiM, where the mutation coverage is larger than the branch coverage. Furthermore, there are also a significant number of
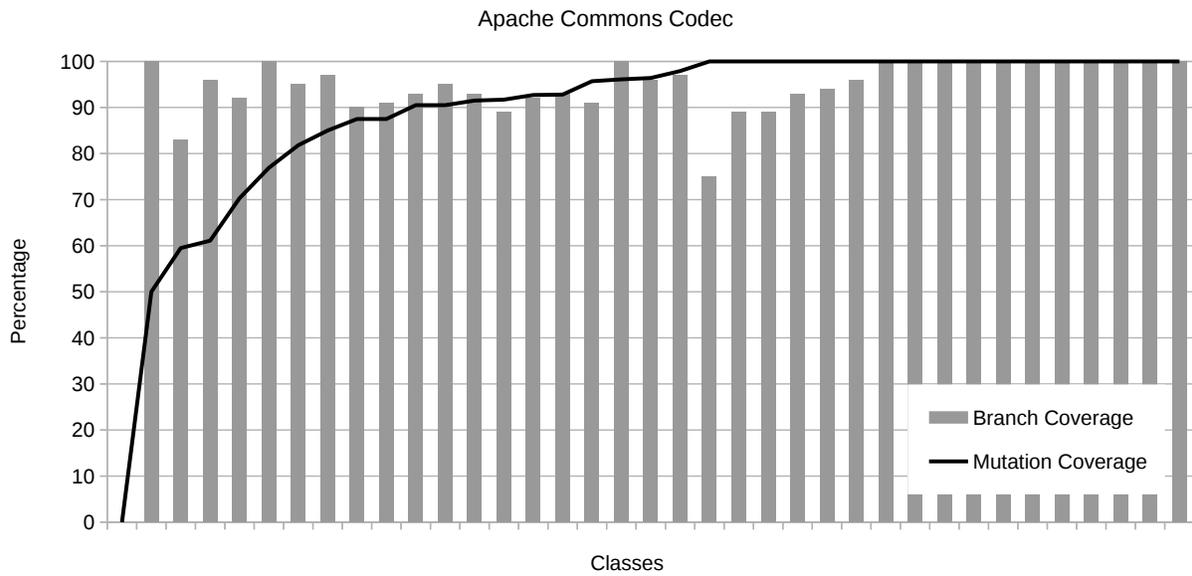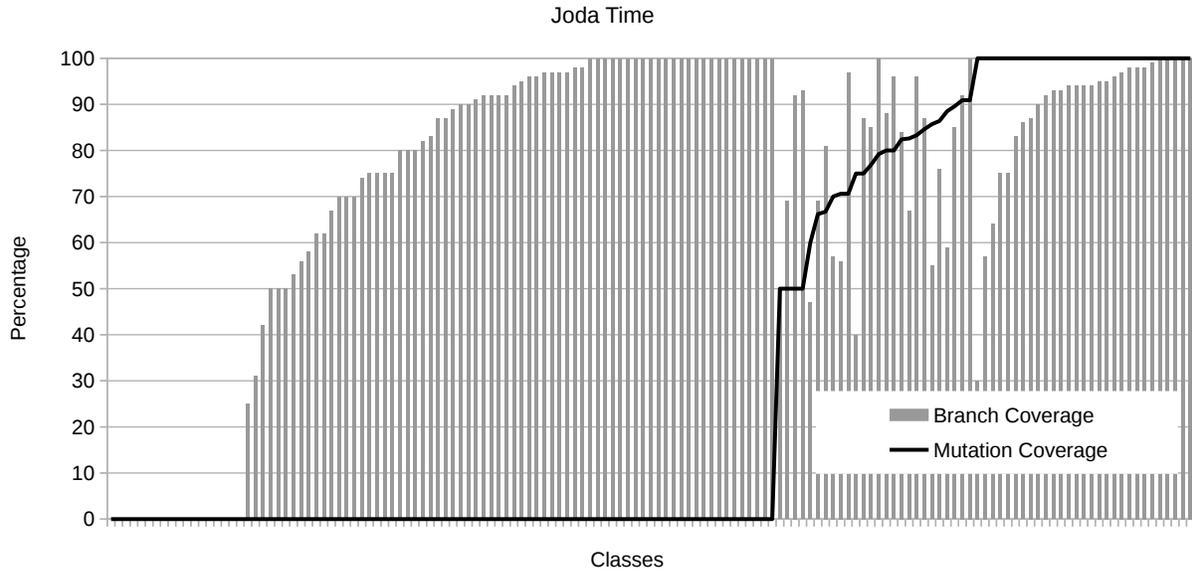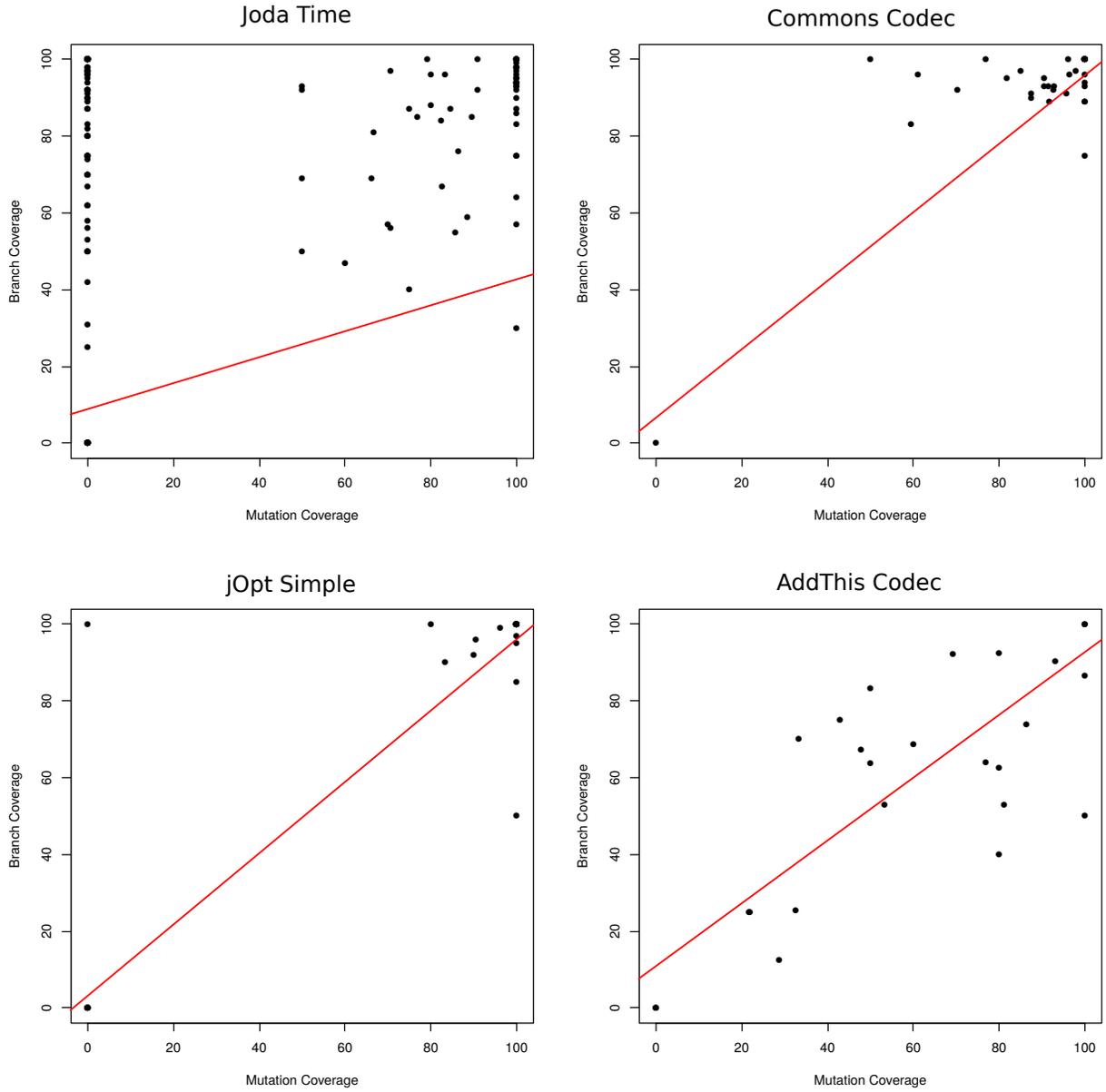
[The horizontal axis represents the classes sorted by mutation coverage; the vertical axis is the coverage percentage. Each bar represents the branch coverage for a class, and each point on the line represents the mutation coverage for a class.]

Figure 6: Branch coverage (JaCoCo) and mutation coverage (LittleDarwin) at class level for the open source cases
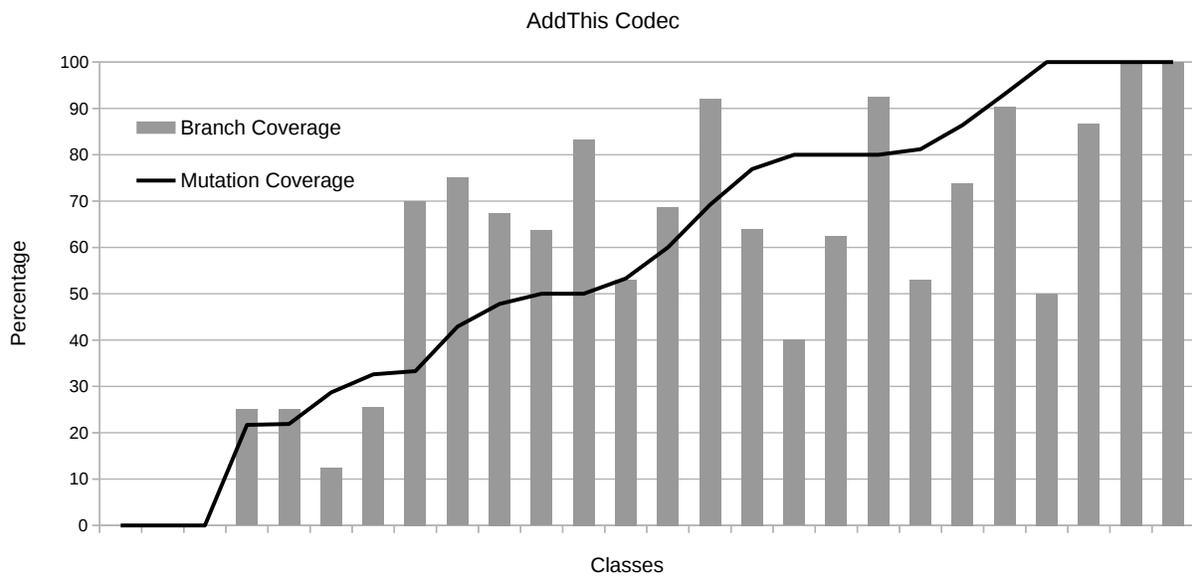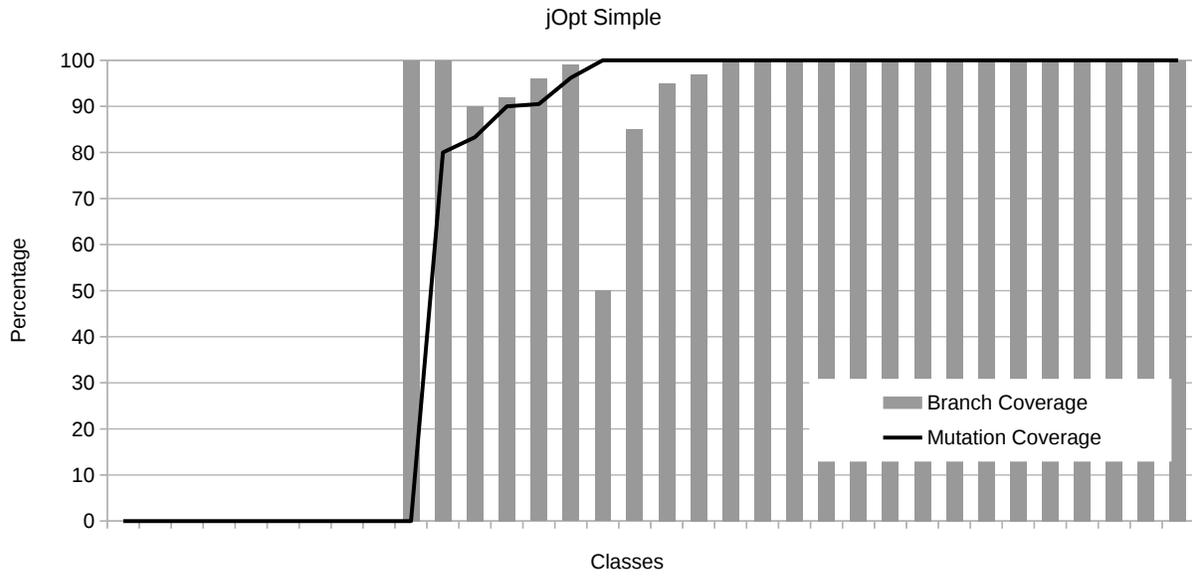
[The horizontal axis represents the classes sorted by mutation coverage; the vertical axis is the coverage percentage. Each bar represents the branch coverage for a class, and each point on the line represents the mutation coverage for a class.]

Figure 6: (Continued from previous page) Branch coverage (JaCoCo) and mutation coverage (LittleDarwin) at class level for the open source cases

[The horizontal axis represents mutation coverage; the vertical axis represents branch coverage. The red line is the linear regression line.]

Figure 7: Correlation between branch coverage (JaCoCo) and mutation coverage (LittleDarwin) on the open source cases

19

Table 6: Comparing branch coverage (JaCoCo) versus mutation coverage (LittleDarwin) on cases under study

Case	Categorization					Correlation	
	SimCov	LoB-HiM	HiB-LoM	NoB	NoM	Kendall τ_b	p-value
Industrial Case							
Segmentation	102	8	8	90	4	0.25	8.951×10^{-6}
Open Source Cases							
Joda Time	43	18	12	0	70	0.11	1.132×10^{-1}
Apache Commons Codec	27	3	7	0	0	0.31	1.755×10^{-2}
jOpt Simple	30	2	1	0	1	0.71	2.863×10^{-6}
AddThis Codec	9	9	8	0	0	0.53	2.146×10^{-4}
Total	211	40	36	90	75	N/A	

classes in the category HiB-LoM, where the mutation coverage is smaller than the branch coverage. For three of the four cases, the values in column HiB-LoM are larger than the values in column LoB-HiM and here as well `Joda Time` is the exception. Thus, although much less than in the industrial case, there are still a significant number of classes where mutation tests reveals additional weaknesses.

Analyzing the correlation between branch and mutation (i.e. columns "Kendall τ_b" and "p-value") we see that the correlation between branch and mutation coverage is rather poor. Figure 7 provides insight into the lack of correlation. For each of the cases the dots in the scatter plot are in distinct regions, hence the coverage values depend a lot on the particular context of the case under investigation.

Lessons Learned. Comparison of branch coverage and mutation coverage seen in Table 6 shows that these measures agree only for 47% of analyzed classes. In 9% of classes, we observed that the density of the mutants were much higher in few branches in the code, and thus despite a low branch coverage, the mutation coverage is much higher. Conversely, in 8% of cases where branch coverage is high and mutation coverage is low, the quality of the test oracles are in question. In the remaining 36% of classes, the peculiarities of the code cause problems in calculation of either metric: In case of branch coverage, complicated structure of Segmentation means that it cannot be accurately calculated, and this is in line with the observations of Tengeri et al. [89]. In case of mutation coverage, the abundance of small data classes or interfaces and stubs in Joda Time means that a more

extensive set of mutation operators is required. All things considered, such analysis on the weaknesses in the tests cannot be done without mutation testing, and it is clear that using branch coverage alone can mislead a developer about quality of the tests.

> *RQ2 Summary*
> *For all the cases under investigation, we discovered that significant number of classes where mutation testing reveals additional weaknesses compared to branch coverage. However, the added value of the mutation testing is context-dependent and varies between the cases we investigated. This warrants further research into the nature of lower coverage values both for branch coverage and mutation coverage.*

RQ3. Can we reduce the performance overhead induced by mutation testing to an acceptable level?

Motivation. The results of RQ2 shows that mutation testing provides valuable complementary information over branch coverage. However, in order to avoid issues connected with specifics of the system architecture, the results of RQ1 suggest that we need to do so with mutation testing tools loosely coupled to the test infrastructure, thus inherently slower. This needs to be mitigated if mutation testing is to be incorporated into development environments with small, frequent, and incremental changes to the soft-
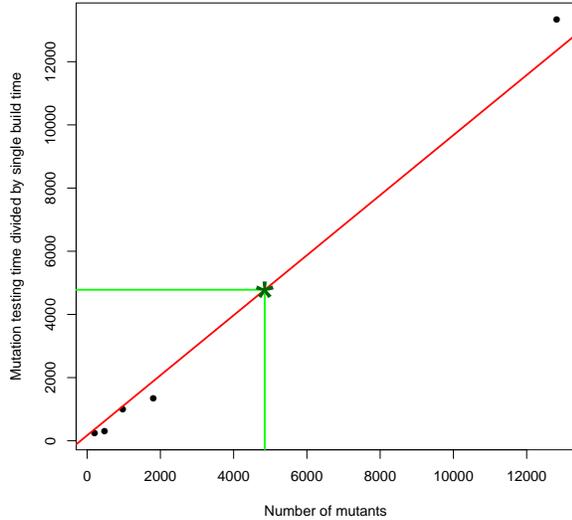
Figure 8: Linear relationship between number of mutants and total analysis time
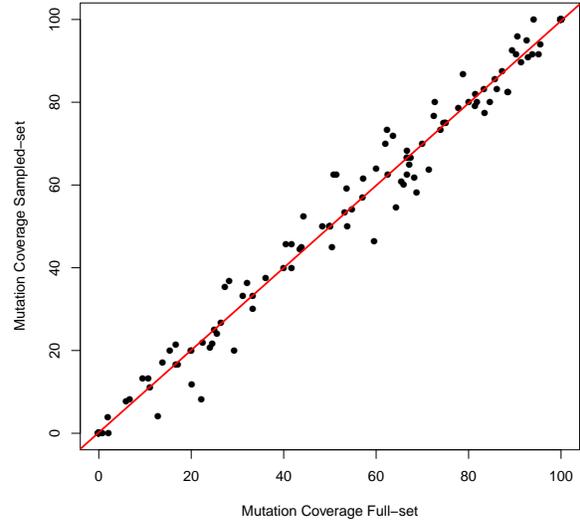

Figure 9: Correlation of full mutation coverage against mutation coverage with weighted sampling

ware. For this reason, in this RQ we want to tackle the "time concern"; demonstrating that (i) the performance overhead induced by mutation testing can be reduced to meet industrial time constraints and (ii) sampled mutation testing preserves the information presented in RQ2. As a realistic scenario we consider a team that starts working from Monday at 8am till Friday at 6pm. Here, we want to verify whether a complete mutation testing could run once a week during the week-end and right before the sprint meeting scheduled on Monday morning. In this context, we define an acceptable level of performance overhead as a mutation testing job that runs in up to 62 hours, namely between Friday 6pm to Monday 8am. If the full mutation testing cycle takes longer, we use *mutant sampling* to reduce the number of mutants injected into the system, because it has been demonstrated that even with a sample size as low as 50% mutation tests still provides reliable results [97].

Approach. We first measure the performance overhead induced by a full mutation testing of the indus-

trial case (injecting 12,825 mutants for 38K lines of code). Here we calculate the average time for a single iteration by dividing the total time for the process by the number of mutants. Next, we estimate the sample size based on the average single iteration time so that the total analysis time would drop below the 62 hour maximum value. We then compare the coverage values for the full mutation testing and the sampled mutation testing with the same categories as RQ2.

To settle the sampling rate, we used the following procedure. We first analyze the relationship between the number of mutants, and the time required for the analysis. In principle, this should be a linear relationship, which is confirmed in Figure 8. Via a linear regression ($\rho = 0.9992$, p-value $= 0.000026$) we achieve the slope of the regression line as 0.9506 with an offset value of 169.5 (Equation 3). Using this equation, we can estimate the upper limit for the number of mutants. The total time needed to perform the analysis is $= 62$ hours; a single iteration takes 46 seconds, thus we get as an upper limit for the number of mutants $4,780$.

21
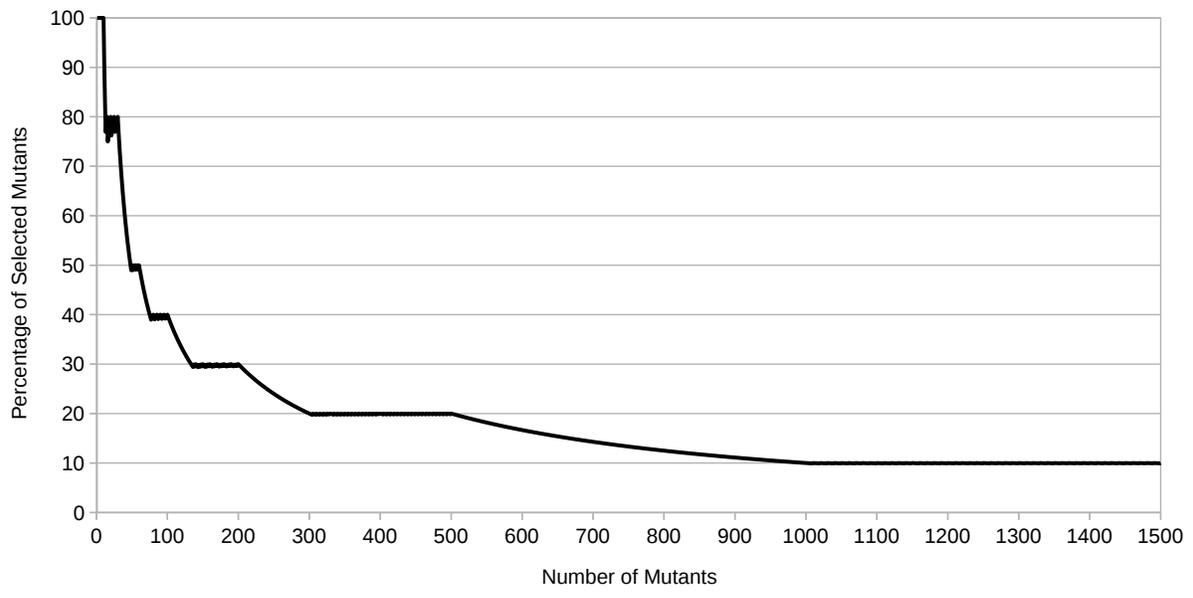
[The horizontal axis shows the number of mutants generated for a class;
the vertical axis shows the percentage of selected mutants.]

Figure 10: The distribution of the *weighted mutation sampling*

$$Number\ of\ mutants = 169.5 + 0.9506 \times \frac{Total\ analysis\ time}{Time\ for\ a\ single\ iteration} \quad (3)$$

To guarantee that the classes with smaller set of mutants are still represented in the sample set, we used a procedure called *weighted mutation sampling* [77], and we performed the random sampling at class-level rather than project level. The weights are chosen based on the size of the mutant set for each class, thus reduces the masking effect of classes with larger set of mutants in the final coverage score. This way, our sampling method randomly selected 4,448 mutants out of 12,825 generated mutants (34.7%). The percentage of mutants selected based on the size of the mutant set is shown in Figure 10.

Findings. The full mutation testing of the industrial case (i.e. the Segmentation component) takes 163 hours to complete; way more than the established upper limit of 62 hours. Via the weighted sampling method, we reduced the number of mutants from $12,825$ to $4,448$ ($34,6\%$). With this reduction, the time required to perform the mutation testing dropped to 58 hours (almost one-third of the full analysis) well under the established upper limit.

Out of the $4,448$ mutants in the sampled mutant set, $1,721$ were killed by the test suite, resulting in a 38.7% overall mutation coverage. The difference between the overall coverage calculated from the sampled set and the one calculated from the full set is only 0.1%. Table 7 classifies the differences in mutation scores using the categories in Table 5. The first row shows the values for the weighted mutation sampling, the second row (copied from 6) show the values for full mutation testing. We see that the number of classes in each category remains roughly the same, thus confirming that sampling does not diminish the confidence in the validity of mutation coverage. This is confirmed in Figure 9, which shows the correlation between the full mutation coverage values and the sampled ones; they do indeed have a very strong correlation ($\rho = 0.99$, p-value < 0.00001).

> **RQ3 Summary**
> *When a full mutation testing exceeds the time limitations imposed by a continuous integration setting (i.e. an analysis once a week during the week-end), we can use weighted mutation sampling to reduce the performance overhead and at the same time preserve an accurate estimation of the mutation coverage.*

6 Threats to Validity

We now identify factors that may jeopardize the validity of our results and the actions we took to reduce or alleviate the risk. Consistent with the guidelines for case studies research (see [81]) we organize them into four categories.

Internal Validity.

Threats to internal validity focus on confounding factors that can influence the obtained results. In this study, this mainly concerns equivalent mutants and a limited set of mutation operators. Because of the large number of generated mutants, it is very difficult to check for equivalent mutants in the final generated results due to the amount of manual labor needed to find and remove such mutants. Nevertheless, since equivalent mutants would add to false positives, they would be discovered when the developers are trying to create new tests or improve the available tests by referring to the information acquired from mutation testing. As is common practice in today's research, we just accept the risk [28].

Another threat stems from the limited set of mutation operators used in LittleDarwin. The addition of more mutation operators does not impact the results of our industrial case study, where mutation coverage (even with the limited set of mutation operators) produced more information than branch coverage. However, addition or removal of mutation operators can have an adverse effect on the quality of mutant sampling, since there are mutation operators that can produce a large number of redundant mutants (e.g. Null-Type mutation operators [75]), and thus affect the distribution of the sampled mutants. To identify

Table 7: Comparing branch coverage (JaCoCo) versus mutation coverage (LittleDarwin)—full coverage + weighted sampling)

Case	Categorization					Correlation	
	SimCov	LoB-HiM	HiB-LoM	NoB	NoM	Kendall τ_b	p-value
Industrial Case (full)	102	8	8	90	4	0.25	8.951×10^{-6}
Industrial Case (sampled)	102	10	9	87	4	0.26	4.106×10^{-6}

such effects in practice, further case studies on industrial software is required; nonetheless, we did not pursue this line of research in this study.

A large majority of the produced mutants are indeed redundant, and this affects the stability of mutation coverage as a metric [72]. In order to remove the redundant mutants, mutant subsumption relationships need to be used [53]. However, determining these relationships is a very difficult task at large scale. There are no available tools to our knowledge that performs static subsumption analysis on Java programs of this scale. In addition, dynamic subsumption analysis requires a very high quality test suite to be accurate. Therefore in case of non-adequate test suites of our subject projects, the dynamic subsumption relationships are unreliable to detect redundancy among mutants [7]. For this reason, we did not filter redundant mutants in this study.

During the course of the study we discovered that JaCoCo does not report branch coverage correctly in some cases. This phenomenon has been already documented in Tengeri et al. [89, 16]. Despite this fact, we decided to use the results as is for two reasons: first, we were informed by the developers of the industrial project that no other tool was capable of integration with their environment, and second, the results of the tool were used as is for the decision making process regarding the testing of the software. Given the fact that JaCoCo is one of the most commonly used tools in maven builds, we believe its weaknesses are a reflection of the difficulties in computing branch coverage in practice, and therefore worth studying.

Construct Validity.

Threats to construct validity focus on how accurately the observations describe the phenomena of interest. This research is driven by RQ2 where we compare test coverage provided by two different tools. To minimize the risk on making wrong observations, we compare the test coverage in two different ways, once using the categories in Table 5 and once via the Kendall's τ_b coefficient as described in statistic handbooks [2, 5]. Moreover, we manually inspect certain results, especially outliers.

In order to evaluate mutant sampling process, a common procedure is to create many smaller subsets of a test suite, and compare the mutation coverage obtained from sampled mutants and the mutation coverage obtained from all mutants for all of these smaller subsets of the test suite. Despite our attempts, this proved not to be possible in the industrial case. Because of the interdependencies between the tests, complicated setup process of the testing harness, and the use of shared resources, the order in which the tests are included and executed are important for a successful execution of the test suite. Therefore, it is not possible to randomly generate subsets of the original test suite. For this reason, we omitted this kind of analysis in our study.

Reliability.

Threats to reliability validity correspond to the degree to which the result depends on the tools used. The most important threat to validity concerns the tool LittleDarwin implemented by first author. Compile errors and errors in tests can affect the final results, since LittleDarwin checks only if the build process has failed or not, and it does not go further to determine the reason for the failure. This was addressed by inspecting the output of the build system. In this process, 107 invalid mutants were detected. These mutants could not be compiled, and were excluded from the final results. Another threat to reliability validity is the fact that the data gathered

by JaCoCo might not be accurate (especially due to dynamic loading of components), and therefore the conclusions based on the comparison between branch coverage and mutation coverage might not be accurate enough. However, since JaCoCo is the tool that is being used in the structure of Segmentation component to acquire this information in the first place, the conclusions are still relevant by challenging the previously held beliefs about the system. Because of active development of JaCoCo and its popularity as an integral part of continuous integration systems, it is debatable whether better accuracy can be achieved.

External Validity.

Threats to external validity correspond to the generalizability of our results. Since this study was performed only on a software running on a single platform with a specific target language and a specific continuous integration environment. Therefore, the results are certainly not representative for all possible industrial systems. However, it provides an outlook on the feasibility of applying mutation testing in an industrial environment, especially concerning the challenges we faced and workarounds we performed. Nevertheless, we partially addressed the generalizability by extending the analysis regrading RQ2 on other open source cases. By comparing these results with those of the industrial case, we determined the situations in which our conclusions can be generalized.

7   Related Work

There are several studies that assess the effectiveness of test coverage metrics. Offutt and Voas [71] use generated test cases for Fortran, and conclude mutation coverage subsumes condition coverage techniques. Li and Offutt [57] compare mutation coverage with edge-pair, all-uses, and prime path coverage. They use hand-seeded faults and manually developed test cases in their comparison, and conclude that mutation coverage is more effective in detecting faults, and requires a smaller test suite to be satisfied. Gligoric et al. [35] compare non-adequate test suites by using several coverage criteria, and conclude that branch coverage is the best predictor of mutation coverage. Gopinath et al. [36] compare coverage criteria that is available to developers on a large set of open source cases, and conclude that statement coverage, and not branch coverage, is the best predictor of mutation coverage.

Literature is clear on the dangers of using code coverage as a threshold for quality of the test suite. Marick [63] points out a scenario that relying solely on the code coverage metrics could result in faults not being detected. Inozemtseva and Holmes [44] compared decision coverage, modified condition coverage, and statement coverage on large subjects using generated test cases and mutation coverage as the effectiveness criteria, and concluded that while code coverage is good for identifying under-tested parts of the subject, it should not be used as a quality target. Aaltonen et al. [1] reach the same conclusion in their analysis comparing mutation coverage and code coverage metrics in assessment of students' skills. They propose adoption of both metrics in order to have an accurate assessment of the test suite quality. Smith and Williams [86, 87] studied the effects of using mutation testing to augment a test suite. They concluded that developing new test cases that increase the mutation coverage also increases branch coverage, and statement coverage of the test suite. They also conclude that the inclusion of new mutation operators is less important than the speed and efficiency of mutation testing process. Andrews et al. [10] validate the use of mutation testing as a benchmark for other coverage criteria using an industrial case with known faults, and conclude that not only mutation testing can be used in a research context, but also in a practical context, it can be used as a threshold to develop new test cases. Li et al. [58] use industrial cases written in Ruby and demonstrate that using mutation testing still adds value to a test suite that has 97% statement coverage.

8   Conclusion and Future Work

With the increasing interest in continuous integration and development environments where changes

to the code occur in small and frequent steps, developers rely fully automated tests to find faults as early as possible. Therefore, they require to continuously monitor the coverage of their automated test suites. Unfortunately, the state of the practice is reluctant to adopt strong coverage metrics (namely mutation coverage), instead relying on weaker kinds of coverage (namely branch coverage). We argue that there are three issues for this reluctant attitude towards mutation testing: (a) the complexity of continuous build environments and (b) the perception that branch coverage is "good enough"; (c) the performance overhead during the build. Consequently, we set out to investigate the pros and cons that arise when adopting mutation testing in an industrial continuous integration setting, namely the Segmentation component of the Impax ES medical imaging software used by Agfa HealthCare. The Impax ES system is configured as a product-line released in two main variants (production or prototype), with a few minor variants for the target hardware platform. The extensive use of (dynamic) OSGI headers implies a complicated build process where the Maven plug-in Tycho [https://eclipse.org/tycho/] is used to fetch dependencies, compile source files, and run the test suite. This lead us to pursue the following research questions.

RQ1: *Is it feasible to integrate mutation testing in a continuous integration system?*

- Byte-level mutation tools such as PITest cannot be easily integrated into a complicated build process. Yet, if one decouples the mutation tool from the test infrastructure (thus relies on the build system to manipulate the tests) it is feasible to integrate mutation testing in a continuous integration setting. However, one does so at the expense of performance; that is to say mutation testing cannot be implemented as smartly and efficiently as the tightly coupled counterpart.

RQ2: *Does mutation testing reveal additional weaknesses in the test suite compared to branch coverage?*

- We discovered that mutation coverage reveals additional weaknesses compared to branch coverage in all of the investigated cases. More specifically, there were several classes which had a high branch coverage yet a low mutation coverage, hence in such situations branch coverage gives a false sense of confidence. However, the added value of the mutation coverage is context dependent and varies between the cases we investigated. This warrants further research into the nature of weak coverage values both for branch and mutation coverage. Especially because here too we noticed that dynamic loading of components interferes with calculating branch coverage.

RQ3: *Can we reduce the performance overhead induced by mutation testing to an acceptable level?*

- When a full mutation testing exceeds the time limitations imposed by a continuous integration setting (i.e. an analysis once a week during the week-end), we can reduce the performance overhead by means of weighted mutation sampling without sacrificing the fault detection capability.

The contributions of this research are fourfold. First, we report the challenges that arise when mutation testing is integrated in a continuous integration tool of a real industrial case. Second, we adapt and use our mutation testing tool called LittleDarwin to overcome these challenges. Third, we perform a joint analysis of mutation and branch coverage on cases with non-adequate test suites. For this analysis we used four open source cases (totaling close to 40 thousand lines of code with varying degrees of test coverage) and one industrial case (with more than 38 thousand lines of code with limited unit test coverage). We demonstrate that in cases without full branch test adequacy, mutation testing does not subsume (as expected) branch coverage. Yet, it provides complementary information that can be exploited for both determining the fault detection ability of the test suite, and forming a long-term plan to improve it. We also did not find sufficient evidence to support previous conclusions in literature regarding the relation of branch coverage and mutation coverage, namely, we cannot confirm that branch coverage is a good estimator of mutation coverage in complicated systems. Fourth, we describe how to adapt mutation testing in order to satisfy industrial time constraints and yet preserve its ability to evaluate the quality of test suite, therefore making it useful in a development

environment with small, frequent changes.

Future Work. There are several ideas following this study that are worthy of further investigation. In this work, we investigate whether mutation testing reveals more weaknesses in the test suite compared to branch coverage. Given the fact that the ultimate goal of software testing is to reveal as many faults as possible, it is interesting to investigate whether test suites optimized for mutation coverage are more successful in finding faults when compared to test suites optimized for branch coverage. Similarly, the validity of mutant sampling can be investigated deeper by comparing the fault detection capability of test suites optimized for sampled and full-set of mutants. In addition, including more mutation operators to increase the density of mutants in code might in turn increase the efficiency of random sampling. Another prospective research topic is to replicate the results of this study using a minimal set of mutants by detecting the redundant mutants through subsumption analysis. The feasibility of performing subsumption analysis on large industrial software is still in question, and therefore, worth further study.

Acknowledgements

References

[1] Kalle Aaltonen, Petri Ihantola, and Otto Seppälä. Mutation analysis vs. code coverage in automated assessment of students' testing skills. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion - SPLASH 10*, OOPSLA '10, pages 153–160, New York, NY, USA, 2010. ACM Press.

[2] Hervé Abdi. Kendall rank correlation coefficient. In *The Concise Encyclopedia of Statistics*, pages 278–281. Springer New York, 2007.

[3] Robin Abraham and Martin Erwig. Mutation operators for spreadsheets. *IEEE Transactions on Software Engineering*, 35(1):94–108, jan 2009.

[4] Allen Troy Acree Jr. *On Mutation*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, USA, 1980.

[5] Alan Agresti. *Analysis of Ordinal Categorical Data*, volume 656. John Wiley & Sons, Inc., mar 2010.

[6] Zaheed Ahmed, Muhammad Zahoor, and Irfan Younas. Mutation operators for object-oriented systems: A survey. In *2010 The 2nd International Conference on Computer and Automation Engineering (ICCAE)*, volume 2, pages 614–618. IEEE, feb 2010.

[7] Paul Ammann, Marcio Eduardo Delamaro, and Jeff Offutt. Establishing theoretical minimal sets of mutants. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 21–30. IEEE, mar 2014.

[8] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2 edition, dec 2016.

[9] James H. Andrews, Lionel C. Briand, and Yvan Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th international conference on Software engineering - ICSE 05*, ICSE '05, pages 402–411, New York, NY, USA, 2005. ACM Press.

[10] James H. Andrews, Lionel C. Briand, Yvan Labiche, and Akbar Siami Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, 32(8):608–624, aug 2006.

[11] B. Assylbekov, E. Gaspar, N. Uddin, and P. Egan. Investigating the correlation between mutation score and coverage score. In *2013 UK-Sim 15th International Conference on Computer Modelling and Simulation*, pages 347–352. IEEE, apr 2013.

[12] Benoit Baudry, Franck Fleurey, and Yves Le Traon. Improving test suites for efficient fault localization. In *Proceeding of the 28th international conference on Software engineering - ICSE 06*, ICSE '06, pages 82–91, New York, NY, USA, 2006. ACM Press.

[13] Sebastian Bauersfeld, Tanja E. J. Vos, and Kiran Lakhotia. Unit testing tool competitions – lessons learned. In E.J. Tanja Vos, Kiran Lakhotia, and Sebastian Bauersfeld, editors, *Future Internet Testing: First International Workshop, FITTEST 2013, Istanbul, Turkey, November 12, 2013, Revised Selected Papers*, pages 75–94. Springer International Publishing, Cham, 2014.

[14] K. Beck. *Test Driven Development: By Example.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[15] Kristian Bjerke-Gulstuen, Emil Wiik Larsen, Tor Stålhane, and Torgeir Dingsøyr. High level test driven development – shift left. In Casper Lassenius, Torgeir Dingsøyr, and Maria Paasivaara, editors, *Agile Processes in Software Engineering and Extreme Programming*, pages 239–247, Cham, 2015. Springer International Publishing.

[16] Vincent Blondeau, Anne Etien, Nicolas Anquetil, Sylvain Cresson, Pascal Croisy, and Stéphane Ducasse. Test case selection in industry: an analysis of issues related to static approaches. *Software Quality Journal*, 25(4):1203–1237, Dec 2017.

[17] Grady Booch. *Object Oriented Design: With Applications.* The Benjamin/Cummings Series in Ada and Software Engineering. Benjamin/Cummings Pub., 1991.

[18] Jeremy S. Bradbury, James R. Cordy, and Juergen Dingel. ExMAn: A generic and customizable framework for experimental mutation analysis. In *Second Workshop on Mutation Analysis (Mutation 2006 - ISSRE Workshops 2006)*, MUTATION '06, page 4, Washington, DC, USA, nov 2006. IEEE.

[19] Frederick P. Brooks. No silver bullet essence and accidents of software engineering. *Computer*, 20(4):10–19, apr 1987.

[20] Benjamin Brosgol. Do-178C: The next avionics safety standard. *ACM SIGAda Ada Letters*, 31(3):5–6, nov 2011.

[21] Timothy Alan Budd. *Mutation Analysis of Program Test Data.* PhD thesis, Yale University, New Haven, CT, USA, 1980. AAI8025191.

[22] Boyuan Chen, Jian Song, Peng Xu, Xing Hu, and Zhen Ming (Jack) Jiang. An automated approach to estimating code coverage measures via execution logs. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 305–316, New York, NY, USA, 2018. ACM.

[23] Huo Yan Chen and Su Hu. Two new kinds of class level mutants for object-oriented programs. In *2006 IEEE International Conference on Systems, Man and Cybernetics*, volume 3, pages 2173–2178. IEEE, oct 2006.

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

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

[26] Anna Derezińska and Marcin Rudnik. Quality evaluation of object-oriented and standard mutation operators applied to c# programs. In Carlo A. Furia and Sebastian Nanz, editors, *Objects, Models, Components, Patterns*, pages 42–57. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

28

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

[28] Tom Fawcett. An introduction to ROC analysis. *Pattern Recognition Letters*, 27(8):861–874, jun 2006. ROC Analysis in Pattern Recognition.

[29] Martin Fowler. Continuous integration. Technical report, http://www.martinfowler.com/, May 2006. http://www.martinfowler.com/articles/continuousIntegration.html.

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

[31] Phyllis G. Frankl, Stewart N. Weiss, and Cang Hu. All-uses vs mutation testing: An experimental comparison of effectiveness. *Journal of Systems and Software*, 38(3):235–253, sep 1997.

[32] Gordon Fraser and Andreas Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering*, 38(2):278–292, mar 2012.

[33] Vahid Garousi and Erdem Yildirim. Introducing automated GUI testing and observing its benefits: An industrial case study in the context of law-practice management software. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops – NEXTA Workshop*, pages 138–145. IEEE, apr 2018.

[34] Gregory Gay, Matt Staats, Michael Whalen, and Mats P. E. Heimdahl. The risks of coverage-directed test case generation. *IEEE Transactions on Software Engineering*, 41(8):803–819, aug 2015.

[35] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. Comparing non-adequate test suites using coverage criteria. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis - ISSTA 2013*, ISSTA 2013, pages 302–313, New York, NY, USA, 2013. ACM Press.

[36] Rahul Gopinath, Carlos Jensen, and Alex Groce. Code coverage for suite evaluation by developers. In *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, ICSE 2014, pages 72–82, New York, NY, USA, 2014. ACM Press.

[37] Bernhard J. M. Grün, David Schuler, and Andreas Zeller. The impact of equivalent mutants. In *2009 International Conference on Software Testing, Verification, and Validation Workshops – Mutation Workshop*, pages 192–199. IEEE, April 2009.

[38] Mark Harman, Yue Jia, and William B. Langdon. Strong higher order mutation-based test data generation. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering - SIGSOFT/FSE 11*, ESEC/FSE '11, pages 212–222, New York, NY, USA, 2011. ACM Press.

[39] Siamak Haschemi and Stephan Weißleder. A generic approach to run mutation analysis. In Leonardo Bottaci and Gordon Fraser, editors, *Testing – Practice and Research Techniques*, volume 6303 of *Lecture Notes in Computer Science*, pages 155–164. Springer Berlin Heidelberg, 2010.

[40] K.J. Hayhurst and D.S. Veerhusen. A practical approach to modified condition/decision coverage. In *20th DASC. 20th Digital Avionics Systems Conference (Cat. No.01CH37219)*, volume 1, pages 1B2/1–1B2/10 vol.1. IEEE, Oct 2001.

[41] Hadi Hemmati. How effective are code coverage criteria? In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 151–156. IEEE, aug 2015.

[42] Richard Hopkins and Kevin Jenkins. *Eating the IT Elephant: Moving from Greenfield Development to Brownfield*. IBM Press, 2008.

[43] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria. In *Proceedings of 16th International Conference on Software Engineering*, ICSE '94, pages 191–200, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[44] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, ICSE 2014, pages 435–445, New York, NY, USA, 2014. ACM Press.

[45] ISO. Road vehicles – Functional safety, 2011.

[46] Werner Janjic and Colin Atkinson. Utilizing software reuse experience for automated test recommendation. In *2013 8th International Workshop on Automation of Software Test (AST)*, AST '13, pages 100–106, Piscataway, NJ, USA, may 2013. IEEE.

[47] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, sep 2011.

[48] René Just. The Major mutation framework: Efficient and scalable mutation analysis for java. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014*, ISSTA 2014, pages 433–436, New York, NY, USA, 2014. ACM Press.

[49] Tomek Kaczanowski. *Practical Unit Testing with TestNG and Mockito*. Tomasz Kaczanowski, Cracow, 2012.

[50] Susanne Kandl and Sandeep Chandrashekar. Reasonability of MC/DC for safety-relevant software implemented in programming languages with short-circuit evaluation. *Computing*, 97(3):261–279, aug 2014.

[51] Kim N. King and A. Jefferson Offutt. A fortran language system for mutation-based software testing. *Software: Practice and Experience*, 21(7):685–718, jul 1991.

[52] Jeshua S. Kracht, Jacob Z. Petrovic, and Kristen R. Walcott-Justice. Empirically evaluating the quality of automatically generated and manually written test suites. In *2014 14th International Conference on Quality Software*, pages 256–265. IEEE, oct 2014.

[53] Bob Kurtz, Paul Ammann, Marcio E. Delamaro, Jeff Offutt, and Lin Deng. Mutant subsumption graphs. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation Workshops – Mutation Workshop*, ICST '14, pages 176–185, Washington, DC, USA, 2014. IEEE Computer Society.

[54] Hyo-Jeong Lee, Yu-Seong Ma, and Yong-Rae Kwon. Empirical evaluation of orthogonality of class mutation operators. In *11th Asia-Pacific Software Engineering Conference*, APSEC '04, pages 512–518, Washington, DC, USA, 2004. IEEE.

[55] Nan Li, Xin Meng, Jeff Offutt, and Lin Deng. Is bytecode instrumentation as good as source code instrumentation: An empirical study with industrial tools (experience report). In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pages 380–389. IEEE, nov 2013.

[56] Nan Li and Jeff Offutt. Test oracle strategies for model-based testing. *IEEE Transactions on Software Engineering*, 43(4):372–395, apr 2017.

[57] Nan Li, Upsorn Praphamontripong, and Jeff Offutt. An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage. In *2009 International Conference on Software Testing, Verification, and Validation Workshops – Mutation Workshop*, pages 220–229. IEEE, April 2009.

[58] Nan Li, Michael West, Anthony Escalona, and Vinicius H. S. Durelli. Mutation testing in practice using ruby. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops – Mutation Workshop*, pages 1–6. IEEE, apr 2015.

[59] Yu-Seung Ma, Yong-Rae Kwon, and Jeff Offutt. Inter-class mutation operators for java. In *13th International Symposium on Software Reliability Engineering, 2002. Proceedings.*, pages 352–363. IEEE Comput. Soc, 2002.

[60] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. MuJava. In *Proceeding of the 28th international conference on Software engineering - ICSE 06*, ICSE '06, pages 827–830, New York, NY, USA, 2006. ACM Press.

[61] Lech Madeyski, Wojciech Orzeszyna, Richard Torkar, and Mariusz Jozala. Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation. *IEEE Transactions on Software Engineering*, 40(1):23–42, jan 2014.

[62] Brian Marick. Experience with the cost of different coverage goals for testing. In *Proceedings of the Ninth Pacific Northwest Software Quality Conference*, pages 147–164, 1991.

[63] Brian Marick and Testing Foundations. How to misuse code coverage. In *16th International Conference and Exposition on Testing Computer Software*, 1999.

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

[65] J. D. McGregor. Test early, test often. *Journal of Object Technology*, 6(4):7–14, May 2007. (column).

[66] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. Taming google-scale continuous testing.

In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*, ICSE-SEIP '17, pages 233–242, Piscataway, NJ, USA, 2017. IEEE Press.

[67] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology*, 5(2):99–118, apr 1996.

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

[69] A. Jefferson Offutt, Jie Pan, Kanupriya Tewary, and Tong Zhang. An experimental evaluation of data flow and mutation testing. *Software: Practice and Experience*, 26(2):165–176, feb 1996.

[70] A. Jefferson Offutt and Roland H. Untch. Mutation 2000: Uniting the orthogonal. In W.Eric Wong, editor, *Mutation Testing for the New Century*, volume 24 of *The Springer International Series on Advances in Database Systems*, pages 34–44. Springer US, 2001.

[71] A. Jefferson Offutt and Jeffrey M. Voas. Subsumption of condition coverage techniques by mutation testing. Technical report, George Mason University, 1996.

[72] Mike Papadakis, Christopher Henard, Mark Harman, Yue Jia, and Yves Le Traon. Threats to the validity of mutation-based test assessment. In *Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016*, ISSTA 2016, pages 354–365, New York, NY, USA, 2016. ACM Press.

[73] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 936–946. IEEE, may 2015.

[74] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Mutation testing advances: An analysis and survey. *Advances in Computers*, 2018.

[75] Ali Parsai and Serge Demeyer. Do null-type mutation operators help prevent null-type faults? In Barbara Catania, Rastislav Královič, Jerzy Nawrocki, and Giovanni Pighizzini, editors, *SOFSEM 2019: Theory and Practice of Computer Science*, pages 419–434, Cham, 2019. Springer International Publishing.

[76] Ali Parsai, Serge Demeyer, and Seph De Busser. C++11/14 mutation operators based on common fault patterns. In Inmaculada Medina-Bulo, Mercedes G. Merayo, and Robert Hierons, editors, *Testing Software and Systems*, pages 102–118, Cham, 2018. Springer International Publishing.

[77] Ali Parsai, Alessandro Murgia, and Serge Demeyer. Evaluating random mutant selection at class-level in projects with non-adequate test suites. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering - EASE 16*, EASE '16, pages 11:1–11:10, New York, NY, USA, 2016. ACM Press.

[78] Ali Parsai, Alessandro Murgia, and Serge Demeyer. LittleDarwin: A feature-rich and extensible mutation testing framework for large and complex java systems. In Mehdi Dastani and Marjan Sirjani, editors, *Fundamentals of Software Engineering: 7th International Conference, FSEN 2017, Tehran, Iran, April 26–28, 2017, Revised Selected Papers*, pages 148–163. Springer International Publishing, 2017.

[79] Ali Parsai, Quinten David Soetens, Alessandro Murgia, and Serge Demeyer. Considering polymorphism in change-based test suite reduction. In Torgeir Dingsøyr, Nils Brede Moe, Roberto Tonelli, Steve Counsell, Cigdem Gencel, and Kai Petersen, editors, *Lecture Notes in Business Information Processing*, pages 166–181. Springer International Publishing, Cham, 2014.

[80] Linda Rising and Norman S. Janoff. The scrum software development process for small teams. *IEEE Software*, 17(4):26–32, 2000.

[81] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, dec 2008.

[82] Mehmet Sahinoglu and Eugene H. Spafford. A bayes sequential statistical procedure for approving software products. In *Proceedings of the IFIP Conference on Approving Software Products (ASP'90)*, pages 43–56, 1990.

[83] Iman Saleh and Khaled Nagi. HadoopMutator: A cloud-based mutation testing framework. In Ina Schaefer and Ioannis Stamelos, editors, *Lecture Notes in Computer Science*, pages 172–187. Springer International Publishing, Cham, 2014.

[84] Hossain Shahriar and Mohammad Zulkernine. Mutation-based testing of format string bugs. In *2008 11th IEEE High Assurance Systems Engineering Symposium*, HASE '08, pages 229–238, Washington, DC, USA, dec 2008. IEEE.

[85] Rodolfo Adamshuk Silva, Simone do Rocio Senger de Souza, and Paulo Sergio Lopes de Souza. Mutation operators for concurrent programs in MPI. In *2012 13th Latin American Test Workshop (LATW)*, LATW '12, pages 1–6, Washington, DC, USA, apr 2012. IEEE.

[86] Ben H. Smith and Laurie Williams. On guiding the augmentation of an automated test suite via mutation analysis. *Empirical Software Engineering*, 14(3):341–369, aug 2008.

[87] Ben H. Smith and Laurie Williams. Should software testers use mutation analysis to augment a test set? *Journal of Systems and Software*, 82(11):1819–1832, nov 2009.

[88] David Tengeri, Arpad Beszedes, David Havas, and Tibor Gyimothy. Toolset and program repository for code coverage-based test suite analysis and manipulation. In *2014 IEEE 14th*

International Working Conference on Source Code Analysis and Manipulation, pages 47–52. IEEE, sep 2014.

[89] David Tengeri, Ferenc Horvath, Arpad Beszedes, Tamas Gergely, and Tibor Gyimothy. Negative effects of bytecode instrumentation on java source code coverage. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 225–235. IEEE, mar 2016.

[90] Jeff Tian. *Software Quality Engineering: Testing, Quality Assurance and Quantifiable Improvement.* John Wiley & Sons, Inc., jan 2009.

[91] Yi Wei, Bertrand Meyer, and Manuel Oriol. Is branch coverage a good measure of testing effectiveness? In Bertrand Meyer and Martin Nordio, editors, *Empirical Software Engineering and Verification*, volume 7007 of *Lecture Notes in Computer Science*, pages 194–212. Springer Berlin Heidelberg, 2012.

[92] Weichen Eric Wong. *On Mutation and Data Flow.* PhD thesis, Purdue University, West Lafayette, IN, USA, 1993. UMI Order No. GAX94-20921.

[93] W.Eric Wong and Aditya P. Mathur. Reducing the cost of mutation testing: An empirical study. *Journal of Systems and Software*, 31(3):185–196, dec 1995.

[94] Q. Yang, J. J. Li, and D. M. Weiss. A survey of coverage-based testing tools. *The Computer Journal*, 52(5):589–597, May 2007.

[95] Fanping Zeng, Liangliang Mao, Zhide Chen, and Qing Cao. Mutation-based testing of integer overflow vulnerabilities. In *2009 5th International Conference on Wireless Communications, Networking and Mobile Computing*, WiCOM'09, pages 4416–4419, Piscataway, NJ, USA, sep 2009. IEEE.

[96] Lingming Zhang, Milos Gligoric, Darko Marinov, and Sarfraz Khurshid. Operator-based and random mutant selection: Better together. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 92–102. IEEE, nov 2013.

[97] Lu Zhang, Shan-Shan Hou, Jun-Jue Hu, Tao Xie, and Hong Mei. Is operator-based mutant selection superior to random mutant selection? In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE 10*, ICSE '10, pages 435–444, New York, NY, USA, 2010. ACM Press.

[98] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, dec 1997.