Callisto - Selecting Effective Mutation Operators for Mutation Testing

Master Thesis

Student
Jan Smits

Graduation Committee
prof. dr. M. Huisman (chair) *(University of Twente)*

dr. A. Fehnker *(University of Twente)* / *(Macquarie University)*

dr. M.H. Everts *(University of Twente)*

N. Jansen *(Info Support)*

UNIVERSITY OF TWENTE. | InfoSupport Solid Innovator

13th January 2022

Abstract

Mutation testing is a powerful testing technique that injects syntactical faults, called mutants, into a program. This is used to measure the effectiveness of the test suite: the more mutants are detected by it, the better the test suite. However, each mutant must be tested by the test suite and there are many mutants possible. Mutation testing is therefore very costly and has not seen wide use in the software industry. This project tries to lower the performance cost of mutation testing by reducing the number of used mutation operators, which govern what types of mutants are generated. The challenge is to select subsets of mutation operators, called mutation levels, such that mutation testing is still effective. For this purpose the tool Callisto is implemented, which analyses mutation operators by determining their performance impact and calculating their quality using a pre-existing quality metric. The focus lies on operators that can generate hard-to-detect mutants, such that the creation of high-quality test cases is encouraged. Mutation levels can then be designed using the produced analysis. To evaluate this strategy it is applied on the mutation operators of the mutation testing framework Stryker Mutator. Nine example programs are mutated, and Callisto is used to analyse the mutation operators involved. The resulting mutation levels show potential for their use as a means to speed up mutation testing, with one mutation level decreasing the performance cost by 49% while retaining 69% effectiveness.

Acknowledgements

First, I would like to thank the people at Info Support for supporting me in this project. Special thanks go out to my daily supervisor Nico Jansen for his feedback, and excellent assistance with StrykerJS, specifically to find the right configuration needed to mutate the example programs. A special mention goes out to Nico timely implementing the 'disable bail' functionality in StrykerJS, which has proved invaluable for this project. Second at Info Support I would like to thank Jan-Jelle Kester for effectively being a second daily supervisor. His valuable feedback and suggestions have greatly helped with refining the thesis. Last but not least I thank Harry Nieboer at Info Support for supporting me weekly in creating and upholding an effective planning to perform this project.

I would also like to thank my university supervisor, Ansgar Fehnker for his helpful feedback and guidance, specifically concerning the scientific perspective of the thesis. A special thanks goes out to always being available to answer questions, even when there is a ten hour time-zone difference.

I am also grateful to Marieke Huisman and Maarten Everts for providing extensive feedback on the draft version of this thesis.

Finally, I want to thank all the friends and family that have supported me in so many ways during this time period.

Table of contents

Glossary

Adequate Test Suite A test suite that achieves a mutation score of 1 (100%).

CI/CD Pipeline A series of steps to automatically build, test and deploy an application.

Coverage Matrix A 2-dimensional binary matrix that shows which mutant was covered by which test case during a mutation testing session. Can be paired with a kill matrix.

Equivalent Mutant A mutant that is semantically equivalent to the original program, and therefore cannot be killed.

Example Program A program that is mutated and used in an experiment to determine the resolution and performance impact of involved mutation operators. Mutants are killed by an accompanying test suite.

Killed Mutant A mutant that was killed, i.e. the test suite failed for the mutant. Also called a dead mutant.

Kill Matrix A 2-dimensional binary matrix that shows which mutant was killed by which test case during a mutation testing session. Can be paired with a coverage matrix.

Minimal Test Suite The smallest possible non-redundant test suite.

Mutant A program that contains one mutation.

Mutant Schemata A technique for improving mutation testing performance where all mutants are compiled in the code at once and activated one by one. Used by Stryker under the alias ‘mutation switching’.

Mutation A syntax token that has been mutated in a significant way by a mutation operator.

Mutation Level A subset of mutation operators whose use will speed up mutation testing, without losing too much resolution.

Mutation Operator Used during mutation testing to mutate a single type of syntax token to one or more mutations.

Mutation Report A report generated after a mutation testing session with the results, such as which mutants were generated, killed, and what mutation score is achieved.

Mutation Score	The percentage of mutants that is killed out of all generated mutants when mutation testing. Formally equivalent mutants are excluded from this, though in practice this is not always done.
Mutation Testing	A software testing technique where faults are injected into source code (mutations) to test the bug-finding capability of the test suite.
Mutation Testing Framework	Software that performs the necessary steps of mutation testing for one or more programming languages.
Mutator	Group of mutation operators which fits a common category.
Non-redundant Test Suite	Test suite where all tests are non-redundant, i.e. all tests uniquely contribute to the mutation score.
Quality Metric	A metric that tries to determine the suitability of a mutation operator for the purpose of mutation testing. This project uses a quality metric to quantify the resolution of mutation operators.
Resolution	A property of mutation operators that indicates its ability to generate subtle, hard-to-kill mutants that encourage the creation of a high-quality test suite.
ROR Mutation Operator	Relational Operator Replacement mutation operator that interchanges `<`, `>`, `<=`, `>=`, `==`, `!=` with each other, or mutates them and their operands to `true` or `false`.
Selective Mutation	A technique to speed up mutation testing by excluding mutation operators to generate fewer mutants. The goal is to find a representative subset of mutation operators.
Static Mutant	Static mutants are a type of mutants that Stryker recognises, which are executed during the loading of a file, instead of during a test case. Therefore Stryker cannot collect coverage information for static mutants. This is a separate property of mutants.
Stryker	An open-source mutation testing framework with three flavours that support C#, Scala and JavaScript.
StrykerJS	The JavaScript flavour of Stryker.
Stryker4s	The Scala flavour of Stryker.
Stryker.NET	The .NET flavour of Stryker.
Survived Mutant	A mutant that has survived, i.e. the test suite passed for the mutant. Also called a live mutant.
Test Case	A singular test that is used to test the functionality of (a piece of) the concerned program. These are used during mutation testing to try and kill mutants. Usually these are unit tests, but integration tests are also usable.
Test Suite	A set of test cases used to test the concerned program.

1. Introduction

1.1 Mutation Testing

Mutation testing is a software testing strategy that works by artificially introducing bugs, or faults, to the source code, creating *mutant* programs. Mutants are created by *mutation operators*, that work on a specific syntax token in the code and alter it to introduce the fault. A mutant program is then tested using the existing test suite. Should the test suite fail, then the mutant was detected, and is now deemed *killed*. If the test suite passes for the mutant program, the mutant has *survived*. A killed mutant indicates that the test suite is adequate enough to detect faults in the source code, such as the mutant. A survived mutant may indicate that the test suite was not adequate enough, and a test case may be missing. The percentage of killed mutants out of all generated mutants is a metric called the *mutation score*. This metric gives an indication of the effectiveness of the test suite used.

Mutation testing therefore does not test the software directly, but rather the tests. By using mutation testing a software developer can gain more confidence in the test suite they have, or expand it with additional test cases to kill more mutants. Mutation testing also serves as a substitute for code coverage, as any untested code will be exposed by the survived mutants in it. Furthermore, investigating a survived mutant may lead to the direct discovery of a bug in the program.

The primary reason why mutation testing is not widely applied in industry is because of its high performance cost [37]. Every mutant requires up to a full run of the test suite, and there are hundreds to thousands of mutants possible, depending on the program size. In the early days of mutation testing this meant that a full run was done overnight. Nowadays there exist several mutation testing frameworks that allow automated mutation testing through use of a CI/CD pipeline, although the process will still take several minutes at least. Therefore a large part of research in mutation testing focusses on speeding it up [37]. One approach, called *selective mutation* reasons that more mutants are generated than needed, and therefore using only a subset of the mutation operators is sufficient [26] [35] [27]. This saves time because by excluding mutation operators fewer mutants are generated and thus fewer executions of the test suite are required. The challenge is then to select a subset of mutation operators representative of all mutation operators, such that mutation testing is still effective, i.e. it can still adequately assess the effectiveness of the test suite.

This project tries to lower the performance cost of mutation testing using a technique very similar to selective mutation. Instead of finding a representative set of mutation operators, mutation

operators are selected based on their *resolution* and *performance impact*. Resolution is a novel concept introduced in this project that describes the ability of mutation operators to generate hard to kill mutants, that subsequently require more specific test cases to kill. In other words, the generated mutants represent a broader range of more subtle bugs that are harder to detect. Thus using mutation operators with a high resolution and killing all their mutants incentivises the user to create a test suite that tests the program most thoroughly and can catch the most bugs. Resolution is explained in more detail in Section 3.1. Performance impact describes how much a mutation operator contributes to the overall performance cost of mutation testing, relative to other mutation operators.

Resolution is highest when all mutation operators are used: the more mutants are generated, the more incentive for users to create test cases to kill those mutants. Conversely, mutation testing is slowest when all mutation operators are used, due to the large amount of mutants. Thus, when selecting a subset of mutation operators it must balance resolution with performance such that performance heavy mutation operators are excluded without losing too much resolution. Such a subset is named a *mutation level*, and designing them is the primary goal of this research. This is named from the perspective of the user of a mutation testing framework, as choosing to only use the mutation operators in a mutation level effectively allows them to choose the *level* of resolution and performance that they want to use when mutation testing.

The difficulty of designing the mutation levels lies in choosing which mutation operators are worth keeping and which can be excluded, based on their resolution and performance impact. To help with this a mutation operator quality metric from existing literature [6], which is an improvement upon an older quality metric [12], is used to quantify the resolution of mutation operators. This quality metric is calculated empirically. Several example programs are mutated and their mutants are killed by their accompanying test suites. The quality is then determined for each killed mutant, based on the number of test cases that kill it. The fewer test cases kill a mutant, the higher its quality will be. The idea behind this is that a mutant that is killed by only a few test cases is deemed hard to kill, and therefore requires more specific test cases to kill it. Thus those kind of mutants will enforce the creation of a higher quality test suite. The quality metric therefore measures the ability of mutants to incentivise the creation of a high-quality test suite. This matches the definition of resolution given above. The quality of a mutation operator is then simply the average quality of the mutants generated by it. The used quality metric is described in more detail in Chapter 3. Based on the quality of the mutation operators a mutation level can be created by establishing a threshold: only mutation operators above the quality threshold are included in the level.

Next to resolution the performance impact of a mutation operator is determined by counting the number of performed test case executions during a mutation testing run for all mutants generated by a mutation operator. As mentioned above, executing the test suite for each mutant takes up the majority of time when mutation testing, so this accurately represents how a mutation operator contributes to the time needed. This value can be used to justify choices regarding the inclusion of mutation operators in a mutation level, besides their resolution. Determining the performance impact of mutation operators is further explained in Section 4.1.

To assess the feasibility and validity of designing and using mutation levels as a technique to speed up mutation testing, it is applied to an existing mutation testing framework: Stryker Mutator [18]. Stryker provides mutation testing for JavaScript, Scala and C#. It takes care of all steps in mutation testing; mutant generation, execution of the test suite on mutants and a report of the results, including the calculation of the mutation score. Stryker is a suitable mutation testing framework, as it provides detailed information in its reports regarding a mutation testing session: this information

is needed to quantify the resolution and performance impact as described above for each mutation operator. Furthermore, Stryker is a prominent and active mutation testing framework focussing on practical software development that has seen constant improvements since its inception and features a large set of mutation operators to create mutation levels from. Finally, the developers of Stryker have expressed that they would like to introduce mutation levels to their framework.

In order to design mutation levels for Stryker the resolution and performance impact of its mutation operators must be determined first, by using the chosen quality metric and counting the test case executions as described above. For this purpose a tool is designed and implemented named Callisto. First a suitable set of example programs is collected and mutated using Stryker. Then Callisto takes the mutation reports generated by Stryker after a mutation testing run and calculates the quality and performance impact of all mutation operators involved. The results for each program are then averaged to obtain the final results. Using multiple programs in the calculation of quality and performance impact limits any bias that a single program can have on the outcome.

With the resolution and performance impact of mutation operators known, several candidate mutation levels can be designed. This can be done using several techniques, including establishing thresholds on resolution and performance, so that only mutation operators above those thresholds are selected [12]. Then the effectiveness and performance of these mutation levels can be determined using an existing technique in an empirical setting [6]. Effectiveness is measured by comparing the size of the minimal test suite needed to kill the mutants generated by the mutation level to that of the test suite needed for all mutation operators. If the sizes are equal[1], then the mutation level incentivises the creation of a test suite with equal testing capabilities as when using mutation testing with all mutation operators. Since the mutation level will generate fewer mutants, some performance is saved without losing any resolution. Similarly, the performance of a mutation level is measured by comparing the number of test case executions needed for the mutation level to using all mutation operators. A mutation level that saves a large percentage of test case executions has a high performance.

Finally, based on these metrics, suitable mutation levels can be identified. Just as with resolution and performance impact, a balance must be found between the effectiveness and performance of a mutation level. Remember that the goal of using mutation levels is to save performance by using fewer mutation operators, while still inciting the creation of a high-quality test suite, i.e. the remaining mutation operators have a high resolution.

1.2 Research Questions

Based on the project description above, the main research question is formulated as follows:
How to partition a set of mutation operators into mutation levels such that these levels balance performance with resolution?
To help answer this, several research sub-questions are defined:

1. How can the resolution of mutation operators be determined?
2. How can the performance impact of mutation operators be determined?
3. How consistent is the resolution of mutation operators over multiple programs?
4. What techniques can be used for partitioning mutation operators into mutation levels?
5. How can the effectiveness and performance of mutation levels be determined?

[1] This is an ideal case, and will generally not occur in reality.

The answers to sub-questions 1 and 2 go deeper into the definition of resolution and performance impact of mutation operators, such that these can be quantified. This is the basis on which the mutation levels are defined. Answering sub-question 3 then builds on this by statistically analysing whether the resolution of mutation operators is consistent when using different example programs. Consistency is preferred for defining the mutation levels. If a mutation operator consistently achieves a high resolution, then it can be confidently included in a mutation level. Conversely, if it consistently scores low then it can be excluded from a mutation level to save performance without losing too much resolution. Without the consistency these choices are harder to make. Next sub-question 4 tries to find techniques that can be directly used for partitioning the mutation operators into mutation levels, using the found resolution and performance impact of mutation operators. An example of this can be setting thresholds on the values of resolution and performance impact. Once an initial design for mutation levels is made, its effectiveness and performance can be determined using the answer to sub-question 5. Following the five sub-questions thus provides an answer to the main research question.

1.3 Contributions

The first contribution of this project is Callisto: the first[2] tool of its kind that automates the analysis of mutation operators using a quality metric. Besides calculating the quality of mutation operators it can also determine their performance impact by either counting the number of mutants generated, or the number of performed test case executions per mutation operator[3]. Finally it can determine the effectiveness and performance of a designed mutation level, or any type of subset of mutation operators, given an example program and its test suite.

The second contribution is the use of mutation levels as a technique to speed up mutation testing. It is evaluated by applying it to the mutation testing framework Stryker Mutator, specifically the JavaScript flavour of Stryker called StrykerJS. Nine example codebases are mutated and the results are used with Callisto to calculate the resolution and performance impact of all the mutation operators of StrykerJS. This is subsequently used as input for the design of several mutation levels for StrykerJS. Finally the effectiveness and performance of these levels is determined with Callisto. The evaluation shows that there is potential to use mutation levels as a means to increase the performance of mutation testing, without losing too much resolution.

Besides these general contributions, this project has also helped Stryker in several aspects. First of all, the performed evaluation resulted in two suitable mutation levels for StrykerJS that can be implemented and made available to its users. They allow for a decrease of the performance cost of 49%, while retaining 69% effectiveness, or a decrease of the performance cost of 71% while retaining 48% effectiveness. These values were experimentally determined using Callisto and the nine example codebases, see Chapter 6.

Second of all, the performed evaluation provides additional insight in the mutation operators of StrykerJS. Callisto counts the number of mutants generated per mutation operator, so that an overview is given of how often an operator is applied. Such an analysis had never been done before for Stryker. Moreover Callisto also counts the occurrence of static mutants (see Section 2.2), such that their relative performance impact can be gauged. This shows the performance that can be gained, should Stryker decide to exclude static mutants in the future.

[2] To the authors knowledge

[3] Depending on what information is available to Callisto.

Third of all, Callisto can be used to design mutation levels for Stryker in the future as well. The experiments performed with Callisto in this project serve as an initial example of its usage. The set of mutation operators that Stryker uses evolves over time as their users request new operators be implemented. Callisto can then serve as an automated tool to evaluate the usefulness of new operators to see if their addition is worthwhile.

Last of all, for the evaluation of Callisto, several example programs were configured for mutation testing with StrykerJS. These adaptations themselves are useful for Stryker, as they allow using these programs to design mutation levels in the future, or for testing StrykerJS in general. In truth, creating these adaptations, and using Callisto, has brought to light several issues present in StrykerJS, which could subsequently be solved [40] [41] [42] [43]. This project was therefore also able to help improve Stryker besides the introduction of mutation levels.

1.4 Outline

The remainder of this thesis is structured as follows. Chapter 2 formally explains mutation testing, describes Stryker in more detail and shows related work. In Chapter 3 the used quality metric is introduced and its workings are explained. Chapter 4 describes the design of Callisto and discusses several implementation details. Chapter 5 starts the evaluation by describing the example programs used, the experiments done with Callisto and their results. Chapter 6 continues by using the results of Callisto to design and evaluate several mutation levels. Finally Chapter 7 discusses and concludes the project and describes several options for future work.

2. Background

This chapter explains mutation testing in detail and formalises the concepts described in the introduction. Additionally the mutation testing framework Stryker, which is used to evaluate the use of mutation levels, is described in more detail. Finally related work is discussed.

2.1 Mutation Testing

Mutation testing is a white-box software testing strategy to discover new test cases and measure the effectiveness of the existing test suite of a given program. The concept of mutation analysis was originally published by DeMillo, Lipton, and Sayward in 1978 [8]. Their paper defined the *coupling effect* hypothesis, stating that simple faults and complex faults are coupled such that should a test detect the simple fault, the complex fault is often detected as well. In other words, complex faults are often caused by simple faults. In addition the *competent programmer* hypothesis was introduced, stating that experienced programmers created most bugs by way of small syntactical mistakes. Mutation testing, mutation analysis applied to software testing, is based on these assumptions.

During mutation testing, a mutation testing framework first injects minor defective changes, or bugs, into the source code, called *mutations*. Each change of the source code results in a new program, which is called a *mutant*. Therefore every mutant contains only one mutation. Formally, let M be the set of valid[1] mutants generated from program p, and T the set of unit tests forming the test suite for p.

Listing 2.1 shows an example program `getMax` in JavaScript that returns the largest item in a given array. A possible mutation for this program is changing the relational operator in line 4 from `>` to `<`. This mutant program would then always return `Number.MIN_SAFE_INTEGER`, which is incorrect.

Next, every produced mutant is run against the existing test suite T of program p. If the outcome of a test case for a mutant differs from the original program[2], the change in the source code was detected and the mutant is marked *killed*. For the example mutant described above, if a test case checks that the max value of `[1, 2, 3]` is 3, this test case will fail, as the mutant always

[1] Mutants can be invalid when they introduce syntax errors.

[2] This means the test case failed, provided the test suite passed for the original program.

```
1  function getMax(arr) {
2          let max = Number.MIN_SAFE_INTEGER
3          for (const item of arr) {
4                  if (item > max){
5                          max = item;
6                  }
7          }
8          return max;
9  }
```

Listing 2.1: Example JavaScript program for finding the maximum value in an array.

returns `Number.MIN_SAFE_INTEGER`. Thus this mutant is killed by this test case. If no test case kills the mutant then the change has gone undetected and the mutant has *survived*, also called a *live* mutant. Survived mutants can then be killed by adding new test cases, often resulting in an improved test suite.

Several formalisations for mutation testing from Estero-Botaro et al. [12] are repeated here. This allows for more precise reasoning about these concepts, and will lay the foundations for explaining the quality metrics in Chapter 3. When a test case t kills a mutant m during mutation testing, this is written as the binary relation 't kills m'. Then the set of mutants from M killed by test cases in T, denoted D for *dead*, is defined as:

$$D = \{ m \in M \mid \exists t \in T \ t \text{ kills } m \} \tag{2.1}$$

Similarly, let P be the set of live mutants from M.

$$P = \{ m \in M \mid \neg\exists t \in T \ t \text{ kills } m \} \tag{2.2}$$

Thus $P = M - D$ and similarly $D = M - P$. It then follows that $M = D \cup P$ and $D \cap P = \varnothing$.

There can exist live mutants that are semantically equivalent to the original program. These are called *equivalent mutants* and are impossible to kill. For example, if in line 4 of `getMax`, `>` is mutated to `>=`, then this mutant program functions exactly the same as the original. Therefore no test case that succeeds for the original program will fail for this mutant, and the mutant is thus equivalent to the original program and unkillable.
Let E be the set of equivalent mutants in M:

$$E = \{ m \in M \mid \neg\exists t \in I \ t \text{ kills } m \} \tag{2.3}$$

Here I is the input space of p, which represents the set of all possible test cases. A test case is then defined as a particular input. I is used instead of T, as a mutant is only equivalent when there exists no possible test case, whether in T or not, that can kill mutant m. It follows that $E \subseteq P \subseteq M$. P can therefore also be seen as the set of *potentially* equivalent mutants, as they have not been killed yet, but also not proven equivalent. Detecting equivalent mutants is difficult, as determining a semantic equivalence between two programs is undecidable, going back to the halting problem [32]. Because equivalent mutants cannot be killed, they cannot help in assessing the effectiveness of a test suite. This makes them undesirable in mutation testing, as they still impact performance.

Based on the above definitions, the mutation score metric $\mathcal{S}$ can now be defined. The mutation score is the fraction of killed mutants out of all mutants, excluding equivalent mutants. Formally:

$$\mathcal{S} = \frac{|D|}{|M| - |E|} \tag{2.4}$$

Since equivalent mutants are impossible to kill, they do not count towards the mutation score. Moreover, when undetected, equivalent mutants will unjustly lower the mutation score, because they will appear as (non-equivalent) survived mutants.

The mutation score gives an indication of the quality of the test suite. If many mutants were killed, it means the test suite is well-equipped to detect alterations of the source code. Mutation testing measures the quality of the test suite and thus tests a program indirectly. In addition investigating a survived mutant can lead to the discovery of previously undiscovered bugs in the program.

Conventionally, code coverage is used as a metric to determine test suite quality. However, achieving 100% code coverage can still lead to only partially tested code. In the extreme case, unit tests can be written that do not contain any assertions. These increase the code coverage, but provide no real testing value, as these tests will not fail in the presence of a bug[3]. On the contrary, mutation testing requires tests to contain assertions, or all mutants will survive. Furthermore, it has been proven that mutation testing subsumes condition coverage techniques such as statement- and decision coverage, in the sense that if the requirements of mutation testing are satisfied, then the requirements of condition coverage are also satisfied [34].

Mutants are generated using *mutation operators*. These focus on a particular syntax token and mutate it to one or several predetermined mutations. For example, an often-used mutation operator is the Relational Operator Replacement (ROR) mutation operator [27] [7] [21] [22]. It interchanges relational operators with one another, or mutates them to boolean literals `True` and `False`. For instance, it can mutate `A < B` to `A <= B`, `A >= B`, `A > B`, `A == B`, `A != B`, `True`, `False`. Two such mutants were shown above with the example program `getMax`. The ROR mutation operator will create seven separate mutants per relational operator it finds in the source code. Most mutation operators only produce one mutant per token, for example mutating `A + B` to `A - B` and vice versa. These kind of mutation operators are rather generic and will work for most programming languages. Mutation operators can also be defined for a syntax token only present in a specific language. Often, such as in the example of `getMax`, mutation operators mutate a token in such a way that it mimics a realistic (accidental) modification that a programmer could make, as is assumed by the competent programmer hypothesis. This way the mutation operator creates realistic bugs in the code, and tests whether the test suite is capable to find such bugs.

The major reason why mutation testing is not widely adopted in the software industry is its relatively high performance cost [37]. This has been a problem since the inception of mutation testing. To determine whether the generated mutants are killed or not requires up to a full run of the test suite per mutant. Depending on the size of the program under test, mutation testing can take minutes to several hours. For example, if 6,000 mutants are generated for a program, and the test suite takes 2 seconds to execute, then mutation testing could take up to $6{,}000 \cdot 2 = 12{,}000$ seconds ≈ 3.3 hours. Therefore a good portion of research is focussed on speeding up mutation testing. This research has been categorised in three areas [33]: (1) *Do fewer* (that is, fewer mutants), (2) *Do smarter*, and (3) *Do faster*. Do fewer consists of bringing down the number of mutants generated, using a variety of techniques, such as selective mutation [26] [35] [27]. Do

[3] Unless the bug causes a crash.

smarter is a broader category that tries to avoid executing parts of a mutation testing session, for example by using incremental mutation, where only newly added code is mutated and the mutation score is calculated based on previous sessions as well [3]. Do faster tries to more efficiently perform mutation testing by speeding up the processing of individual mutants. Examples of this are compiler integration for mutant generation [9], and mutant schemata, where multiple mutants are compiled into one program at once and individually activated one by one using code flags [48]. This ensures the code only needs to be compiled once.

2.2 Stryker Mutator

Stryker Mutator is an open-source mutation testing framework with three prominent variants, called flavours, which each provide mutation testing for a particular programming language. They are: StrykerJS for JavaScript and TypeScript, Stryker.NET for C#, and Stryker4s for Scala [18]. Originally Stryker started out as a graduation project at Info Support B.V. [17] for JavaScript only, implemented in TypeScript. Since then Info Support has sponsored Stryker and it has grown into one of the most well-known and used mutation testing frameworks among software practitioners, with around 40,000 weekly downloads for StrykerJS alone on npm [30]. As opposed to many other frameworks, Stryker was made for practical use in the software industry and not for the purpose of research. This has allowed Stryker to garner a steady user group of software developers providing constant feedback and cooperation on GitHub [19].

Stryker has a centralised system of *mutators*, which are categories of multiple similar mutation operators. For example, the arithmetic mutator can mutate `a + b` to `a - b` and vice versa, `a * b` to `a / b` and vice versa, and `a % b` to `a * b`. Not all mutators are supported by all three flavours, due to language differences. For example the update mutator (`a++` to `a--` and vice versa) is not available in Stryker4s, as Scala does not support that operator. A complete overview of mutators and flavour support is available on the Stryker website [20], and also in Section 4.2. Stryker's mutators are mostly inspired by other prominent mutation testing frameworks, such as PIT for Java [4], and based on developer intuition to create mutants that mimic realistic faults a programmer can make.

Stryker employs several techniques to speed up mutation testing. It uses *mutation switching*, an alias of mutant schemata [48], where all generated mutants are encoded in one program and 'turned on and off' using conditional statements in the code, as was mentioned in Section 2.1. Furthermore StrykerJS stops the execution of a mutant by default as soon as one test case has failed: a feature called *bail*. The idea is that it takes only one failing test case to kill a mutant, therefore there is no point in executing any more tests once this has happened. This saves significant performance, as fewer test cases need to be run. In addition Stryker supports concurrency of test runners. Stryker by default also analyses coverage data from the test runner to optimise the mutant testing phase. Mutants that have no coverage by any test cases are marked as `NoCoverage` and are not tested. When the coverage of each test case is determined, Stryker can use this to only run test cases that have coverage for the mutant under test. A test case that does not cover the mutant under test has no chance of detecting the mutant, and is therefore not executed to save performance.

After Stryker has completed mutation testing a mutation report will be generated, based on which reporters the user has configured to use. The report can show each generated mutant in the code, and whether it was killed or not, per file. The mutation score is calculated over the whole program and also for individual files, making it easier to see where code is poorly tested. Most often this report comes in the form of an interactive HTML file, which can also be uploaded to the

Stryker Dashboard[4], for online access. Stryker can also create a JSON file containing the same information, such that this can be further used or analysed by a subsequent program.

Besides the number of killed and survived mutants Stryker also handles several other types of mutants: timeout, no coverage, ignored, runtime error and compile error mutants. Mutants are classified as timeout mutants when during execution of the test suite the timeout is reached. This can for example occur when the mutant causes an infinite loop. Such mutants are marked as detected. Mutants with no coverage cannot be killed, as there are no test cases that cover them. Ignored mutants are marked by the user, for example when a mutant is wrongly generated, equivalent, or causes other problems. Runtime and compile error mutants cause the error they are named after, and are not taken into account when calculating the mutation score by Stryker. Stryker then has two intermediate metrics, the number of detected and undetected mutants.

$$\#\,detected\ mutants\ =\ \#\,killed\ mutants\ +\ \#\,timeout\ mutants \tag{2.5}$$

$$\#\,undetected\ mutants\ =\ \#\,survived\ mutants\ +\ \#\,no\ coverage\ mutants \tag{2.6}$$

The mutation score is then calculated by Stryker as:

$$mutation\ score = \frac{\#\,detected\ mutants}{\#\,detected\ mutants\ +\ \#\,undetected\ mutants} \cdot 100\%$$

Which can be rewritten using the definitions of Section 2.1 to:

$$mutation\ score = \frac{|D| + \#\,timeout\ mutants}{|M|} \tag{2.7}$$

Of note is that Stryker does not include equivalent mutants in the calculation of the mutation score. Stryker does not employ any techniques for automatically detecting equivalent mutants and leaves that task for the user. In addition timeout mutants also count towards the mutation score, since a test that times out for a mutant is deemed to have detected it, and therefore the mutant can be seen as killed. Thus Stryker's mutation score differs from the one defined in Formula 2.4, in that equivalent mutants will bring down the score and timeout mutants will raise the score, compared to Formula 2.4.

Besides the abovementioned types, Stryker also identifies whether mutants are *static*. This is a separate property of mutants[5]. A mutant is static when it is executed during the loading of a file instead of during a test case. A good example of this is the constructor of a singleton class. Any mutant generated there is only executed once to create the singleton object during startup. Because of this, Stryker cannot collect coverage information for that mutant. As Stryker does not know which test cases cover a static mutant, it resorts to executing the whole test suite. Therefore, static mutants have a large performance impact.

Stryker is used in this project to evaluate the technique of applying mutation levels to speed up mutation testing. Stryker is suitable for this purpose for several reasons. First of all, it produces a detailed mutation report describing exactly which mutants were killed by which test cases. This information is needed to quantify the resolution of mutation operators, as will be explained in Chapter 3. The report also counts the number of performed test case executions per mutant, which is

[4] See `https://dashboard.stryker-mutator.io/`

[5] I.e., there can be killed static mutants, survived static mutants, etc.

used to determine the performance impact of mutation operators, as will be explained in Section 4.1. Moreover, Stryker has published the metadata and source code for their mutation report separately from the flavours, under the name 'mutation testing elements'[6]. This has established it as an open source standard for mutation reports, which allows other mutation testing frameworks to use it as well. As of writing, PIT supports generating Stryker mutation reports. Infection PHP, a prominent mutation testing framework for PHP [38], is in the process of supporting it.

Second of all, Stryker is a prominent and well-maintained mutation testing framework aimed at practical software development, as was mentioned above. This makes it easier to use in practice with a wider range of possible programs, and allows for more precise configuration of mutation testing sessions, such that the required results are guaranteed. Additionally it allows use of the performance optimisations described above.

Third and last of all, Stryker features a large collection of mutation operators across its three flavours. This provides a better opportunity to evaluate the technique proposed in this thesis, as there is a broader range of mutation operators to analyse. Subsequently it also allows for more possibilities to design mutation levels, as this consists of selecting a subset of mutation operators.

2.3 Related Work

This section describes related work that tries to speed up mutation testing by generating fewer mutants using similar techniques as in this project. First fault hierarchies are described as a means of improving the ROR mutation operator. Next five existing mutation operator quality metrics are described and their intentions and applications are discussed. These are *score* [27], *utility* [39], *strength* [16], *effectiveness* [10] [11], and *quality* [12].

Fault Hierarchies

Kaminski et al. [22] have provided several improvements to logic-based testing. Among them is a proposal to improve the ROR mutation operator using mutant subsumption. In short, mutant A subsumes mutant B when every test case that kills A, also kills B. This relationship allows for a shortcut during mutation testing, as subsumed mutants do not need to be executed when their subsuming mutant is already killed. The authors have identified such relations by looking at the detection condition of each mutant generated from the ROR operator. The detection condition is the condition that must be true while executing a test case to detect the ROR mutant and subsequently kill it. For example, when a statement `a < b` is mutated to `False`, then its detection condition is `a < b`. This means that when `a` is less than `b` in a test case, the mutated statement will resolve different than the original, and thus the mutant will be killed. Similarly, a mutant where `a < b` is mutated to `a >= b` will always be detected, as all input to the mutated statement will result in a different outcome than the original. Its detection condition is thus the literal `True`

Now mutant subsumption relations can be found by comparing the detection conditions of the mutants. For example, comparing the two detection conditions above, `a < b` and `True`, it becomes clear that whenever a test case satisfies the first, it will also satisfy the second. Therefore if such a test case kills the mutant corresponding to the first detection condition, it is guaranteed to also kill the mutant corresponding to the second detection condition. Thus the first mutant (`a < b` replaced

[6] See https://github.com/stryker-mutator/mutation-testing-elements

by `False`) subsumes the second mutant (`a < b` replaced by `a >= b`).

This way Kaminski et al. have identified all the subsumption relations between the types of mutants that the ROR mutation operator can generate. They present these in fault hierarchies, which are subsumption graphs as defined by Kurtz et al. [23]. A fault hierarchy contains seven nodes corresponding to the seven possible mutations for one syntax token that the ROR operator mutates. For example, for < these are >=, >, <=, ==, !=, `True`, `False`. Then one-directional edges between the nodes indicate subsumption relations.

Thus seven fault hierarchies are created for each of the relational operator tokens that can be mutated. They each have the same structure, due to the symmetrical nature of these mutations. Three nodes always form the root of a fault hierarchy, which implies that if test cases kill the mutants corresponding to these three nodes, all other mutants in the fault hierarchy will be killed as well, due to the subsumption relations. Therefore the improvement to the ROR mutation operator that Kaminski et al. propose is to only generate the mutants from the three root nodes in the fault hierarchy corresponding to the logical operator being mutated, as killing these mutants is sufficient. This would prevent the generation of four mutants and thus reduce the performance cost of the ROR operator by more than half. This is especially appealing since the ROR mutation operator can be responsible for a large portion of generated mutants, up to 45% [21]. Furthermore the fault hierarchies are constructed based purely on logic, and hold for any programming language using such relational operators.

Lindström and Márki [24] have done further research into the fault hierarchies of the ROR mutation operator. They conclude that the subsumption relations in the fault hierarchies only hold when using weak mutation testing. In weak mutation testing mutants are detected immediately after the execution of the mutated statement. Thus the behavioural difference that a mutant in the fault hierarchy causes is immediately detected and the mutant is marked killed. However, by default strong mutation is used, where mutants are killed at the end of the execution of the program using test cases. In that case the fault caused by the behavioural difference must propagate to the end of the test case where an assertion can detect it. If the propagation stops before then, the mutant is not killed. The authors have found examples where the subsumption relations in the fault hierarchies by Kaminski et al. are broken this way when using strong mutation.

Lindström and Márki have investigated why the subsumption relations in the fault hierarchies were broken and concluded that it is caused by the fact that mutated statements are executed multiple times. If a mutated statement is executed only once by a test case, the fault hierarchies will hold, but if the mutated statement is executed multiple times, the subsumption relations in the fault hierarchies cannot be reliably used any more. In their study done on Java, Lindström and Márki have found that over 50% of mutants are re-executed and therefore cannot be analysed using the fault hierarchies.

In conclusion, the fault hierarchies found by Kaminski et al. show promise for how subsumption relations between mutation operators can be determined using detection conditions and logic. This way mutation operators can be improved to generate fewer mutants and save performance during mutation testing. Using this method on other operators may prove to be difficult however, as not all types of mutants have well-defined detection conditions. Furthermore, Lindström and Márki have shown that such subsumption relations based on detection conditions often do not hold when using strong mutation. Fortunately the cause has been found, and lies in the repeated execution of mutated statements. If a mutation testing framework can monitor the number of times a mutated statement is executed, then fault hierarchies can still be used to generate fewer mutants,

and subsequently lower the cost of mutation testing.

Score

Mresa and Bottaci [27] have defined the *mutation operator score*, a metric for mutation operators. They state that an efficient operator should force the user to write tests that not only kill its own mutants, but those of other operators as well. The mutation operator score measures this effect. It is calculated by determining an adequate test suite exclusively for the mutants of the concerned mutation operator. An adequate test suite achieves a mutation score of 1, or 100%, as is explained in more detail in Section 3.3. Then this operator-specific test suite is used to kill mutants using all mutation operators, and the mutation score achieved by it, as calculated using Formula 2.4, forms the mutation operator score metric.

Mresa and Bottaci calculated the mutation operator score for 21 of the mutation operators of Mothra, an old mutation testing framework for Fortran [7]. They used 11 programs performing various functions, chosen such that no bias exists for a certain operator. On average 3211 mutants were generated per program. Then for each program, operator adequate test suites were constructed. The authors approached this by first using an automatic test data generator tool to generate a set of test cases. If this set was not yet adequate, test cases were added manually. Equivalent mutants were also detected manually. Using the mutation operator score, they were able to select five operators that were the most efficient. All these operators had a score of 85% or higher, and using them a mutation score of just under 99% was achieved on average over all 11 programs.

Utility

Smith et al. [39] have conducted an empirical study to determine how mutation testing contributes to the writing of additional test cases for a test suite. While doing so they define the *utility* of a mutation operator. They assume that when a tester uses mutation testing on their program, they start out with an initial test suite and use this to kill generated mutants. Any mutants killed by this initial test suite are labelled “Dead On Arrival” or *DOA*. Such mutants give an indication that the initial test suite was well-formed, and no new test cases are discovered. Alternatively it may mean that the mutation operator generates easy to kill mutants.

Next the tester selects a live mutant as a target to kill and adds a new test case to the suite. This test case should only kill the targeted mutant. If the augmented test suite kills the targeted mutant, it is marked *killed*. If any other mutants were also killed by the new test case, they are marked *crossfire*. Crossfire mutants are not targeted for killing once marked. This process of targeting a live mutant and adding a test case can repeat until only *stubborn* mutants are left. These mutants are equivalent in some way[7] to the original program, and cannot be killed by adding test cases.

When this process is applied only to the mutants of a mutation operator and all mutants are classified according to the above four types, the utility of that operator can be calculated. This is done using a linear combination of the ratios of mutants in each class, where DOA, killed and crossfire mutants positively affect utility, while stubborn mutants negatively affect the utility as they cannot be killed and have a high detection cost. Two coefficients allow tuning the impact of stubborn and killed mutants compared to DOA and crossfire mutants for calculating utility.

[7] The term *stubborn* deviates from standard terminology. Smith et al. mutate Java source code in their study and *stubborn* also relates to mutant programs that compile to the same bytecode as the original.

A low utility indicates that the operator under test produced few useful mutants for the purpose of adding test cases. A high value means the operator produced many 'killed' mutants that each needed an additional test case, thus enhancing the test suite.

There are however several drawbacks to using utility as a metric. First of all the calculation of the metric depends strongly on the choice of initial test suite, as that determines the percentage of DOA mutants. Second of all, the order in which live mutants are targeted determines the number of killed and crossfire mutants. This also depends on the specific test that is added to kill the targeted mutant. Multiple distinct tests can kill the targeted mutant, but might cause variable amounts of crossfire mutants. Finally the presence of the two coefficients in the calculation adds another complication, as it is not easy to justify what values should be given to them.

Strength

Hu et al. [16] performed an empirical study on killing class-level mutants in Java using the MuJava tool. In their study they define the *Mutation Operator Strength (MOS)*. An operator is deemed strong when many test cases are needed to kill the mutants generated by that operator. MOS is therefore the ratio of the minimal number of test cases needed to kill a set of mutants, to the size of that set of mutants. If all mutants of an operator require a separate test case to kill them, the ratio becomes 1 and the operator is strong. If individual test cases kill many mutants, the ratio becomes small, with a minimum at $1/|M|$, where one test kills all mutants, and no equivalent mutants were generated.

Hu et al. used mutation operator strength to evaluate 28 class-level mutation operators for Java, using 38 classes in the study. This allowed them to identify several operators with a low MOS, whose removal would reduce the number of generated mutants by 82.8%. The few operators with a low MOS generated a majority of all the mutants. This could indicate there is a possible relation between the MOS and the number of generated mutants for an operator, where a large number of generated mutants leads to a low MOS. However, the authors were not convinced of this relation. They state that "it is possible that operators that produce more mutants create more overlap among the mutants." This overlap could lead to test cases killing multiple mutants at once, resulting in a lower MOS.

Effectiveness

Derezińska [10] has assessed the quality of mutation operators for C#. In her work she defines the *effectiveness* of mutation operators as "a ratio of the number of test runs which killed mutants generated by this operator over all test runs performed on these mutants. Only nonequivalent mutants are taken into account."[8] Note that the word effectiveness relates to the test cases killing the mutants, not the mutants themselves. Estero-Botaro et al. [11] have taken this definition and formalised it by dividing the number of test case executions that resulted in a killed mutant, by the total number of test case executions that *could* have killed a mutant (i.e., not considering equivalent mutants). Therefore the effectiveness is equal to the ratio of test runs killing mutants to all test runs, just as Derezińska described. Thus it always falls between 0 and 1, where a high effectiveness indicates that many mutants were killed by many test cases, whereas a low effectiveness means that few mutants were killed by few test cases. When an adequate test suite is used a low effectiveness signifies that the mutants are hard to kill. Effectiveness can be calculated using any set of mutants.

[8] 'Test runs' refers to runs of individual test cases, not the whole test suite.

Therefore, when using only the mutants of one specific mutation operator, it can be used to gauge if the operator typically generates hard-to-kill mutants.

Quality

Estero-Botaro et al. have defined their own quality metric for evaluating the effectiveness of mutation operators, called the *quality of mutation operators* [12]. The purpose of this metric is to determine which mutation operators are worth keeping, and which can be discarded on the basis that they are ineffective. The end goal is to improve the performance of mutation testing by removing mutation operators, without losing effectiveness. Furthermore the metric penalises mutation operators for generating equivalent mutants.

The authors reason that mutants are of higher quality when they are killed by fewer test cases. They therefore classify mutants further after they have been killed. A mutant is *weak* when it is killed by every test case in the test suite. Mutants are *resistant* when they are killed by only one test case. *Hard-to-kill* mutants are a special type of resistant mutant, where the one test case that kills them *only* kills the hard-to-kill mutant. In other words hard-to-kill mutants require their own test case that does not kill any other mutants. The authors therefore deem these mutants the most interesting of all, as they force the creation of their own test cases, adding to the testing value of the test suite. Note that hard-to-kill mutants as defined here should not be confused with the the term hard-to-kill mutants as is used outside this section, such as in Section 3.1 to explain the concept of resolution.

A large part of killed mutants does not fall into one of the above classifications, as there is a whole spectrum between being killed by all test cases (weak) and only one (resistant). Estero-Botaro et al. therefore define mutant quality as a finer metric, depending on the number of test cases killing a mutant, and how many other mutants are killed by those test cases. Quality is calculated per generated mutant in a mutation testing setting. The quality of a mutation operator is determined by averaging the qualities of the mutants generated by that operator.

Since the information needed for calculating this metric, such as which test case kills which mutant, is only available during a mutation testing run, this metric must be determined empirically. A program and its corresponding test suite must be used to perform mutation testing, and only afterwards can the quality of the involved mutation operators be calculated.

Estero-Botaro et al. have applied their quality metric on the WS-BPEL 2.0 language, an XML-based language used to specify the behaviour of a business process based on its interactions with other web services [31]. For this purpose they developed a mutation testing tool for WS-BPEL, called MuBPEL, which includes 26 mutation operators. They mutated four WS-BPEL 2.0 compositions and calculated the quality of each mutation operator for each composition, such that high-scoring ones can be retained and those that generate many invalid or equivalent mutants can be improved or removed. As a result the authors were able to discard six mutation operators. Simultaneously they also calculate Derezińska's effectiveness [10], and the mutation operator strength of Hu et al. [16], as described above. This was done so these two metrics can be compared with mutation operator quality in their ability to identify effective mutation operators. This showed that quality was the only metric of the three that penalises mutation operators for generating equivalent mutants. In the words of Estero-Botaro et al., quality was also deemed "a finer metric than effectiveness and mutation strength in the sense that it can distinguish operators that, from the viewpoint of the other metrics, are similar in goodness."

3. Quality Metric

This chapter provides further background to this project and concerns the mutation operator quality metric that is used to quantify resolution and provides an answer to research sub-question 1. First the concept of resolution is introduced and explained further. Then a quality metric suitable to quantify resolution is selected from the existing five metrics described in the related work, Section 2.3. This is the *quality* of mutation operators of Estero-Botaro et al. [12]. Next this quality metric is further explained, and its characteristics and calculation are described in detail. This forms the basis for another metric, the *coverage quality* of Delgado-Pérez et al., which is a direct improvement to the quality of Estero-Botaro et al. by including coverage information [6]. The improvements and adjustments are discussed in detail. This second quality metric is used in the remainder of the thesis.

3.1 Resolution

The *resolution of mutation operators* is a new concept introduced in this project. It describes the degree to which mutation operators generate subtle, hard-to-kill mutants, such that the creation of high-quality test suites is encouraged. This concept reasons from the perspective of the software tester: they want to create a high-quality test suite for their program. For this they can use mutation testing, which will gauge the effectiveness of their test suite via the achieved mutation score. In addition, they can investigate any survived mutant. If it is not equivalent, they can design a test case to kill the survived mutant, thus expanding the test suite. This can be repeated until no non-equivalent, survived mutants are left, and thus the mutation score is 1, or 100%, according to Formula 2.4. The resulting test suite was therefore (partly[1]) inspired by the generated mutants. Following this process, the generated mutants thus determine what test cases are added to the test suite to kill them. If a mutant represents a bug that is easily found by a test case, then the mutant is easily killed. As a consequence the test case designed to kill such a mutant will not test the program thoroughly, and thus the mutant has a low resolution. Conversely, if a mutant represents a subtle bug that is harder to find, then it will be harder to kill and thus require a more specific test case that tests the program more thoroughly. Such mutants have a high resolution. As an example of this two possible mutants of the function in Listing 3.1 are discussed.

[1] Of course the test suite can contain test cases from before mutation testing was done.

```
1  function isAllowedToBuyAlcohol(customer) {
2          return customer.age >= 18;
3  }
```

Listing 3.1: Example JavaScript program that checks the legal drinking age of a person, to illustrate the difference in resolution of mutants.

The function is given a customer, and checks if they have the legal age to buy alcohol[2]. Both mutants change the relational operator in line 2. The first mutant changes line 2 to `return true;`, such that all customers are allowed to buy alcohol. This mutant is easily killed, as any test case where the age of the customer falls under 18 and it is asserted that the function returns `false`, will kill it. Therefore this mutant has a low resolution. The second mutant changes the relational operator to `return customer.age > 18;`, such that a customer of age 18 is no longer allowed to buy alcohol. To kill this mutant, a test case must specifically assert that when the age of the customer is 18, the function should return true. This is a more specific test case than was needed for the first mutant, and therefore the second mutant has a higher resolution.

So far the resolution of individual mutants has been discussed. The resolution of a mutation operator is simply the average resolution of the mutants generated by it. A mutation operator with a high resolution will therefore have a tendency to generate mutants with a high resolution, i.e. mutants that represent more subtle bugs and thus require more specific test cases to kill. Subsequently, when mutation testing is performed using mutation operators with a high resolution, the tester is incentivised to create more specific test cases to kill the generated mutants. These test cases are therefore capable of finding a broader range of more subtle bugs, and thus form a higher-quality test suite.

In the example above it is quite clear which mutant has the higher resolution of the two. However, when comparing more mutants originating from different mutation operators it becomes unclear how the resolution of each mutant compares. Therefore the first research sub-question tries to solve this problem by finding a method to quantify the resolution of mutants, and subsequently mutation operators. For this purpose a mutation operator quality metric is chosen in the next section.

3.2 Choice of Quality Metric

In Section 2.3 related work was described, which included five mutation operator quality metrics: score, utility, strength, effectiveness and quality. Each of these metrics are closely related to the concept of resolution of a mutation operator, and could therefore be used to quantify it, to provide an answer to research sub-question 1. Out of the five metrics discussed, the mutation operator quality of Estero-Botaro et al. is deemed the best metric for this purpose.

Score does not compete with quality as it deviates in its measured property. Score rewards operators that produce mutants that require test cases that kill many mutants from other operators. This is useful from a performance view, as then the operators with the highest score can be selected, reducing the number of generated mutants while retaining testing value. However this does not

[2] In this case the age of 18.

match the definition of resolution, where mutation operators are valued for their ability to enhance the test suite with new test cases. Therefore score is not considered a suitable metric for quantifying resolution.

As was stated in Section 2.3, using utility has several disadvantages. Its method of determination is flawed, as utility is greatly influenced by the initial test suite, the order of targeting mutants and how new test cases are designed. Furthermore, a choice for the two involved coefficients must also be justified. Therefore utility is also not chosen.

The other three metrics, effectiveness, strength and quality are all related. Estero-Botaro et al. have in fact shown that quality can be calculated from both effectiveness and strength using a common factor [12]. This factor "modulates the value of effectiveness and mutation strength when computing quality from them." All three metrics measure the difficulty to kill the mutants of an operator. The more difficult to kill a mutant, the more specific the test case needs to be to kill it, and therefore the better the resulting test suite will be. This concept matches the definition of resolution.

Compared to effectiveness and strength, quality is the better metric. Quality is the only metric out of the three that penalises mutation operators for generating equivalent mutants. It does this by giving the mutants a quality value of 0, thus bringing down the average quality of the operator. This is seen as an advantage, as equivalent mutants are undesirable in mutation testing, as was explained in Section 2.1. Besides this, quality also provides a more detailed analysis of mutation operators than the other two metrics, as was stated in Section 2.3 using the comparisons Estero-Botaro et al. made. It is therefore recommended by them over effectiveness and strength for these very reasons.

Thus going forward the metric of quality of mutation operators is used. In short, it provides a finer-grained analysis of operators than its two competitors effectiveness and strength, takes equivalent mutants into account and matches the concept of resolution best. In Section 3.4 the coverage-based quality metric of Delgado-Pérez et al. [6] is described, which is a direct improvement to the quality metric discussed here by also taking the coverage of mutants by test cases into account. The 'normal' quality metric of Estero-Botaro et al. is still explained in detail in the next section, as it forms the basis for the calculation of coverage-based quality. In subsequent chapters only the coverage-based quality will be used.

3.3 Calculation

This section describes in detail how the quality of Estero-Botaro et al. is calculated. Before the formula for quality can be explained, several formal concepts are defined first.

Tests and Mutants

In mutation testing, let K_m be the set of test cases killing a mutant m:

$$K_m = \{ t \in T \mid t \text{ kills } m \} \tag{3.1}$$

Similarly, C_t is the set of mutants killed by test case t.

$$C_t = \{ m \in M \mid t \text{ kills } m \} \tag{3.2}$$

Adequacy of Test Suites

A test suite is adequate when it achieves a mutation score of 1, according to Formula 2.4. Simply said it kills all non-equivalent mutants. Formally:

$$T \text{ is adequate} \iff \forall m \in (M - E)\ \exists t \in T\ t \text{ kills } m \tag{3.3}$$

When T is adequate, $P = E$, and $M = D \cup E$.
$D \cap E = \varnothing$, therefore
$D = M - E$ and
$|D| = |M| - |E|$.
Using this in Formula 2.4 shows the mutation score $\mathcal{S}$ is indeed 1.

Redundancy of Test Suites and Minimal Test Suites

The definition of C_t can be extended to test suites:

$$C_T = \bigcup_{t \in T} C_t \tag{3.4}$$

C_T is therefore the set of mutants killed by T. Thus $C_T = D$ [3].

A test suite is non-redundant when removing any test case from the set would cause a decrease in mutation score, i.e. some mutants are no longer killed. In other words all tests in a non-redundant test suite contribute to the mutation score. Test cases are redundant when their removal does not lower the achieved mutation score. Formally:

$$t \in T \text{ is redundant} \iff C_T = C_{T-\{t\}} \tag{3.5}$$

A test suite is then non-redundant if all its test cases are non-redundant. Formally:

$$T \text{ is non-redundant} \iff \forall t \in T\ |C_T| > |C_{T-\{t\}}| \tag{3.6}$$

Usually non-redundant test suites are obtained from an initial redundant test suite by removing redundant test cases. Note that the order of removal determines the resulting test suite. Often multiple test cases are redundant at a point in the removal process and the choice of removal determines the next choice. Therefore multiple distinct non-redundant test suites which can vary in size are possible when starting out with an initial redundant test suite. The size of the non-redundant test suite can create a bias when used in experiments. Therefore the use of a *minimal test suite* is recommended. This is the smallest non-redundant test suite possible with respect to a program and set of mutants when derived from an initial redundant test suite. It is possible that multiple minimal test suites of equal size can be derived. Choosing one minimal test suite out of multiple may create another bias, but this bias is much less significant compared to choosing between non-redundant test suites of different sizes.

[3] When D is determined using T.

Quality

Now the calculation of the quality metric can be explained.
Q_m, the quality of mutant m, is defined as [12]:

$$Q_m = \begin{cases} 0, & m \in E \\ 1 - \dfrac{1}{(|M| - |E|) \cdot |T|} \displaystyle\sum_{t \in K_m} |C_t|, & m \in D \end{cases} \tag{3.7}$$

Again, here K_m is the set of test cases that kill mutant m, and C_t the set of mutants killed by test case t. The sum $\sum_{t \in K_m} |C_t|$ is thus equal to the sum of the number of mutants killed by each test case that kills mutant m. The denominator of $(|M| - |E|) \cdot |T|$ is a constant value, equal to the total number of possible combinations of test cases killing mutants. Equivalent mutants are removed from this total, as they cannot be killed. The sum is therefore always smaller or equal to the denominator, and subsequently, Q_m will always have a value between 0 and 1, or equal to 0. Intuitively, quality is thus determined by comparing the number of test cases that kill mutant m, and the other mutants those test cases kill (the sum), to all possible test cases that could have killed m, and all other mutants that could have been killed by all test cases (the constant denominator). The smaller the sum, the higher the quality will be.

This metric is greatly dependent on the test suite, so the authors assume that the used test suite is adequate. Any mutant that survives is therefore an equivalent mutant, and D encompasses all non-equivalent mutants, such that $M = D \cup E$ and $|M| = |D| + |E|$. In addition test suites are also assumed to be non-redundant when determining mutant quality. As was stated above, ideally a minimal test suite is used, so that the size of the test suite does not create a bias when calculating the quality of mutants.

Some examples of mutant quality are given here to illustrate how the metric functions. First of all, equivalent mutants have a quality of 0, as they do not force the creation of any tests and cannot be killed. Second of all, weak mutants have the next lowest quality, as then $K_m = T$, causing the sum in Formula 3.7 to reach a higher value. In the extreme case that all mutants in M are weak, their quality will be 0, since $\sum_{t \in K_m} |C_t| = (|M| - |E|) \cdot |T|$, which leads to $Q_m = 1 - \frac{1}{1} = 0$. This example again shows that the quality of a mutant always falls between 0 and 1, as this is the worst-case scenario for the quality of a mutant. Next resistant mutants are killed by a single test case ($|K_m| = 1$), so their quality depends on the number of other mutants that test case kills. The lowest quality resistant mutant is killed by a test case that kills all other mutants, so $\sum_{t \in K_m} |C_t| = |D| = |M| - |E|$, and thus its quality is simplified to $1 - (|M| - |E|)/((|M| - |E|) \cdot |T|) = 1 - 1/|T|$. Similarly the quality of a resistant mutant goes up the fewer other mutants its test case kills, until it only kills the resistant mutant, making it a hard-to-kill mutant. Such mutants have the maximum attainable quality, where the sum $\sum_{t \in K_m} |C_t| = 1$, and thus falls away in formula 3.7.

Note that the number of generated mutants and test suite size also influence the quality of mutants. In the extreme case that there is only one non-equivalent mutant and one test case that kills it, its quality will be 0, even though it is technically hard to kill. This also illustrates why it is important to have a minimal test suite, as otherwise useless test cases could be added to increase mutant quality.

To calculate the quality of a mutation operator the average is taken of the qualities of the mutants generated by that operator.

$$Q_O = \frac{1}{|M_O|} \sum_{m \in M_O} Q_m \tag{3.8}$$

Here Q_O is the quality of mutation operator O and M_O the set of mutants generated by the operator. In addition the test suite used to kill mutants of operator O should be adequate and minimal with respect to the set of generated mutants M_O.

Kill Matrix

The information needed to calculate the quality of mutants can almost entirely be retrieved from a data structure called the *kill matrix*. This is an $|T|$ by $|M|$ binary matrix which shows exactly which test case killed which mutant during a mutation testing run. If there is a value `true` in the matrix, then the test case corresponding to that row killed the mutant corresponding to that column. An example (small) kill matrix is given in Table 3.1, where a cross (×) indicates a value `true` and an empty cell `false`.

	m_1	m_2	m_3	m_4
t_1	×	×		×
t_2	×		×	×
t_3				×
t_4		×		×

Table 3.1: Example kill matrix. Instead of `true` and `false` (or 0's and 1's) crosses and empty spaces are used.

Then the values for K_m and C_t can be easily read from the matrix. K_{m1} is equal to all the test cases that have a cross in the column of m_1, so $K_{m1} = \{t_1, t_2\}$. Similarly, C_{t1} is equal to all the mutants that have a cross in the row of t_1, so $C_{t1} = \{m_1, m_2, m_4\}$. Assuming the test suite is adequate, any surviving mutants in the matrix[4] are automatically equivalent, so the set E can be determined. The number of generated mutants and the size of the test suite can be derived from the number of columns and rows respectively. With that, all information is present to calculate Formula 3.7 for each mutant. From the perspective of the kill matrix this is done by counting crosses and dividing that by the size of the kill matrix minus equivalent mutants: $(|M| - |E|) \cdot |T|$. For a mutant m, the crosses are counted by looking at all test cases that have a cross in the column of m (K_m), and summing the number of crosses in the rows of those test cases (C_t). Subsequently, when using Formula 3.8 to calculate the quality of a mutation operator one must also know which mutants in the kill matrix were generated by which mutation operator.

The concept for this kind of data structure is not new, as two other authors have defined it before. Estero-Botaro et al. have defined such a matrix, which they call the execution matrix [12]. The only difference is that it is the transpose of the kill matrix defined above. Thus, mutants are labelled per row, and test cases per column. Ammann et al. independently define the same structure and call it a score function [1]. They label the test cases per row and mutants per column as the kill matrix above. The labelling of Ammann et al. is deemed more intuitive than that of Estero-Botaro et al.: the matrix can also be interpreted as a table, and in a table conventionally rows are added and removed, whilst the number of columns remains constant. The number of mutants generated for

[4] These can be recognised by a column of empty cells.

a program is constant, but the number of involved test cases can change by creating adequate or minimal test suites. Thus the labelling of Ammann et al. is used for the kill matrices in this project.

Method

With the formal definition of the quality of mutation operators given, a workflow is defined by Estero-Botaro et al. to calculate it. Given are a program and its test suite.

1. Perform an initial run of mutation testing on the program using the existing test suite.
2. If necessary, expand the test suite until it is adequate. This will require analysing the results of the initial mutation testing run and may require multiple additional runs. Additionally all equivalent mutants will have to be identified.
3. Extract the kill matrix from the final mutation testing run performed with the adequate test suite.
4. Reduce the test suite to a minimal size, i.e. the minimal non-redundant test suite.
5. Per mutation operator: select only the mutants originating from that operator and reduce the test suite again for these mutants only. This is the mutation operator specific minimal test suite.
6. Calculate the quality of the mutants of the mutation operator *isolated* from any other mutants and average them to obtain the quality of the mutation operator itself.
7. Repeat step 5 and 6 for all desired mutation operators.

Step 2 is the most time-consuming, as expanding a test suite requires insight in the workings of the program and its underlying programming language and testing framework. Furthermore while trying to kill any surviving mutants one must also be watchful for any equivalent mutants, as these are usually detected by hand and cannot be killed.

Step 3 requires the used mutation testing framework to support providing the information contained in the kill matrix to the user, in addition to linking any mutant to its parent mutation operator.

Step 4 tries to derive a minimal test suite from the initial test suite by removing as many test cases without causing any mutant to go from status killed to survived. The procedure for this is explained in Section 4.1.

Step 5 focuses on one mutation operator and isolates its mutants from any others. This ensures that the resulting quality of that mutation operator is calculated independently from any other operators, so that the choice of which mutation operators are used in steps 1 and 2 does not influence it. The test suite needs to be reduced again, as many test cases which were designed to kill mutants that are now not considered can be removed.

Step 6 rounds off the process by calculating the quality of the chosen mutation operator. Note that $|T|$, $|E|$ and $|M|$ in Formula 3.7 are now determined in context of the chosen mutation operator. Thus $|T|$ is equal to the size of the mutation operator specific minimal test suite, $|M|$ only accounts for mutants generated by the operator, and $|E|$ is determined from M only.

In the end a significant amount of experimental setup is needed to calculate the quality metric. This cannot be avoided however, since the metric needs to be calculated in a empirical setting.

3.4 Coverage-based Quality Metric

Delgado-Pérez et al. have proposed an improvement to the quality metric described above [6]. They argue that the metric in its current state overlooks the fact that not all mutants are covered by all test cases. As an example, mutant m is killed by only one test case out of a total of 100, where 10 cover mutant m. Quality as defined by Estero-Botaro et al. would take into account all 100 test cases for calculating the quality of m, and give it a high quality according to Formula 3.7. Delgado-Pérez et al. reason that only the 10 test cases that cover the mutant can detect the defect caused by it, and thus only those test cases should be used in calculating the quality of mutant m. They therefore introduce an improved quality metric called the *coverage-based quality* of a mutant or mutation operator, based on the original quality metric defined above. Note that to calculate this metric, the coverage information of each used test case in the test suite for the generated mutants is needed, in addition to the previously required information for the quality of Estero-Botaro et al. Also of note is that the use of this coverage-based metric does *not* interfere with any other coverage-based techniques a mutation testing framework may use, such as executing only the test cases that can reach a mutant when mutation testing[5]. Before presenting the improved metric, several new definitions are given first.

Coverage of Mutants

First, several concepts connecting mutants and coverage are defined.
Much like 't kills m', let 't reaches m' be the binary relationship that indicates if a test case t reaches (covers) a mutant m. Note that a test case can only kill a mutant when that test case reaches the mutant. Therefore, t kills $m \implies t$ reaches m.
Let M_t be the set of mutants reached by test case t:

$$M_t = \{ m \in M \mid t \text{ reaches } m \} \tag{3.9}$$

Therefore $M_t \subseteq M$.
Let E_t be the set of equivalent mutants reached by test case t:

$$E_t = \{ m \in E \mid t \text{ reaches } m \} \tag{3.10}$$

Therefore $E_t \subseteq E$.
Let N_t be the set of non-equivalent mutants reached by test case t:

$$N_t = M_t \setminus E_t \tag{3.11}$$

Thus $N_t \subseteq M \setminus E$.
Let T_m be the set of test cases that reach mutant m:

$$T_m = \{ t \in T \mid t \text{ reaches } m \} \tag{3.12}$$

Thus $T_m \subseteq T$.

Coverage-based Mutant Categorisation

Delgado-Pérez et al. redefine the categories of mutants introduced by Estero-Botaro et al. taking coverage information into account.

[5] A feature present in Stryker.

A *resistant mutant* was killed by only one test case. A *coverage-based resistant mutant* is reached by all test cases, but killed by only one.
Hard-to-kill mutants were a subset of resistant mutants, where the one test case that kills the mutant *only* kills that mutant. A *coverage-based hard-to-kill mutant* is reached by all test cases and killed by only one test case that reaches all other mutants but kills none of them.
The concept of *weak mutants*, mutants that are killed by all test cases as defined by Estero-Botaro et al., do not have to be redefined, since a killed mutant implies that the mutant is covered by the killing test case. Once again these type of mutants rarely occur and only serve as reference points. The coverage-based quality metric will calculate a value for every mutant based on their kill and coverage information.

An important new type of mutant is the *difficult to reach mutant*. These mutants occur in parts of code that are hard to cover or are barely used. The authors state: "while coverage-based resistant mutants are easy to reach but difficult to kill, we do not know whether difficult to reach mutants are easy or difficult to kill. In other words, we have no certainty about whether or not new test cases reaching a difficult to reach mutant would be able to kill it. Because of the lack of coverage information, we should not use the coverage of these mutants to value their quality." Therefore the authors give a formal definition for difficult to reach mutants, so they can be handled separately when calculating their coverage-based quality. Let DR be the set of difficult to reach mutants:

$$DR = \{\, m \in D \mid \sum_{t \in T_m} |N_t| \leq MCOV \,\} \tag{3.13}$$

Here MCOV is a constant representing the minimum amount of coverage information needed for the coverage to be deemed significant. As an example, the authors gave MCOV the value of 4 in their experiments. Note that only dead mutants are considered, since equivalent mutants are given a quality of 0 either way.

Calculation

With the above definitions, the coverage-based quality metric can now be defined. Let QC_m, the coverage-based quality of mutant m, be calculated as [6]:

$$QC_m = \begin{cases} 0, & m \in E \\ 1 - \dfrac{1}{\sum\limits_{t \in T_m} |N_t|} \sum\limits_{t \in K_m} |C_t|, & m \in D \setminus DR \\ 1 - \dfrac{1}{\sum\limits_{t \in T} |N_t|} \sum\limits_{t \in K_m} |C_t|, & m \in DR \end{cases} \tag{3.14}$$

An extra case is added for mutants that are difficult to reach. Note that the only difference between this case and when mutants are not difficult to reach is that the former uses T and the latter T_m. Difficult to reach mutants are therefore considered as if the whole test suite covers them, because of the nature of such mutants, as explained above.

This formula is very similar to Formula 3.7. The difference lies in the denominator of the fraction. Formula 3.7 multiplied the size of the test suite with the number of non-equivalent mutants. This can be interpreted as the assumption that all test cases have coverage of all dead mutants. Formula 3.14 uses a different denominator equal to the sum of the number of mutants reached by each test case that reaches mutant m[6]. This shows how the coverage-based quality metric differs

[6] For the case where mutants are not difficult to reach.

from the quality metric of Estero-Botaro et al. by including coverage information. Because the denominator in coverage-based quality is smaller or equal to the denominator of 'normal' quality, the former will always have a lower or equal value to the latter. One could therefore say that 'normal' quality overestimated the quality of mutants by not using coverage information.

A more detailed description for the three cases of calculating coverage-based quality are given below.
Just like the 'normal' quality metric, equivalent mutants are given a quality of 0. This works as a penalty for the quality of the mutation operator that generated them.

The vast majority of mutants fall under the second case, which are killed mutants not classified difficult to reach. First of all, when a test case kills a mutant it implies that this test case also covers that mutant. Therefore the sum $\sum_{t \in K_m} |C_t|$ will always be smaller or equal to the sum $\sum_{t \in T_m} |N_t|$. Thus the former sum divided by the latter sum will always fall between 0 and 1 and subsequently the coverage-based quality will also fall between 0 and 1, just like the 'normal' quality metric. Second of all, when a mutant is covered by all test cases, and all test cases cover all mutants, then $T_m = T$ and $\forall t \in T \ |N_t| = |M| - |E|$. Then the denominator in Formula 3.14 simplifies to $\sum_{t \in T} |M| - |E| = (|M| - |E|) \cdot |T|$, which is equal to the denominator of Formula 3.7. Thus, in this case, $QC_m = Q_m$. Lastly, concerning the different mutant categories, weak mutants are given the lowest quality value, for the same reason as was given when using 'normal' quality. Next a coverage-based resistant mutant will be given a relatively high quality, as it is covered by the complete test suite, but only killed by one test case. Thus in Formula 3.14 the first sum will have a larger value, and the second sum a smaller value, making the overall quality value higher. Coverage-based hard-to-kill mutants are given the highest quality, as they are covered by all test cases ($T_m = T$), and are killed by only one test case which kills no other mutants, $\sum_{t \in K_m} |C_t| = 1$.

One may notice that in the case of the coverage-based resistant mutant, where $T_m = T$, the calculation for coverage-based quality is equal to that for difficult to reach mutants ($M \in DR$). This was done on purpose by the authors, as they argue that even though a mutant may be difficult to reach it can still provide a necessary test case and thus benefit the creation of a high-quality test suite.

Analogous to the quality metric by Estero-Botaro et al., the coverage-based quality of a mutation operator is the average of the coverage-based qualities of the mutants it generated.

$$QC_O = \frac{1}{|M_O|} \sum_{m \in M_O} QC_m \tag{3.15}$$

Here O is the mutation operator and M_O the set of mutants generated by it. Delgado-Pérez et al. also advocate for the minimisation of the used test suite when calculating coverage-based quality, as it makes the results more reliable.

Coverage Matrix

Much like the kill matrix explained above, a coverage matrix is defined to help calculate the coverage-based quality metric. This is a binary matrix with the same dimensions as the kill matrix, $|T|$ by $|M|$. A value of `true` in this matrix indicates that the test case corresponding to that row covers (or reaches) the mutant corresponding to that column. An example coverage matrix is given in Table 3.2, which is coupled to the example kill matrix in Table 3.1.

	m_1	m_2	m_3	m_4
t_1	×	×		×
t_2	×	×	×	×
t_3	×			×
t_4	×	×		×

Table 3.2: Example coverage matrix paired with the kill matrix in Table 3.1. Instead of `true` and `false` (or 0's and 1's) crosses and empty spaces are used.

Similar to the kill matrix, the values for T_m and M_t can be easily read from the matrix. T_{m1} is equal to all the test cases that have a cross in the column of m_1, so $T_{m1} = \{t_1, t_2, t_3, t_4\}$. Similarly, M_{t1} is equal to all the mutants that have a cross in the row of t_1, so $M_{t1} = \{m_1, m_2, m_4\}$. Because of the relation between killing a mutant and covering a mutant, any cell in the kill matrix with a cross, means the corresponding cell in the coverage matrix also has a cross.

Method

The experimental method for calculating the coverage-based quality metric is slightly altered from that of the quality metric of Estero-Botaro et al. Step 3 is extended to also include extracting the coverage matrix from the final mutation testing run. Minimising the test suite in step 4 or 5 remains unchanged. Step 6 uses the coverage matrix in addition to the kill matrix to calculate the coverage-based quality metric. Prior to this it must be checked which mutants are difficult to reach according to Formula 3.13, so that the correct case is used per mutant in Formula 3.14. For this the variable *MCOV* must be set, which is a measure for the minimum coverage information needed for a mutant to be deemed significantly reachable. The rest of the process is unaltered.

4. Callisto

In order to determine the resolution and performance impact of mutation operators a tool is developed called Callisto. This chapter describes the design and implementation of Callisto. Since the mutation testing framework Stryker is used to evaluate the use of mutation levels, Callisto is largely designed to be compatible with Stryker and the mutation report it generates.

First the design of Callisto is explained. This largely revolves around the steps required to calculate the coverage-based quality as described in Section 3.4, since this metric is used to quantify resolution. Several other design choices are discussed, such as the minimising of test suites, and which programming language is used for implementation. Second, several implementation details and encountered problems are discussed. This includes minimising test suites in practice, deducing mutation operator names and testing Callisto using Stryker.NET.

4.1 Design

The primary goal of Callisto is to quantify the resolution and the performance impact of mutation operators, such that this information can be used for the design of mutation levels. As the mutation testing framework Stryker is used to evaluate the use of mutation levels, Callisto is designed to be compatible with Stryker and its JSON mutation reports.

In order to quantify the resolution of mutation operators, the coverage-based quality metric by Delgado-Pérez et al. [6] described in Section 3.4 is used. This metric was chosen over the ‘normal’ quality metric by Estero-Botaro et al. [12], as coverage-based quality provides a direct improvement to ‘normal’ quality. The experiments performed by Delgado-Pérez et al. have shown that coverage-based quality is better equipped than ‘normal’ quality to identify the subtle, hard-to-kill mutants that lead to the design of high-quality test cases [6]. Stryker has the capability to report all the needed coverage information of a performed mutation testing run to calculate coverage-based quality. Therefore in the remainder of this thesis the coverage-based quality metric shall be referred to as just ‘quality’, for the sake of simplicity. This should thus not be confused with the quality metric of Estero-Botaro et al.

Quantifying the performance impact of mutation operators is done by counting the number of test case executions performed during mutation testing. Executing test cases takes up the majority of

time needed for mutation testing, as each mutant requires up to a full run of the test suite. Counting the number of test case executions for all mutants of a mutation operator relative to the total number of test case executions is therefore deemed a suitable metric to quantify the performance impact that operator had. One could also use the number of mutants generated by an operator as a metric, however this would only be an approximation of the performance impact. Every mutant requires a different amount of test case executions, since every mutant is covered by a varying amount of test cases. Remember that Stryker only executes the test cases that have coverage over a mutant, to save time. Furthermore no coverage information is available for static mutants, so Stryker resorts to executing the whole test suite for such mutants. Thus the performance impact of mutants may vary greatly[1]. Stryker already counts the number of performed test case executions per mutant and stores them in its report, so the information is readily available.

This metric assumes that each test case in the test suite takes an equal amount of time to execute for each mutant. In reality this will not be the case, as in practice integration tests can be used to try and kill mutants, which can take significantly longer than using unit tests. Nevertheless, this does not impact the use of this metric to measure performance impact, as the number of test case executions will range from hundreds to thousands per mutation operator, and thus outliers will have less of an impact. Moreover, Stryker currently does not measure the time taken for executing test cases. Therefore Callisto cannot use that to quantify the performance impact. Expanding Stryker to provide this is outside the scope of this project, and thus this assumption is used instead. This does provide an opportunity for future work, see Section 7.3.

Workflow

Counting the number of test case executions is trivial compared to calculating the quality of mutation operators. Therefore, the greater part of Callisto focusses on the latter. The necessary steps needed to calculate quality have already been described in Section 3.4, which is based on the steps for calculating the quality of Estero-Botaro et al. as described in Section 3.3. These steps are illustrated in the context of Stryker in the activity diagram in Figure 4.1. Here the tasks are numbered and colour-coded.

A clear relation can be seen between the steps in the workflow in Chapter 3, and the tasks in Figure 4.1. Task 1 to 3 relate to the program and test suite that are used and do not occur in the workflow. Step 1 and 2 of that workflow set up the adequate test suite, which is Task 4 and 5 in Figure 4.1. Step 3, extract the kill and coverage matrices, is Task 6, and Step 4, minimise the main test suite, is Task 7. Finally Step 5 and 6, which minimise the test suite for one mutation operator and subsequently calculate its quality, are represented by Task 8 to 10. Task 11 does not relate to calculating quality, but counts test case executions per mutation operator to determine their performance impact.

Task 1 up to 5 are done manually with help from Stryker and prepare the JSON mutation report that Stryker generates. In Task 6 to 11 Callisto then takes this report as input and calculates the quality and counts the test case executions for each mutation operator involved.

Task 1, 2 and 3 define the context in which quality will be determined. First of all a flavour of Stryker must be chosen, out of StrykerJS, Stryker.NET and Stryker4s. This determines the programming language used. For Task 2 a set of mutation operators from the chosen Stryker flavour

[1] A mutant covered by only one test case will be processed approximately ten times faster than a mutant covered by ten test cases.

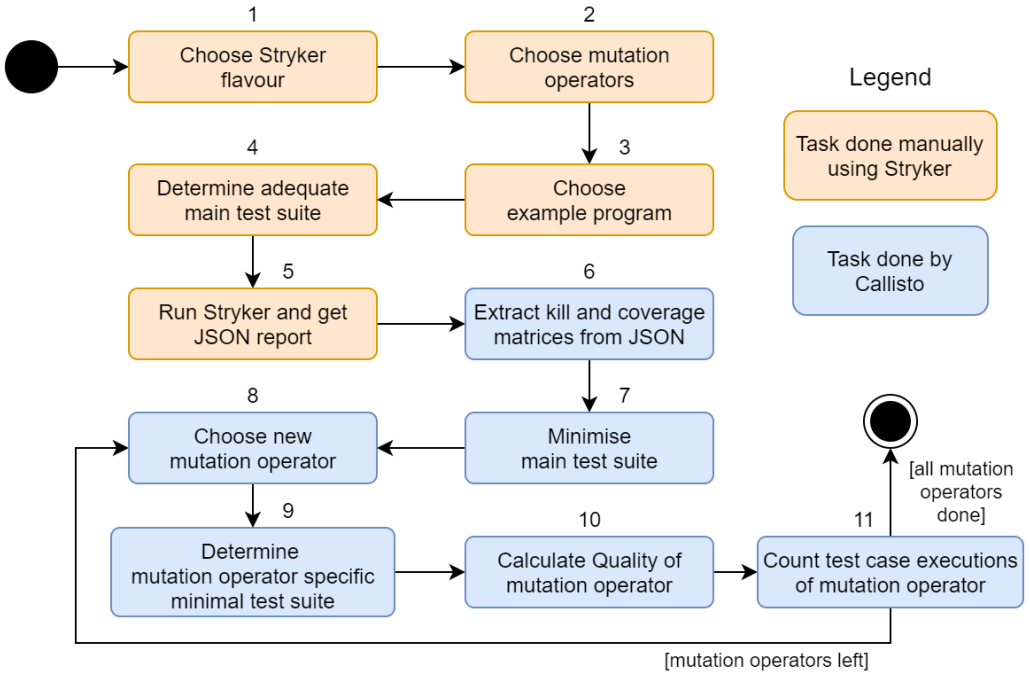

Figure 4.1: Activity Diagram for calculating the quality and performance impact of mutation operators of Stryker using Callisto. The orange tasks are done manually, the blue tasks are performed by Callisto.

must be selected for which quality will be determined. Task 3 is to choose the program for which mutants are generated. This program must be compatible with the chosen Stryker flavour[2].

Task 4 consists of expanding the existing test suite of the chosen example program until it is adequate, i.e. a mutation score of 100% is achieved for the mutants generated by the chosen mutation operators. In general this task will take the most time and effort, as a good understanding of the example program is required to design new test cases. This is an iterative process: survived mutants are investigated, new test cases are designed to kill them and Stryker is used to see if the new test cases are effective in raising the mutation score. This is repeated until the test suite is adequate. This process also involves identifying all equivalent mutants generated from the example program.

The purpose of Task 5 is to use Stryker to generate the mutation report in JSON format, which forms the singular input for Callisto. Callisto will require specific information to calculate quality, such as coverage information. Therefore Stryker must be correctly configured in this step to generate a suitable mutation report. The required configuration is explained in Chapter 5.

In Task 6 Callisto deserialises the given JSON report into a suitable data structure to hold all the raw information contained within. As is explained in Chapter 3, the kill- and coverage matrix hold most of the required information to calculate quality. Therefore the next step in the process is to extract the kill- and coverage matrix from the deserialised structure.

Task 7 consists of determining the main minimal test suite from the adequate test suite designed in Task 4. This is done solely from the information contained in the kill matrix. In short, rows corresponding to redundant test cases are removed from the matrix, while every killed mutant

[2]The program must match the chosen programming language and be able to run Stryker on it.

remains killed[3]. The process to accomplish this goal is explained in detail in the next section, 4.2.

Task 8 to 10 calculate the quality for each involved mutation operator. In Task 8 a mutation operator is selected which has not been processed yet. In Task 9 the mutation operator specific minimal test suite is determined. This involves extracting a smaller kill- and coverage matrix from the one created in Task 6 by selecting only the mutants generated by the chosen mutation operator. Then the main minimal test suite of Task 7 is minimised again to correct for the reduced number of mutants. Now these matrices are used in Task 10 to calculate the quality of the mutation operator according to Formulae 3.14 and 3.15, as is explained in Chapter 3.

Alongside quality, the deviation in the quality of individual mutants is also calculated per mutation operator. This allows insight into whether the quality of individual mutants is dispersed or not, which helps judge the reliability of the mutation operator quality. This plays a role in answering research sub-question 3. The deviation used is the Mean Absolute Deviation (MAD). This was chosen over the standard deviation, as it better reflects reality: MAD is the average absolute distance from the mean of a set.

Finally in Task 11 the test case executions are counted for each mutant generated by the selected operator and summed. This is done using the JSON file directly, since Stryker saves the count per mutant in its report, as was mentioned above. The resulting quality values and test case execution totals of all involved mutation operators are reported back to the user.

Workflow Remarks

In Task 2, generally all available mutation operators are chosen, however should the quality of only one mutation operator be of interest (for example a mutation operator new to Stryker), then this can be selected at that point in time to save work for subsequent tasks.

For Task 3, the example program that is used should have a high initial mutation score, as this will save work during Task 4. Section 5.2 will go into more detail concerning the criteria for suitable example programs.

Stryker does not have any formal label for equivalent mutants, so they will remain as ‘survived’ mutants when expanding the test suite in Task 4. Stryker will therefore not report a mutation score of 100%, but that is because it calculates mutation score using Formula 2.7, where equivalent mutants are not taken into account, as opposed to Formula 2.4 upon which the definition of adequate test suite is based. As there should be no actual survived mutants after Task 4, Callisto assumes that any mutants marked as survived by Stryker are in fact equivalent, and therefore gives them a quality of 0. In the kill matrix they can be identified by a column of empty cells.

As was explained in Section 2.2, no coverage information is available for static mutants. When calculating their quality, Callisto therefore assumes they are covered only by the test cases that kill them, formally $T_m = K_m$. This is based on the fact that when a mutant is killed by a test case, it implies it is also covered by that test case. Because of this, static mutants have a lower quality on average than others, according to the calculation of quality, Formula 3.14.

For the experiments performed with Stryker in the remainder of this thesis, Task 4 is skipped. This was done because expanding the test suite to adequacy was infeasible for the scope of this

[3]Equivalent mutants are ignored during this process.

project. Task 4 is not essential for the rest of the tasks to function. When only the mutants killed by the initial test suite are used, and all others are ignored, then the initial test suite is adequate for the remaining subset of mutants. Stryker cannot leave out this subset, so in the JSON report of Task 5 any non-killed mutants will simply have another status. Callisto therefore needs to be configured to only use killed mutants. Section 5.3 goes into more detail on the decision to skip Task 4.

Filtering Mutants

During Task 9 all relevant mutants of a mutation operator are selected to calculate quality for. At this point the user is given the option to filter out mutants in two ways. First of all, when skipping Task 4, only mutants killed by the initial test suite should be used to calculate quality. By default Callisto assumes that adequate test suites are used and any survived mutant is equivalent. The user can change this so that Callisto only selects the killed mutants of a mutation operator. This way any non-killed[4] mutants are completely ignored by Callisto and do not show up in any results.

Second of all, the user is given the option to include static mutants. As was explained in Section 2.2, every static mutant requires a full run of the test suite to determine if it is killed. As mentioned above, this causes static mutants to achieve a lower quality on average, which influences the quality of mutation operators. Therefore, Callisto ignores static mutants by default. One can configure Callisto to include static mutants in the analysis. This allows investigating the impact of static mutants during mutation testing, by comparing results with and without static mutants.

These two filtering options also influence the counting of test case executions to determine the performance impact of a mutation operator: when static mutants are ignored, then only the test case executions of non-static mutants are counted, and similarly for using killed mutants only. It should be stated that these two options can be used simultaneously.

Minimising a Test Suite

As is explained in Chapter 3, minimising the adequate test suite of an example program increases the reliability of the resulting quality calculation. Formula 3.14 shows that the size of the test suite influences the found quality: the sum $\sum_{t \in K_m} |C_t|$ has the tendency to attain a higher value with a larger test suite, as K_m can contain more test cases. Therefore minimising the test suite removes this bias caused by the size.

Minimising the test suite can be done entirely from the information contained in the kill matrix. In that context, minimising consists of removing a maximum amount of rows from the matrix, such that every column retains at least one cell with a `true` value, excluding columns corresponding to equivalent mutants. This minimisation problem can be formulated as a *Binary Integer Linear Programming* (BILP) problem, as is shown by Palomo-Lozano et al. [36]. In an ILP problem, the problem is described using a linear function of integers, the value of which must be minimised (or maximised) under a set of constraints, which are also described as a linear function of integers. A Binary ILP is simply an ILP where the integer values are restricted to the values 0 and 1. Palomo-Lozano et al. show that using binary decision variables $x_1, ..., x_n$, where $n = |T|$ can be used to represent which test cases are kept when minimising. A value of $x_i = 0$ would then indicate that test case t_i is removed. The objective is then to minimise the sum $\sum_{i=1}^{n} x_i$,

[4] These are mostly survived mutants, but in the case of Stryker other types of mutants can occur, such as timeout mutants.

such that the maximum amount of test cases is removed. This is done under the constraints that every killed mutant remains killed after minimising is done. These constraints are formulated by Palomo-Lozano et al. for one mutant $m_j \in D$ as the sum $\sum_{i=1}^{n} k_{ij}x_i \geq 1$, where k_{ij} corresponds to the binary integer value in the kill matrix at row i and column j. Mutant m_j initially has status killed, as a survived mutant is deemed equivalent and thus ignored when minimising. Multiplying[5] k_{ij} with x_i ensures that only the test cases which are preserved in minimising can contribute to the killing of mutant m_i. Now the full BILP can be formulated as:

$$\textbf{Minimise} \sum_{i=1}^{n} x_i \textbf{ subject to } \forall i \in [1, |M|] \sum_{j=1}^{n} k_{ij}x_j \geq 1 \tag{4.1}$$

where kill matrix k only contains killed mutants, i.e. all survived mutants are removed, and has size $n \times |M|$. With this formalisation of the test suite minimisation problem, suitable (B)ILP solver software can be used to find a solution. The solution will then consist of values 0 or 1 for the binary decision variables $x_i, \ldots, x_n$. Minimising is then the simple matter of removing all test cases with a binary decision variable of value 0.

Design Choices

Here several design choices regarding Callisto are explained. First of all, Callisto is designed as a Command-Line Interface (CLI) tool. This was done as Callisto's input and output are clearly defined, making a CLI tool a suitable form factor. Furthermore, Stryker itself is also a CLI tool, which makes running Callisto alongside Stryker easier in, for example, a pipeline. Moreover, any configuration for Callisto can be easily passed by use of command-line options this way. See Appendix A for a complete overview of the command-line usage of Callisto.

Second of all, Callisto is implemented in C#. This language was chosen for several reasons. Firstly C# and the .NET environment allow access to libraries needed for calculating quality. Most notably, the ability to solve (B)ILP problems to minimise test suites is a mandatory requirement for Callisto. This can be performed using libraries available in the .NET environment. Secondly C# is a suitable language to develop CLI tools. Lastly it allows the use of Stryker (Stryker.NET) for mutation testing during development.

Third of all, Callisto's output takes the form of a simple CSV (Comma-Separated Values) file. Since the result of running Callisto consists of several statistics paired with a mutation operator (quality, performance impact, mutant count, etc.), this output can be most easily written as comma-separated values. This format allows them to be easily interpreted by other programs.

Last of all, as was explained at the start of this section, Callisto has been designed to be compatible with Stryker and the JSON mutation reports it provides. However, this does not mean that Callisto is solely compatible with Stryker. As was explained in Section 2.2, Stryker has presented their mutation report structure as an open-source standard to be used by other mutation testing frameworks as well. This automatically makes Callisto compatible with those frameworks as well, provided they store all the needed information for Callisto in their reports. Moreover, Callisto can always be extended to accept other forms of mutation reports. The activity diagram in Figure 4.1 can easily be adapted for other frameworks and languages. Callisto can then still use the same internal data structures such as the kill and coverage matrices, and the same algorithm to minimise test suites.

[5] Multiplying two binary values can also be interpreted as a logical AND.

4.2 Implementation

This section goes into detail concerning the implementation of Callisto and discusses any major problems that were encountered and their solutions.

Minimising Test Suites in Practice

In order for Callisto to be able to minimise test suites using the formalised problem described in Section 4.1, a framework and solver for (B)ILP problems compatible with .NET had to be found. In addition the solver should be open source, so that it can be distributed with Callisto for free. Many well-known solvers such as CPLEX and Gurobi either require a paid license or are only usable from C++. In the end the framework that is used in Callisto is Google OR-tools [15], an open-source software suite for solving several types of problems such as vehicle routing, flow graphs, integer and linear programming, and constraint programming. OR-tools provides an API to programmatically model the problem to solve, after which one of several open-source solvers can be chosen, either third-party or developed by Google.

Encoding and solving a linear programming problem in Google OR-tools follows these steps:

1. Instantiate the main `Solver` object, with the name of the solver that will be used.
2. Define the variables of the problem.
3. Define the constraints using the variables.
4. Define the objective of the problem using the variables, stating whether the goal is to minimise or maximise.
5. Call upon the `Solver` object to find an optimal solution for the problem.
6. Retrieve the found solution from the `Solver` object.

Performing these steps for the problem of minimising a test suite is relatively straightforward using the formal definition of the problem shown in Section 4.1. For the variables an array of boolean variables is created which will represent $x_1, ..., x_n$, the decision variables for the test cases. Next they are used to encode the constraints that each killed mutant must remain killed after minimising, for which the information in the kill matrix is used. The sum of the decision variables forms the objective, which must be minimised. Now the `Solve()` method can be called from the `Solver` object to try and find an optimal solution. Once this process is completed the status of the result is interpreted to see if an optimal solution was indeed found. If this is the case, then the values for the decision variables defined earlier are retrieved to interpret which test cases are kept and removed. This is then used to remove rows from the kill- and coverage matrix corresponding to test cases which are removed. Although minimising is done solely using the kill matrix, the coverage matrix must always match its corresponding kill matrix, thus the same rows are also removed from the coverage matrix.

For step 1 a solver must be chosen. Google provides its own linear programming solver, called GLOP (Google Linear OPtimisation). Besides this, third-party solvers can be chosen from the open-source world, such as GLPK, or from proprietary projects, such as CPLEX or Gurobi, provided a license has been acquired. For Callisto the main qualities needed from a solver are speed and reliability. Therefore a benchmark was set up to test the available solvers. This benchmark consisted of minimising a test suite of 50 test cases, subject to killing 1000 mutants. The corresponding kill matrix of 50 rows and 1000 columns was filled with random data, i.e. every cell has a 50% chance to contain `true` and a 50% chance to contain `false`. Such a kill matrix is not representative of a

real scenario for Callisto, but for the purposes of a benchmark it was suitable, and easy to set up. Next the time in milliseconds needed to solve this problem and the status of the found solution was recorded for each solver. The results were as follows:

1. GLOP: 16 ms
2. CLP: 27 ms
3. CP-SAT: 4927 ms
4. SAT: 4500-5500 ms
5. SCIP: 20704 ms
6. CBC: 87880 ms
7. BOP: 20+ minutes

The BOP solver was the only solver which did not complete the benchmark in a reasonable time, and its execution was halted after 20 minutes. The other solvers were all able to achieve optimal solutions. As can be seen, GLOP and CLP outperform the other solvers by several orders of magnitude. As it performed the best, GLOP was chosen as the solver for minimising test suites.

However, when testing the functionality of GLOP further, an error in the solutions was found. For kill matrices with a certain configuration GLOP would give solutions with non-integer values for the variables, despite the fact that the variables were configured to only take integer values. Such a kill matrix is shown in Table 4.1.

	m_1	m_2	m_3
t_1	×	×	
t_2		×	×
t_3	×		×

Table 4.1: Kill matrix with three separate optimal solutions for minimising. GLOP cannot solve this matrix.

Minimising the test suite according to this kill matrix results in three separate optimal solutions. Removing one of the three test cases automatically makes the remaining two a minimal test suite for mutants m_1, m_2, m_3. One would therefore expect to see a solution from GLOP where 2 of the three decision variables are given value 1, and the other value 0. Instead GLOP gives all three a value of 0.5, which cannot be interpreted as a solution. This could be interpreted as GLOP not being able to choose one of the three solutions and instead 'spreading' the solution out over all three test cases. However this choice of value is not by chance, as it is technically a better solution according to the objective and constraints of the problem. The objective now resolves to a minimal value of 1.5 (sum of all decision variables), which is lower than the expected value of 2. Meanwhile the constraint for each column is satisfied, as they become $0.5 \cdot 1 + 0.5 \cdot 1 \geq 1$. GLOP therefore does not hold itself to the implied constraint that all decision variables should only take values 0 or 1. This is caused by the fact that GLOP is a pure linear programming solver, which only solves the relaxation of a Mixed Integer Programming (MIP) problem. Therefore an alternative solver is required for this minimising case, for which SAT, CBC or BOP are suited as they do not have this problem. Out of these three SAT is chosen as it performed the best in the above benchmark. In conclusion, Callisto uses GLOP by default and warns the user to use SAT when GLOP gives back non-integer solution values. A command-line option for Callisto allows specifying which solver to

use.

Deducing Mutation Operator Names

As mentioned in Section 2.2, Stryker's mutation operators are divided over a set of mutators, which serve as categories of mutation operators. In the mutation report JSON file that is given to Callisto the data for each mutant only shows the name of the mutator category it belongs under, and not the mutation operator name. In this project resolution and performance impact are determined per mutation operator and not per mutator. This is done to allow a more fine-grained analysis of the different kinds of mutants that Stryker generates. Therefore, Callisto must deduce the name of the mutation operator used per mutant. Fortunately the JSON file holds all the necessary data for this: the original source code of the program, the location of the mutation in the source code, and the replacement code which forms the mutation. Stryker uses all this information in the HTML mutation report to show a user what part of their code is mutated and in which way. Using the location and the source code the original piece of unmutated code is determined. This can then be compared with the replacement code to deduce the mutation operator that was used. For example, if the original code was `a + b` and the replacement is `a - b`, then the 'arithmetic operator replacement' mutation operator was used which replaces an addition operator with a subtraction operator. In the context of Stryker and Callisto, a distinction is made between the individual mutations that can take place and labels them as separate mutation operators. Therefore the mutation `a - b` to `a + b` is generated by a different mutation operator than the mutant described before.

Stryker does not currently have an official system of names for each of the mutations performed. Therefore Callisto uses a placeholder system. In general the name of a mutation operator uses the name of the containing mutator category with a postfix that describes the mutant. For mutation operators where a syntax token is replaced, the postfix contains the original code and the mutated code separated by the word 'To'. For example, the mutation `a - b` to `a + b` is performed by the ArithmeticOperator-To+. The use of symbols in the name was chosen as it shortens the name significantly[6] and quickly conveys what mutation is performed. For mutations where a part of code is 'emptied', 'filled' or removed, the postfix contains the word 'Empty', 'Fill' or 'Removal' respectively. For example, mutating any string to an empty string `""` is done by the StringLiteralEmpty mutation operator. When a syntax token is removed that token is also mentioned in the postfix. Some of the mutators of Stryker only contain a single mutation, for example the BlockStatement mutator in StrykerJS, which empties any block statement by mutating it to `{}`. As the mutator only performs one mutation, the underlying mutation operator is simply given the same name as the mutator, without a postfix.

Following these rules, a full list of every mutation operator in StrykerJS at the time of writing is provided here:

- ArithmeticOperator%To*
- ArithmeticOperator*To/
- ArithmeticOperator/To*
- ArithmeticOperator+To-
- ArithmeticOperator-To+
- ArrayDeclarationEmpty
- ArrayDeclarationEmptyConstructor
- ArrayDeclarationFill
- ArrowFunction
- BlockStatement
- BooleanLiteralfalseTotrue
- BooleanLiteralRemoveNegation
- BooleanLiteraltrueTofalse
- ConditionalExpression!==Tofalse
- ConditionalExpression!==Totrue
- ConditionalExpression!=Tofalse

[6] A much longer name could be ArithmeticOperatorSubtractionToAddition.

- ConditionalExpression!=Totrue
- ConditionalExpression<=Tofalse
- ConditionalExpression<=Totrue
- ConditionalExpression<Tofalse
- ConditionalExpression<Totrue
- ConditionalExpression===Tofalse
- ConditionalExpression===Totrue
- ConditionalExpression==Tofalse
- ConditionalExpression==Totrue
- ConditionalExpression>=Tofalse
- ConditionalExpression>=Totrue
- ConditionalExpression>Tofalse
- ConditionalExpression>Totrue
- ConditionalExpressionConditionTofalse
- ConditionalExpressionConditionTotrue
- ConditionalExpressionEmptyCase
- EqualityOperator!==To===
- EqualityOperator!=To==
- EqualityOperator<=To<
- EqualityOperator<=To>
- EqualityOperator<To<=
- EqualityOperator<To>=
- EqualityOperator===To!==
- EqualityOperator==To!=
- EqualityOperator>=To<
- EqualityOperator>=To>
- EqualityOperator>To<=
- EqualityOperator>To>=
- LogicalOperator&&To||
- LogicalOperator??To&&
- LogicalOperator||To&&
- ObjectLiteral
- OptionalChaining
- Regex
- StringLiteralEmpty
- StringLiteralFill
- UnaryOperator+To-
- UnaryOperatorRemove~
- UnaryOperator-To+
- UpdateOperatorPost++To--
- UpdateOperatorPost--To++
- UpdateOperatorPre++To--
- UpdateOperatorPre--To++

There are a few notable exceptions in this list when it comes to names. ConditionalExpressionConditionTo- true and false is a more generic mutation operator. Any mutant that falls under the ConditionalExpression mutator but not in any of the other mutation operators under this mutator are named this way. These are mutants where a condition in for example an if-statement or for-loop is given using method calls or a boolean variable, and not using a relational operator. An example of such a mutation is '`while (array.isEmpty()) {}`' to '`while (false) {}`'. Similarly the ConditionalExpressionEmptyCase mutation operator groups all mutants where a `case:` or `default:` is emptied inside a switch-case statement. Furthermore the Regex (regular expression) mutator is not split up into mutation operators here, as it contains 22 mutation operators and this would clutter the list. Regular expressions are written in their own language, and for that reason they are deemed largely outside of the scope of this project. Their quality and performance impact are still determined in this project, but for all regex mutants as a whole. For a finer-grained analysis of regex mutations they should be analysed as a set of mutation operators for the regex language in a future project.

The developers of Stryker intend to implement their own system of mutation operator names in the near future, which is then also included in the JSON report. Once this is implemented Callisto can simply use that information to find mutation operator names, and the deduction technique above becomes obsolete.

Testing

In order to ensure Callisto functions correctly, a test suite of unit tests was created using the MSTest framework. In total 35 test cases were created which mostly test the four major parts of Callisto. These are deserialising and interpreting the input JSON file, manipulating the kill- and coverage

matrix, minimising test suites, and calculating the quality of mutation operators. Ensuring these four parts function correctly is essential for Callisto to determine the resolution and performance impact of mutation operators. In most test cases example data is used to test functionality, such as a test JSON report or example kill matrix.

As Callisto is written in C#, Stryker.NET can be used to determine the quality of the test suite. Running Stryker.NET version 1.01 on Callisto generates a total of 1099 mutants. The achieved mutation score is 45.13%. This is a low score, however this is mostly caused by the fact that 459 mutants have no coverage by the test suite. The test suite does not test all code of Callisto. Most notably, configuration like that of the CLI interface such that the right command-line options are available to the user are not tested, but do contain mutants. When only the covered mutants are taken into account, 481 out of 609 mutants are killed, 78.98%. In important parts of Callisto, such as the actual calculation of quality or minimising a test suite nearly all mutants are killed. This is deemed the most important aspect of testing Callisto, and therefore the created test suite is considered to be sufficient. An improvement can be made by including end-to-end tests for Callisto, as currently the high-level control code which outlines the steps for calculating quality is untested.

5. Evaluation

This chapter starts to evaluate the technique of designing and using mutation levels by applying it on Stryker. The main purpose of the evaluation is to assess the viability and validity of designing and using mutation levels to speed up mutation testing. This is done by first mutating a set of example programs using a flavour of Stryker. Then Callisto can use the resulting mutation reports to determine the resolution and performance impact of the involved mutation operators. This is used to design several trial mutation levels. Doing this will determine how difficult this process is to follow, and if it can be easily applied to other frameworks and languages. By analysing the resulting levels it can be assessed if they indeed speed up mutation testing, without losing too much resolution. This evaluation also provides an opportunity to see how Callisto performs in a realistic use case.

This chapter starts by choosing a flavour of Stryker to use for the evaluation. Then the criteria for suitable example programs are explained, after which nine example programs are found. Next, the use of inadequate test suites is discussed. To finish, the methodology used to analyse the mutation operators of StrykerJS with Callisto is described in detail, after which it is applied on the nine example programs and its results are discussed.

The evaluation is continued in Chapter 6, where the results of this chapter are used to design and assess mutation levels for StrykerJS.

5.1 Choice of Stryker Flavour

Figure 4.1 shows that the first step in using Callisto with Stryker is choosing a flavour of Stryker. For the experiments performed in this project, StrykerJS was chosen. This choice was primarily made because, as of writing, StrykerJS is the only flavour able to provide test case information. This information, such as which test case covers and/or kills which mutant, is essential for calculating quality. Stryker.NET and Stryker4s do use test coverage information to save time during mutation testing, but the information is not saved in the resulting JSON report yet. For the purpose of calculating quality, StrykerJS is therefore the only choice left. Consequently, only the resolution and performance impact of mutation operators applicable to JavaScript can be determined. Thus all further experiments performed with Callisto are done using StrykerJS with example programs written in JavaScript and/or TypeScript. The mutation levels which are designed in Chapter 6 are

therefore for StrykerJS only.

5.2 Example Programs

The example programs form the basis of the data that Callisto uses. Using the mutants generated from these programs the resolution and performance impact of mutation operators will be determined.

Criteria

There are several criteria that an example program should meet for use with Callisto and StrykerJS. A suitable example program:

1. is written in JavaScript and/or TypeScript,
2. is compatible with StrykerJS and its required configuration,
3. has at least 200 killed mutants.

The first criterion is quite evident: as StrykerJS is used, example programs should be written in JavaScript and/or TypeScript. The second criterion builds on this: to be compatible with StrykerJS several requirements must be met. First the example program should use a compatible framework. Several popular JavaScript frameworks are supported, such as React, Angular, VueJS and NodeJS. Next the accompanying test suite should also be compatible with StrykerJS. In practice this means the test runner used should be supported by StrykerJS. Supported test runners are CucumberJS, Jasmine, Jest, Karma and Mocha. Lastly the StrykerJS configuration needed for Callisto should be working with the example program. Most notably, Stryker must be configured to collect kill- and coverage information for each test case and mutant. For this StrykerJS version 5.4.0 or later is required.

As was mentioned before in Section 4.1, in this project the test suites of example programs are not expanded until adequate. Quality can therefore only be determined for the mutants killed by the initial test suite[1]. Remember that the quality of a mutation operator is determined by taking the average of the qualities of its mutants. If only a handful of mutants are killed, then the resulting quality of the operator will be unreliable. Therefore the last criterion states that a minimum of 200 mutants are generated and killed in an example program. This value was chosen based on a previous initial experiment done with a program called Robobar (see Section 5.3), which contained 97 killed mutants. Because of the large amount of mutation operators present in StrykerJS (see the list in Section 4.2), many generated very few mutants for Robobar, which resulted in unreliable quality measurements. Thus for a reliable result a minimum of 200 killed mutants is deemed sufficient. Using (much) more than 200 mutants should show little difference in quality and only make it more reliable. To meet this requirement a program should be of sufficient size to generate enough mutants, and have a test suite that kills at least 200 of these mutants.

Example Programs Used

Table 5.1 gives an overview of the nine example programs used to evaluate Callisto. All of these programs were accessed using GitHub.

[1] Quality cannot be determined for survived, non-equivalent mutants.

Name	Mutant Count	Mutation Score	Description
BigMath [2]	1671	88.51%	A library for working with large numbers
CucumberJS [5]	2677	66.23%	A test framework adding cucumber testing to JavaScript
Express [13]	1984	88.77%	“Fast, unopinionated, minimalist web framework for Node”
Freecodecamp [14]	8465	10.02%	FreeCodeCamp.org’s open-source codebase and curriculum
Minesweeper [29]	398	97.14%	Well-known puzzle game implemented in JavaScript
Mutation Testing Elements [44]	1170	74.28%	Metadata and visualisation for the mutation reports of Stryker
Nest [28]	8779	51.85%	Framework for server-side Node.js apps
StrykerJS Core [46]	3052	69.82%	Core module of StrykerJS
StrykerJS Instrumenter [47]	1197	90.63%	Instrumenter module of StrykerJS

Table 5.1: Overview of all example programs used to evaluate Callisto. The mutation score shows the achieved score using the initial test suite.

Several of these programs were found by looking through GitHub’s most starred public repositories under the topic ‘Node.js’[2]. Node.js projects have a high chance of being compatible with StrykerJS. The other programs were referred to this research project by peers and supervisors.

As the data consists of mutants, the number of generated mutants for a program are used as an indication of its size. This is not directly comparable with using the number of lines of code, a more traditional measure of program size, as certain code can cause more mutants to be generated than others. However for the purposes of this project it is fitting, as the mutants form the data for Callisto.

There are several reasons why these programs were chosen. First of all, they cover a wide variety of types of programs. This includes games (Minesweeper), educational (Freecodecamp), testing (CucumberJS), maths (BigMath), frameworks (Express, Nest), frontend (Mutation Testing Elements) and mutation testing (StrykerJS) software. This diversity ensures the group better represents the wide spectrum of the practical applications of JavaScript, and no bias is possible by only focussing on one type of software.

Second of all, for five of these programs StrykerJS was used during development. These are BigMath, Minesweeper, Mutation Testing Elements, StrykerJS Core and StrykerJS Instrumenter. This ensures that the test suite of those programs has been created with the help of Stryker, which generally means a higher mutation score is achieved, as developers could react to survived mutants. When the developers make use of the Stryker Dashboard, the mutation report these programs initially generate is also available upfront. This helps assessing the suitability of an example program, without taking the time to set up StrykerJS.

Third of all, some of these programs achieve a high mutation score. This ensures that a large portion of the generated mutants are killed, which provides more data to calculate quality. Furthermore a higher mutation score brings the test suite closer to adequate without requiring work. This decreases any possible bias introduced by the decision not to expand the test suites (see Section 5.3).

[2]The URL for this is `https://github.com/topics/nodejs?o=desc&s=stars`.

Fourth of all, large programs are preferred, simply because they will generate more mutants. This gives a higher chance of passing the third example program criterion, which requires a minimum of 200 killed mutants. The more mutants are generated, the lower the mutation score must be to pass this requirement. When searching for example programs one generally[3] does not know beforehand how many mutants will be generated from it exactly, or how many of these will be killed. Freecodecamp is an example of such a case. It seemed a large program, which ended up having over 8000 mutants. It has a very low mutation score of only 10%, significantly lower than all the other example programs. However, it is included as 10% of 8000 still results in 800 killed mutants. This passes the criterion, and also provides more data than for example Minesweeper, which has a near perfect mutation score.

Last of all, several programs were found because they are quite well-known. This is seen as an advantage, as well-known open-source software has a tendency to be well-tested[4], partly because all its users have the ability to contribute to and check the software. Some of the programs above were found due to their popularity on GitHub (measured in 'stars'). These are CucumberJS, Express, Freecodecamp and Nest.

5.3 On Inadequate Test Suites

Estero-Botaro et al. state that when calculating quality, an adequate test suite should be used [12]. This is recommended, as any mutants left alive can influence the resulting quality of its mutation operator. The workflow shown in Figure 4.1 has been created from the process described by Estero-Botaro et al. and therefore contains a task dedicated to ensuring that the test suites of example programs are made adequate.

However, the work involved in this is far greater than that of all other tasks combined. Expanding the test suite of any program requires a good understanding of the functionality, structure and programming of said program. Furthermore, creating an adequate test suite implies that all equivalent mutants must also be found. As this cannot be easily automated, it requires even more work.

As an experiment to gauge the effort required for this work, the test suite of the program Robobar was expanded. Robobar is a small web application written in JavaScript using Angular by the developers of Stryker as an example of the effectiveness of mutation testing [45]. It features a test suite with 100% code coverage, but a mutation score of only 58.73%, to show that code coverage is not always a suitable metric to measure the effectiveness of a test suite. Robobar contains a total of 126 mutants and a test suite with 16 test cases. During the expansion of the test suite it was discovered that several parts of the code were difficult to test, due to the nature of Angular. Thus these mutants were ignored. Out of the remaining 98 mutants, 97 were killed after writing an additional 8 test cases. The surviving mutant was an equivalent mutant. This work took a full day, as discovering which mutants could be ignored was not trivial. This example shows that significant time is needed for the expansion of a test suite, and unforeseen complications may arise in the process.

In the end Robobar is not used as an example program, as it is too small. Despite its size it still took considerable time to expand its test suite. The amount of time needed for this step scales with

[3] Unless Stryker has been used before.

[4] Being open source, the test suite can also be inspected beforehand.

the size of the program. Therefore, with limited time, it effectively restricts the size of example programs that can be used. Thus the decision was made to forego the expansion of test suites to adequacy, and instead use the unaltered test suites that come with example programs to calculate quality. This works by only taking into account the mutants which are killed, and ignoring anything else. Effectively this subset of killed mutants can be viewed as a 'new' set of generated mutants, for which the test suite achieves a mutation score of 100% and is thus adequate.

The advantage of this decision is that far larger and more complex programs can be used, such as Nest. Expanding the test suite of Nest to adequate is simply infeasible in the scope of this project[5]. Using larger programs means that far more data can be used to calculate quality, which makes the result more reliable. The nine example programs above provide 11,033 killed mutants to calculate quality with. As a comparison, the experiments of Estero-Botaro et al. only used 749 killed mutants [12].

This section investigates how using inadequate test suites can influence the found quality of mutation operators. This is done to ensure that this decision will not significantly skew the quality results, as this could compromise the mutation levels that are designed based on those results as well.

Deviation Experiment

There is a risk that by ignoring a large part of generated mutants, the calculated quality of mutation operators deviates from its 'true' value when all mutants are killed. In order to measure this deviation, a small experiment is performed. This consists of removing a portion of killed mutants from an example program and comparing the resulting quality with that of the whole set of killed mutants. The set of mutants that is removed is chosen randomly.

This experiment was done using BigMath and the two StrykerJS modules, Core and Instrumenter. In order to get a better view of the deviations that occur, the experiment was performed by removing either 10%, 20% or 30% of killed mutants from the example program. Furthermore, for each case the experiment was repeated four times with a different random set of mutants removed, so that the results were less biased for the choice of removal. More repetitions could have been added to increase the reliability of the results, but for the purposes of this experiment four times is deemed sufficient.

The random sets were determined using a Random Number Generator (RNG), for which different numerical seeds were used to choose different sets of mutants. The deviated quality can either end up higher or lower than the original, which is both seen as equally damaging. Each result is therefore the average *absolute* quality deviation of all mutation operators involved in that program. Figure 5.1 shows the average result over the four iterations for all three programs in a graph. See appendix B for the individual results per program and iteration.

Looking at the graph it seems the deviation in quality scales linearly with the percentage of mutants removed. This cannot be concluded with certainty however, as the experiment was not performed with higher percentages of removal of mutants.

Note that none of these programs have an adequate test suite. Therefore the deviations in quality are measured from the 'true' quality calculated according to the method in this section: by

[5] One would practically need to become an active contributor and developer for Nest.

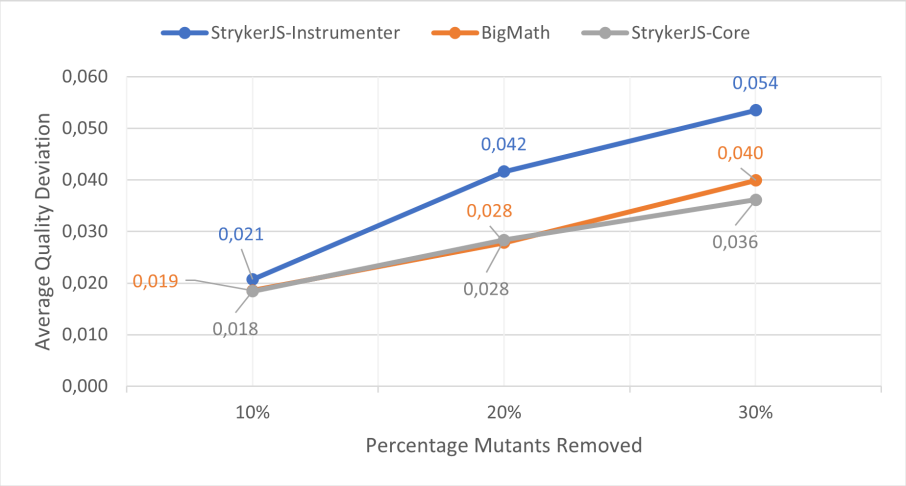

Figure 5.1: Quality deviations for removing 10%, 20% and 30% of killed mutants from BigMath, StrykerJS Core and StrykerJS Instrumenter.

only using the mutants which are killed by the (unexpanded) test suite and ignoring any others. This is not a problem for this experiment, as it is assumed that setting the 'true' quality does not affect the measured deviations. After all, the deviations are relative to the 'true' quality.

An interesting phenomenon that was observed during this experiment is that some mutation operators consistently had a low or high quality deviation. For example, the BlockStatement mutator in BigMath originally[6] achieved a quality of 0.027. Its average deviation when removing 10% mutants was 0.001: it practically remained the same. This can be explained by looking at a different kind of deviation: the absolute deviation of the original quality of the individual BlockStatement mutants, which is 0.045. Remember that the quality of a mutation operator is simply the average of the qualities of its mutants. In the case of BlockStatement the quality of all its mutants lies close to 0.027, which means that removing some of these mutants will influence the resulting quality very little. Conversely, the UnaryOperator mutator[7] has a relatively high quality deviation when removing 10% of mutants: 0.054. This can be an indication that the absolute deviation in quality of the individual mutants is high as well. This is indeed the case: it is 0.391. When some mutants with outlier quality values are part of the 10% removed mutants, the resulting new quality will also deviate more. Thus, it seems that this experiment can also be used to gauge the deviation of the quality of individual mutants under a mutator. To test this hypothesis, this deviation was calculated for the mutators of BigMath, and compared to the average deviation of quality when removing 10% of mutants. The result is visible in Figure 5.2.

Here, the blue bars show the relative size of a number compared to others in its column. It can be seen that in general where one deviation is high or low, so will the other be. The BooleanLiteral mutator is a clear exception to this rule: its deviation of quality of individual mutants is relatively very high, but its quality deviation when removing mutants is not. This could be explained by the fact that the quality of an individual mutant can change when other mutants are removed. In other words, the quality of a mutant is influenced by other mutants. This can be seen in Formula 3.14: the value of $|C_t|$ is directly dependent on the presence of other mutants. Thus, due to this effect, when 10% of mutants were removed from BooleanLiteral, the remaining 90% of mutants must

[6] I.e. without removing mutants.

[7] This experiment was held prior to the implementation of deducing mutation operator names, so quality is determined per mutator.

Mutator	Avg Deviation	Mutant Deviation
ObjectLiteral	0,039	0,356
BooleanLiteral	0,010	0,36
BlockStatement	0,001	0,045
ConditionalExpression	0,011	0,194
EqualityOperator	0,019	0,195
UnaryOperator	0,054	0,391
LogicalOperator	0,019	0,135
ArithmeticOperator	0,016	0,243
ArrowFunction	0,013	0,173
StringLiteral	0,021	0,128

Figure 5.2: Comparing the relative sizes (blue bars) of deviation of quality when removing 10% of mutants (left column), and deviation of quality of individual mutants (right column), of mutators in BigMath.

have resulted in a quality not too dissimilar to that of the original 100%. This would explain the relatively low deviation when removing mutants, even though there is a high deviation in quality among the mutants. Looking back at Figure 5.1, this effect could also explain why the quality deviations for StrykerJS Instrumenter are higher than those of the other two programs.

Conclusion

In conclusion, the experiment described above shows that there is indeed a deviation in the found quality when only a subset of generated mutants are used. However, the found deviation is quite low, with absolute values under 0.1 when 30% of mutants or less are removed. Thus it is deemed valid to forego expanding test suites to adequacy for calculating quality. The deviation could still be harmful to the design of mutation levels using quality, but the benefits of skipping this step outweigh the danger. This way more and larger example programs can be used. Furthermore, the risk can be managed by using example programs with high mutation scores, such that the test suite is still close to adequacy, resulting in a minimal deviation of quality. Looking at Table 5.1, most example programs have a high mutation score, with the exception of CucumberJS, Freecodecamp and Nest. Finding programs with a high mutation score is unfortunately not easy. For a program to achieve a mutation score as high as Minesweeper, it usually means that Stryker was already used in the development. As of writing, there are still very few open-source programs that use Stryker and are of sufficient size. Therefore programs such as Freecodecamp, where the mutation score is very low, were still included as they can provide a large amount of data.

Another downside to not creating adequate test suites is that equivalent mutants are also not detected. Remember that mutation operators are penalised for generating equivalent mutants: these are given a quality of 0. Therefore this penality is removed. It is estimated that this will have little impact on the resulting quality however, as it is assumed that equivalent mutants are quite rare in StrykerJS. For example, in Robobar only one equivalent mutant was found out of 98 mutants. Nonetheless, an overview of how many equivalent mutants are generated by each of the mutation operators of Stryker could have been valuable information for the developers of Stryker.

5.4 Methodology

The main experiment that is done with Callisto is determining the resolution and performance impact of mutation operators in StrykerJS. This is done primarily to design and assess mutation levels for these mutation operators. In addition it is used to assess the difficulty of applying this process. It also provides an opportunity to evaluate Callisto and see how it performs in a realistic use case. The data used for this experiment comes from the nine example programs discussed in Section 5.2. There it is explained how and why these programs were chosen.

The steps that were performed with Callisto using these example programs are as follows:

1. Run StrykerJS with the correct configuration for each example program and collect the resulting JSON files.
2. Manually edit JSON files until they are compatible with Callisto.
3. Use Callisto to merge all JSON files into one large JSON file.
4. Run Callisto with the large JSON file as input to determine resolution and performance impact.

The first step requires using StrykerJS with the correct configuration, such that all kill- and coverage information is collected. This comes down to setting configuration options `disableBail: true` and `coverageAnalysis: perTest`. To enable the generation of JSON files, the option `reports` should contain `"json"`. For the experiments in this project the option `timeoutMS` was set to 60,000 ms. This setting adds one minute of time to the timeout before Stryker considers a mutant as a timeout mutant. This was done to minimise the number of false positive timeout mutants, as a computer doing mutation testing may simply not be given the computational resources to execute all test cases in the normal time limit due to circumstances[8].

As is explained in Section 4.2, Callisto deduces mutation operator names using the information in the JSON files. Due to several problems with test frameworks that Stryker uses, this information occasionally contains small mistakes. This prevents Callisto from deducing a mutation operator name. Step 2 in the methodology therefore consists of manually correcting the mistakes by editing the JSON files by hand. This is necessary, as the mistakes are unpredictable and of a diverse nature. Furthermore it is only a temporary solution, until Stryker implements their own system of mutation operator names: taking the time to implement an automated solution was not deemed worthwhile. To help find such mistakes Callisto will report any mutants for which it cannot determine a mutation operator name through the terminal.

Callisto has the ability to accept multiple JSON reports in one session, determine results for each separately, and then merge these results at the end. However, the mean absolute deviation of quality cannot be merged in any statistically meaningful way. Therefore, if this deviation must be calculated for multiple programs, then the JSON files of those programs should be merged prior. Callisto will then regard it as one program. For this purpose Callisto has been expanded with the ability to merge JSON mutation reports as separate functionality. Finding the mean absolute deviation is desirable for this experiment, as it provides an answer to research sub-question 3, and thus step 3 of the methodology consists of merging all nine JSON files into one. Furthermore, several mutation operators in StrykerJS generate very few mutants, as the syntax token they operate on is rarely used in JavaScript and/or TypeScript. A low mutant count will result in an unreliable quality value. Therefore, by merging the nine example programs together, it is hoped enough mutants are collected for such mutation operators to create a more reliable result. Note that only

[8] For example, other programs running in the background.

mutant and test information in JSON files is merged, such that it can be used by Callisto. Other information that is stored, such as the project root directory or the Stryker configuration that was used, is not merged and is taken from the first JSON report during merging.

The last step consists of running Callisto with the merged JSON file as input. As explained in Section 5.3, test suites are not expanded, and therefore only killed mutants[9] are used. For this Callisto is configured to filter non-killed mutants, as was explained in Section 4.1. Similarly, no static mutants are used for calculating quality. Test executions are counted once with and once without static mutants. This allows comparing these two results to observe the performance impact of using static mutants.

5.5 Results

This section shows and discusses the results of applying the methodology in Section 5.4. Table 5.2 shows the raw results that Callisto produced. These are sorted alphabetically based on mutation operator name. This table combines the results of two separate sessions of Callisto: the third column (test executions) was taken from a second run where Callisto was configured to also count the test executions of static mutants. All other results are from one session, where only non-static mutants were used. Note that, aside from whether a mutant is static, test executions are counted for all types of mutants besides killed, and that this is therefore not influenced by the decision to use inadequate test suites. For minimising the test suite Callisto was configured to use the SAT solver, as several indecisive minimisations occurred when using the GLOP solver. Appendix C shows the results that Callisto produced for each example program individually.

As can be seen from the 'count' column, the number of killed mutants a mutation operator may generate ranges from tens (e.g. OptionalChaining) to thousands (e.g. BlockStatement, StringLiteralEmpty) of mutants. Note that the count represents killed, non-static mutants only: it shows how many mutants were used to calculate the quality of an operator, and *not* how many were generated by that operator in total. However, it still gives a good overview of which types of mutants are common and which are rare. Subsequently, it indicates which syntax elements occur often in the example programs, which represents JavaScript and TypeScript as a whole. For example, looking at the counts of ConditionalExpression mutation operators, it is apparent that the strict equality operator `===` is used far more than the 'normal' equality operator `==` in JavaScript.

There is one mutation operator in StrykerJS which is not present in Table 5.2: UpdateOperatorPre--To++. This is because the pre-decrement operator (`--a`) does not occur in any of the nine example programs, and subsequently this mutation operator did did not generate a single mutant. This is most likely caused by the fact that the pre-decrement operator is rarely used.

Table 5.2 makes it seem like mutating `===` to `false` occurs more often than mutating to `true`, which would make no sense considering both mutations originate from the same syntax token and therefore would be equally frequent. The difference in count is caused by the mutant filtering that Callisto does: killed and non-static mutants are ignored. Thus, ConditionalExpression===Tofalse may have more killed mutants and fewer static mutants than ConditionalExpression===Totrue.

[9] That is, only mutants killed by the initial test suite.

Mutation Operator	Count	Test Executions	Test Executions Non-static	Quality	Quality Deviation
ArithmeticOperator%To*	14	440	440	0.929	0.027
ArithmeticOperator*To/	39	5,941	549	0.713	0.344
ArithmeticOperator/To*	24	1,067	447	0.844	0.141
ArithmeticOperator+To-	70	3,045	1,697	0.682	0.361
ArithmeticOperator-To+	73	1,708	1,708	0.728	0.321
ArrayDeclarationEmpty	164	58,218	2,218	0.774	0.251
ArrayDeclarationEmptyConstructor	5	1,433	85	0.767	0.08
ArrayDeclarationFill	83	19,340	3,979	0.776	0.254
ArrowFunction	371	90,508	5,600	0.772	0.246
BlockStatement	2912	169,046	45,468	0.499	0.314
BooleanLiteralfalseTotrue	112	23,899	2,635	0.707	0.357
BooleanLiteralRemoveNegation	397	22,693	8,894	0.756	0.266
BooleanLiteraltrueTofalse	100	32,168	2,133	0.879	0.164
ConditionalExpression!==Tofalse	106	18,329	9,917	0.802	0.243
ConditionalExpression!==Totrue	93	12,056	7,094	0.743	0.254
ConditionalExpression!=Tofalse	9	396	130	0.818	0.06
ConditionalExpression!=Totrue	7	396	130	0.756	0.113
ConditionalExpression<=Tofalse	23	174	174	0.694	0.362
ConditionalExpression<=Totrue	16	144	144	0.852	0.065
ConditionalExpression<Tofalse	44	6,018	2,568	0.886	0.126
ConditionalExpression<Totrue	34	3,142	3,142	0.794	0.249
ConditionalExpression===Tofalse	463	40,842	19,578	0.69	0.274
ConditionalExpression===Totrue	438	43,255	17,099	0.623	0.316
ConditionalExpression==Tofalse	6	1,135	1,135	0.833	0
ConditionalExpression==Totrue	5	437	171	0.8	0
ConditionalExpression>=Tofalse	31	2,693	2,693	0.544	0.408
ConditionalExpression>=Totrue	29	1,715	1,715	0.786	0.194
ConditionalExpression>Tofalse	123	6,656	2,810	0.794	0.236
ConditionalExpression>Totrue	114	17,409	2,779	0.691	0.336
ConditionalExpressionConditionTofalse	755	94,863	32,381	0.804	0.184
ConditionalExpressionConditionTotrue	806	76,647	27,458	0.693	0.26
ConditionalExpressionEmptyCase	142	4,083	2,113	0.704	0.376
EqualityOperator!==To===	80	5,798	4,327	0.804	0.255
EqualityOperator!=To==	16	573	307	0.546	0.41
EqualityOperator<=To<	15	118	118	0.937	0.013
EqualityOperator<=To>	16	97	97	0.931	0.021
EqualityOperator<To<=	29	2,683	2,683	0.892	0.109
EqualityOperator<To>=	41	3,754	1,514	0.82	0.236
EqualityOperator===To!==	364	23,708	10,354	0.68	0.295
EqualityOperator==To!=	17	1,369	219	0.927	0.021
EqualityOperator>=To<	18	787	787	0.933	0.022
EqualityOperator>=To>	16	1,774	1,774	0.934	0.02
EqualityOperator>To<=	93	2,138	2,138	0.689	0.397
EqualityOperator>To>=	56	2,126	2,126	0.78	0.265
LogicalOperator&&Toǁ	267	30,220	7,582	0.775	0.227
LogicalOperator??To&&	27	1,761	414	0.955	0.014

LogicalOperator||To&&	246	35,834	10,839	0.785	0.223
ObjectLiteral	456	113,759	9,945	0.627	0.364
OptionalChaining	24	2,277	929	0.945	0.023
Regex	45	97,309	889	0.688	0.232
StringLiteralEmpty	1410	644,497	47,412	0.61	0.325
StringLiteralFill	111	5,190	4,023	0.682	0.322
UnaryOperator+To-	10	167	167	0.88	0.032
UnaryOperatorRemove~	1	1	1	0	0
UnaryOperator-To+	40	15,086	1,471	0.488	0.435
UpdateOperatorPost++To--	22	1,354	1,354	0.876	0.096
UpdateOperatorPost--To++	4	33	33	0.625	0.125
UpdateOperatorPre++To--	1	2	2	0	0
Totals	11,033	1,752,311	320,589		

Table 5.2: The quality and performance impact of all mutation operators of StrykerJS based on the nine example programs of Table 5.1. Results are sorted alphabetically based on mutation operator name. The performance impact can be seen with and without the inclusion of static mutants.

Figure 5.3 shows a pie chart of the mutant counts relative to the total of 11,033. It is clear that just four mutation operators contribute more than half of all mutants. BlockStatement and StringLiteralEmpty account for 39.2% of all mutants alone. This is not unexpected, as the syntax tokens which these mutation operators operate on, code blocks and non-empty strings, are common in JavaScript and TypeScript. The mutant count also hints at the performance impact of mutation operators. After all, more mutants will require more test case executions.

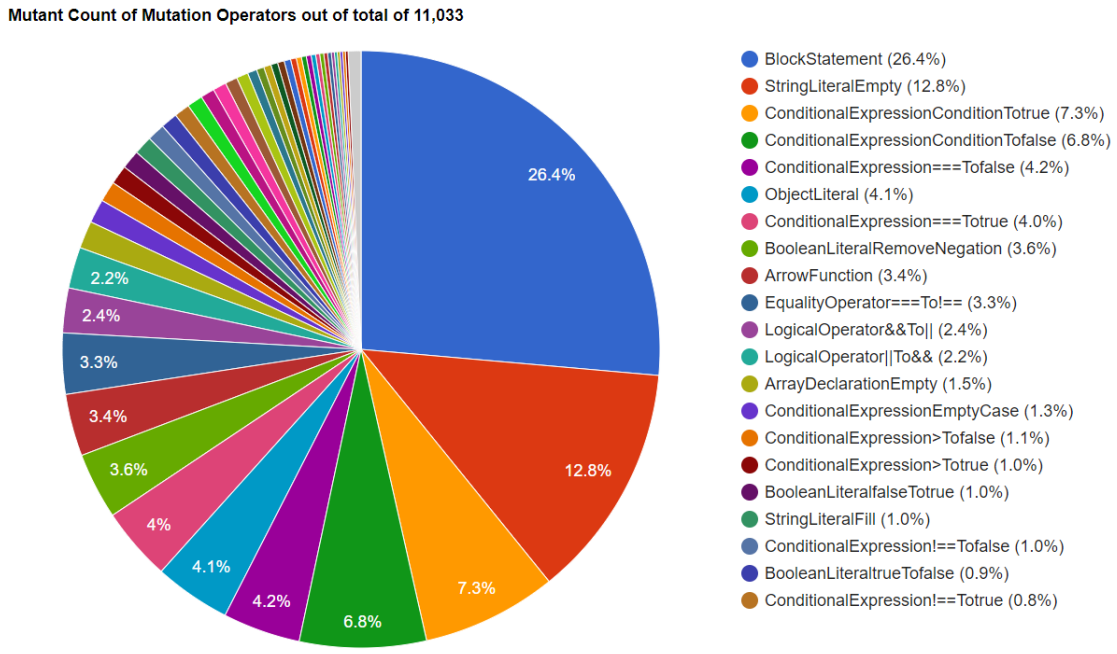

Figure 5.3: Percentage of mutant counts of mutation operators relative to total of 11,033. This only includes mutants which are killed *and* non-static. Note that the legend only shows the 21 mutation operators with the highest mutant count, due to space constraints.

Figure 5.4 shows two more pie charts concerning the test case execution counts including and excluding static mutants. Figure 5.4 A includes static mutants, for a total of 1,752,311 test case executions. This shows that although BlockStatement accounts for the most mutants, StringLiteralEmpty has the highest performance impact when static mutants are included. The third largest slice here belongs to ObjectLiteral, which contributes 6.5%, even though it only contains 4.1% of mutants. The Regex mutation operator seems even more out of place here: it accounts for 5.6% of test executions, but only contains 45, or 0.4% of mutants. This surprising number of test executions can be explained by excluding the static mutants. Figure 5.4 B contains an analogous pie chart for test executions for non-static mutants only, for a total of 320,589. Here Regex only accounts for 0.28% of test executions. Comparing the two values shows that 99.09% of test executions for all mutants of Regex originate from static mutants. In other words, ignoring static mutants could make testing the mutants of Regex approximately 100 times faster. A similar story can be told for many other mutation operators: most have 50% or more of test executions come from static mutants. Some mutation operators do not have any static mutants, such as ArithmeticOperator%to*. This is because they all generated few mutants across the nine example programs, which happened to not include any static mutants.

Comparing the two different counts of test executions shows that static mutants account for a majority of test executions. Only 18.30% (320,589 out of 1,752,311) of test executions are non-static. Therefore ignoring static mutants could speed up the testing of mutants approximately six times. This puts into perspective what performance impact static mutants have during mutation testing. That is not to say that static mutants are less useful than others: they can also contribute to the design of a test suite. However, as was explained in Section 4.1, static mutants have a lower quality on average. This is why static mutants have been ignored in the calculation of quality in Table 5.2.

Figure 5.5 shows the mean absolute quality deviation, sorted by quality. This graph serves to illustrate how the deviation of a mutation operator relates to its quality. It clearly shows that this deviation decreases when the quality approaches 1. This is logical, as quality cannot be higher than 1. The deviation in quality becomes much higher when it goes towards 0.5. This indicates that for those mutation operators the quality of individual mutants is more scattered. This would in turn explain why their quality lies around 0.5, as that is the middle value that quality can take. Thus the quality of those mutation operators is less reliable. However, when the deviation of a mutation operator is significantly lower than that of its neighbours in Figure 5.5, it indicates that this quality calculation is more reliable. This is the case, for example, for the BlockStatement mutation operator, which corresponds to the second to last point in Figure 5.5. Other low values of deviation occur due to a low mutant count. For example, the two neighbouring visibly lower values around the middle of the graph correspond to ArrayDeclarationEmptyConstructor and ConditionalExpression!=Totrue, which only have five and seven mutants respectively.

This figure also gives a general overview of the quality that mutation operators have achieved in the nine example programs. LogicalOperator??To&& has the highest quality at 0.955, and UnaryOperator-To+ has the lowest quality at 0.488. Thus almost all mutation operators have a quality above 0.5. What is noteworthy is that the top ten mutation operators all have a mutant count under 30. This makes their quality less reliable, as fewer mutants were used to calculate it. There are two mutation operators with a quality of 0 as they only have one mutant each[10]. These are UnaryOperatorRemove~ and UpdateOperatorPre++To--. They therefore also have a deviation of 0.

[10]Only one killed mutant automatically results in quality 0, see Formula 3.14.

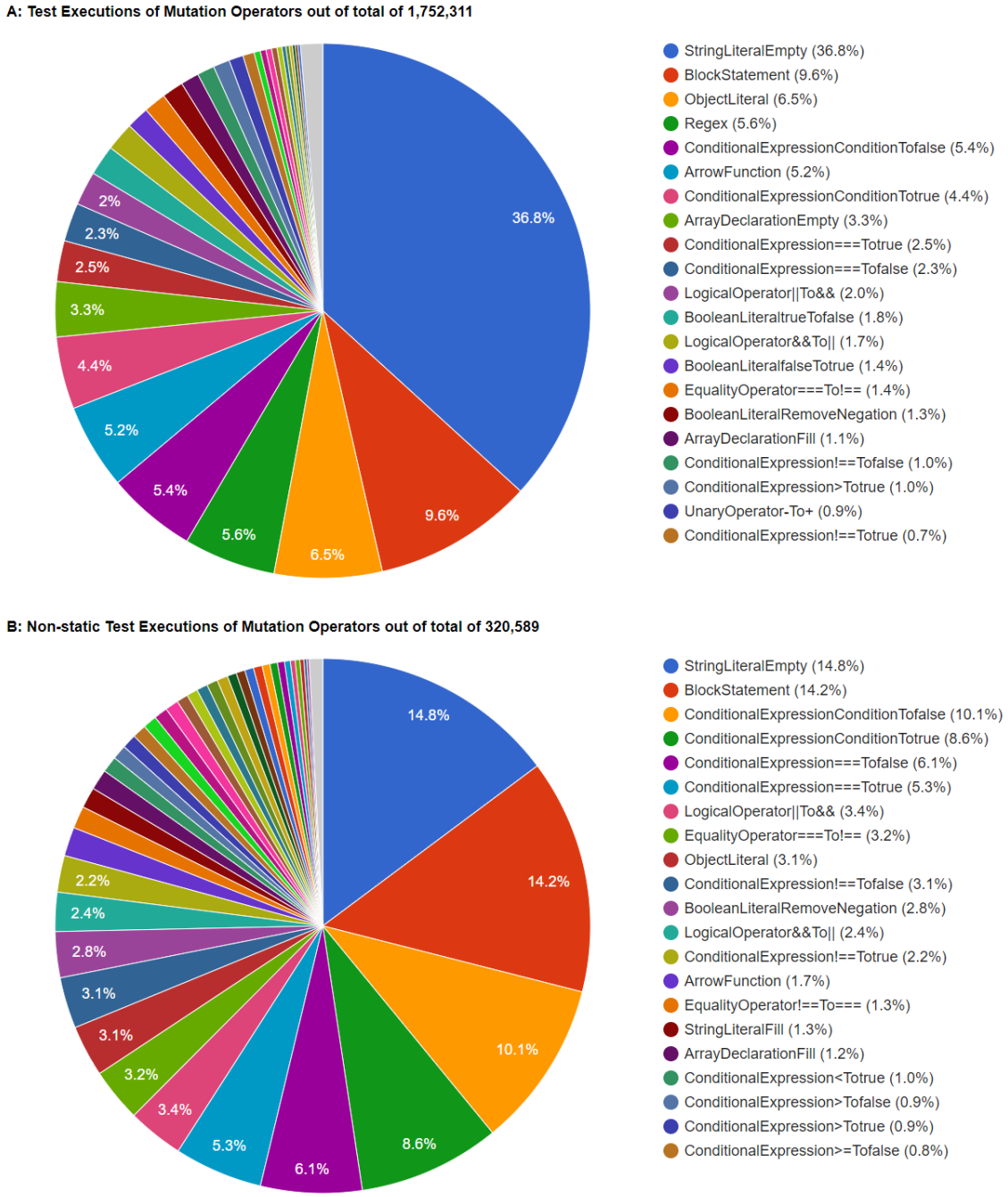

Figure 5.4: A: Percentage of test executions of mutants from mutation operators relative to total of 1,752,311.
B: Percentage of test executions of non-static mutants from mutation operators relative to total of 320,589. Note that the legends only show the 21 mutation operators with the highest test execution count, due to space constraints.

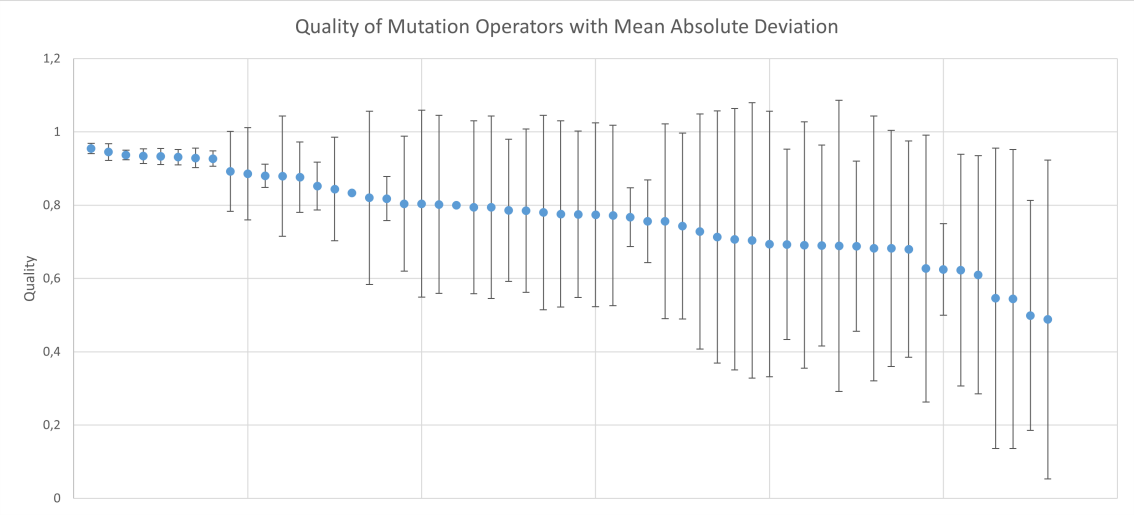

Figure 5.5: Mean absolute quality deviation of all mutation operators, sorted by the achieved quality.

It can also be seen that mutation operators which are paired achieve differing quality values. For example, BooleanLiteralfalseTotrue has quality 0.707, whereas BooleanLiteraltrueTofalse has a higher quality of 0.879. One would expect such mutation operators get the same quality, as they perform a very similar mutation. The difference is most likely caused by a coincidence in the data: in the nine example programs used, BooleanLiteraltrueTofalse happens to achieve a higher quality by chance.

To conclude, this experiment has determined the resolution and performance impact of the mutation operators of StrykerJS, using the nine example programs of Table 5.1. It has also shown that the applicability and difficulty of the process to get to these results is for a large part dependent on the used mutation testing framework. Looking back at the methodology in Section 5.4, the greater part consists of configuring StrykerJS to provide the required information, and subsequently preparing this information for use with Callisto. Therefore, if the used mutation testing framework is able to provide the needed information in a suitable format, then the only remaining step is to let Callisto use this to determine the resolution and performance impact.

The other part of the process consists of finding a set of suitable example programs. This determines how many mutants can be analysed for each mutation operator. Furthermore, to improve the results it is recommended that adequate test suites are developed for each program, although Section 5.3 has shown that it is possible to skip this step. In conclusion, performing the experiment has shown that finding and mutating suitable example programs is the most arduous task in the process. Once this is done Callisto can be easily applied. The process is therefore deemed applicable to other mutation testing frameworks besides Stryker, given that said framework provides the required information to calculate quality and performance impact in a usable format.

This experiment has also provided an opportunity to evaluate the performance of Callisto itself. This has shown that it is able to come to a result in an acceptable time frame. For any example program alone, using a modern laptop, Callisto takes less than 20 seconds to finish, with most finishing in less than five seconds. Once all nine example programs are merged, processing with Callisto takes approximately 15 minutes. This is longer than the times required for the individual programs summed, as the time needed to minimise a test suite scales exponentially with the size of

the test suite, and the number of mutants involved. Indeed, minimising the test suite takes up the majority of the time needed by Callisto. Still, 15 minutes is quite adequate for a tool like Callisto, as it should not be needed frequently and can be automated using a CI/CD pipeline.

The experimental results presented here will be further used in Chapter 6 to create several trial mutation levels for StrykerJS, which are subsequently assessed to finalise the evaluation.

6. Mutation Levels

This chapter continues with the results of Section 5.5 by using them to design mutation levels for StrykerJS. First it is explained how the effectiveness and performance of a mutation level can be determined using Callisto. Then several mutation levels are designed based on the found resolution and performance impact of mutation operators. To conclude the evaluation started in Chapter 5, the effectiveness and performance of these levels is determined, and the results are discussed.

6.1 Effectiveness of Mutation Levels

As was mentioned in the introduction, Section 1.1, selective mutation is a technique to speed up mutation testing [27]. This technique is very similar to that of mutation levels: mutation operators are removed to lower the cost of mutation testing, while trying to retain effectiveness. The goal is to find a subset of mutants which is representative of the full set. To measure the effectiveness of a chosen subset, a test suite is obtained for that set only. The subset is effective, if the mutation score achieved by that test suite for all mutants is comparable to the mutation score achieved for the subset. However, Delgado-Pérez et al. point out that the quality metric used here cannot be judged in such a way [6]: "the presented quality metric focuses on test suite improvement with high-quality test cases. Therefore, it is not the purpose of the quality metric to value operators for their potential to predict the mutation score of the full set of operators." They give a clear example to illustrate this. If mutants are selected based on their quality, resistant hard-to-kill mutants are a good choice, as these mutants are killed by only one test case, which kills only that mutant (see Section 3.4). Then the test suite for such a subset of mutants would consist of only these test cases. However, they would achieve a poor mutation score for the whole set of mutants, as they only kill the subset of mutants, according to their definition. The effectiveness technique of above would therefore judge this a poor subset of mutants for selective mutation, whereas for the design of mutation levels such mutants are desirable.

Delgado-Pérez et al. therefore propose a new metric for evaluating the effectiveness of a chosen subset of mutation operators. This metric will reward a subset when it leads to the design of many high-quality test cases, by investigating survived mutants of those operators. Instead of the mutation score, the size of the minimal test suite corresponding to a subset is compared to the size

of the original minimal test suite. Formally:

$$\mathcal{E}_L = \frac{|T_L|}{|T|} \cdot 100 \tag{6.1}$$

Here $\mathcal{E}_L$ is the effectiveness of the subset of mutation operators L, and $|T_L|$ is the size of the minimal test suite induced by subset L. T is the original minimal test suite when all mutation operators are used. Note that T_L is a subset of T, i.e. it is derived from the original minimal test suite. $\mathcal{E}_L$ is therefore the percentage of test cases remaining in T_L out of T. As a mutation level is nothing other than a subset of mutation operators, $\mathcal{E}_L$ can also be interpreted as the effectiveness of mutation level L.

This metric is a suitable means for evaluating the effectiveness of a mutation level. When $\mathcal{E}_L$ is 100%, this means no test cases were lost when using mutation level L, but as a subset of mutation operators is used, fewer mutants are generated and therefore less time is needed to perform mutation testing. This metric reasons from the perspective of the software tester: they want to create a high quality test suite for their software. They can do this by using mutation testing and subsequently investigating any survived mutants. This should lead to the design of new test cases which kill more mutants. The tester can iterate this until they reach an adequate test suite[1], which was therefore induced by the survived mutants. When a mutation level L is used and $\mathcal{E}_L$ is 100%, they were able to create a test suite of equal size compared to using all mutants, while needing fewer mutants, thus saving performance.

Another metric which naturally follows from this is the amount of performance which is saved when using a mutation level. Callisto measures performance impact with the number of test case executions necessary during mutation testing. Therefore, the performance saved can be calculated using this information:

$$\mathcal{P}_L = \frac{X - X_L}{X} \cdot 100 \tag{6.2}$$

Here $\mathcal{P}_L$ is the performance of mutation level L, and X and X_L the number of test executions when using all mutation operators or only the mutation level, respectively. $\mathcal{P}_L$ is therefore the percentage of test case executions that is saved by using mutation level L, as opposed to using all mutation operators.

The ideal mutation level therefore strives to have a high effectiveness and performance. In order to calculate these two metrics, Callisto has been equipped with the option to give a list of mutation operators that comprises a mutation level. Then, using a provided example program, Callisto will first determine the minimal test suite for all mutants (T), and count the number of test executions (X). Then the mutation level is applied by only keeping mutants generated by mutation operators in the level. Much like with calculating quality for a mutation operator, Callisto then minimises the test suite T again with respect to the remaining mutants, resulting in T_L. Then the test executions of these mutants can be counted to find X_L. Now Formulae 6.1 and 6.2 are used to calculate the effectiveness and performance of the mutation level, and the results are reported back to the user in a short text file. Section 6.2 will use this functionality to gauge the usefulness of the mutation levels designed there.

[1] An adequate test suite is rarely created in practice.

6.2 Designing Mutation Levels

To conclude the evaluation performed with Stryker, several trial mutation levels are designed for StrykerJS. These are based on the results in Table 5.2. The resolution and performance impact determined by Callisto for each mutation operator is used to make decisions for which operators should be selected for a level, such that it has a high effectiveness and performance according to the two above metrics, $\mathcal{E}_L$ and $\mathcal{P}_L$.

Overview

The fewer mutation operators are included in a level, the more performance, as less test case executions are required. However, with fewer mutants the effectiveness of a level will decrease, as the minimal test suite needed to kill them will become smaller. Designing a mutation level is therefore a game of removing many test case executions without decreasing the size of the minimal test suite too much.

Due to the scope of this project, a thorough investigation into which methods are effective to design mutation levels, given the resolution and performance impact of mutation operators, is not performed. Instead, several trial mutation levels are designed here using techniques based on intuition. Therefore not all mutation levels here are suitable for practical use. Nonetheless, for the purposes of the evaluation performed in this project with Stryker, the results here show promise for the use of mutation levels to speed up mutation testing. The used techniques also provide an answer to research sub-question 4.

The first set of mutation levels is designed in a relatively straightforward manner: a quality threshold is set, and only mutation operators that achieved a quality greater than or equal to the threshold are included in the level. Given that the found quality values range from around 0.50 to 0.95 in Table 5.2, thresholds were set from 0.60 to 0.85, with increments of 0.05, for a total of six mutation levels. This increment is deemed right to illustrate how mutation levels behave throughout the spectrum of possible thresholds. No higher or lower thresholds are used, as this would result in mutation levels with either almost all, or barely any mutation operators.

Next a couple of levels are designed based on the performance impact measured by Callisto. First of all, a level is created where only StringLiteralEmpty is removed from the set, as it is the mutation operator with the highest performance impact. In a similar manner a level is created where the four mutation operators with the highest performance impact are removed, which are StringLiteralEmpty, BlockStatement, ConditionalExpressionConditionTofalse and ConditionalExpressionConditionTotrue. Lastly a performance impact threshold was set at 1%: only mutation operators which contribute less than or equal to 1% of test case executions were included: 40 out of 59.

Creating and testing the effectiveness and performance of a mutation level takes little effort, as only a list of mutation operators needs to be written and given to Callisto along with an example program to calculate the metrics. Therefore, in addition to the ones above, some mutation levels are created which intuitively do not seem like good mutation levels. They provide the opportunity to see how the performance and effectiveness of a 'badly' designed mutation level behave. Two of such levels are inversions of previously mentioned levels: they contain all mutation operators which its inverse does not and vice versa. These are two levels which contain only StringLiteralEmpty, and only the four mutation operators with the highest performance impact, respectively. A similar

level was created with only the BlockStatement operator included. Finally, to test the usefulness of the EqualityOperator- and LogicalOperator-based mutation operators, a mutation level is created where these operators are removed. This level is named 'No ROR', as effectively the ROR mutation operator as defined in literature ([27] [7] [21] [22]) is excluded.

All the above levels are designed using techniques based solely on the quality and performance impact of mutation operators. However, it should be remembered that mutation levels are meant to be used in practise with StrykerJS. Although the design of the levels up to now can be justified using quality and performance impact, on a surface level many of them seem like a random sample. A good example of this is a level where a mutation operator such as ArithmeticOperator+To- is included, but its counterpart ArithmeticOperator-To+ is not, due to (small) differences in quality or performance impact. For a user of StrykerJS that wants to use mutation levels this must seem like an odd decision. Therefore, the last set of designed mutation levels consists of several custom, hand-made levels which try to remove mutation operators consistently. These levels build upon themselves: The first custom level removes several mutation operators. Then the second custom level removes more operators in addition to custom level 1, and so forth. This ensures that further levels have a higher performance, but lower effectiveness than previous levels. By offering multiple custom levels this way a user is given the choice of how much effectiveness they wish to 'sacrifice' to gain performance.

The first custom level (Custom 1), is created by removing BlockStatement, StringLiteral, ObjectLiteral, Regex, ConditionalExpression===Tofalse, ConditionalExpression===Totrue, EqualityOperator===To!== and UnaryOperator-based operators. These were chosen as most had a large performance impact, while their quality was not exceptionally high. Any mutation of the syntax token `===` is removed, as this is a common token and therefore generates many mutants.

The second custom level (Custom 2) builds on this by additionally removing ConditionalExpressionEmptyCase, ConditionalExpressionConditionTofalse, ConditionalExpressionConditionTotrue and BooleanLiteralRemoveNegation. Once again these mutation operators are chosen as they have a high performance impact and relatively low quality out of the remaining ones in Custom 1.

Custom level 3 additionally removes ArrayDeclarationEmpty, ArrayDeclarationEmptyConstructor, ArrayDeclarationFill and ArrowFunction, for the same reasons as above. Finally Custom level 4 is the smallest of the set, where LogicalOperator||To&& and LogicalOperator&&To|| are removed as well. Appendix D shows the exact contents of these four custom levels in a table.

Evaluation

Table 6.1 shows the results for evaluating the above mutation levels using Callisto. As example program the merged JSON file of the nine example programs in Table 5.2 is used, as these are also the programs from which quality and performance impact is used to design the levels. When minimising test suites to calculate effectiveness, the SAT solver was used, just as for the results in Section 5.5. Performance was calculated by counting test case executions for non-static mutants only, as the same was done in Section 5.5. In addition to the effectiveness and performance, the percentage of mutants removed is also calculated. This value usually lies close to the performance, as the percentage of mutants removed correlates with the percentage of test case executions saved, which *is* performance.

Mutation Level Name	% Mutants Removed	Effectiveness ($\mathcal{E}_L$)	Performance ($\mathcal{P}_L$)
<1%testsexecuted	88%	26%	83%
Custom1	57%	69%	49%
Custom2	74%	48%	71%
Custom3	81%	37%	75%
Custom4	86%	28%	80%
No ROR	32%	90%	50%
Only4WorstPerforming	47%	85%	52%
OnlyBlockStatement	78%	63%	86%
OnlyStringEmpty	83%	37%	85%
Remove4WorstPerforming	46%	76%	32%
RemoveStringEmpty	17%	92%	15%
Threshold 0.60	23%	88%	16%
Threshold 0.65	50%	74%	39%
Threshold 0.70	66%	63%	60%
Threshold 0.75	70%	57%	65%
Threshold 0.80	87%	36%	80%
Threshold 0.85	96%	13%	96%

Table 6.1: The % of mutants removed, effectiveness and performance for all mutation levels. Results were obtained using Callisto.

Figure 6.1 plots the effectiveness against the performance of each mutation level. Levels placed close to the top-right corner have a high effectiveness and performance, making them better levels. An average trend line is added to better show the outliers.

Contrary to expectations, several of the ‘badly’ designed mutation levels achieved the best effectiveness to performance ratio. ‘No ROR’, ‘Only4WorstPerforming’ and ‘OnlyBlockStatement’ scored the best. ‘No ROR’ managed to retain 90% of the test cases, while removing 50% of test executions, which halves the time needed for executing the test suite for all mutants. This is under the assumption that all test cases take equal time to execute, as was explained before in Section 4.1. This indicates that the ROR-like mutation operators can be omitted without losing too many test cases for these example programs.

Using only the BlockStatement mutation operator resulted in an effectiveness of 63%, while 86% of test case executions are removed: the effectiveness at that performance level is relatively high. This could be caused by the nature of the mutations. First, BlockStatement empties code blocks, which are often present all throughout the code. In other words, BlockStatement mutations can be found spread throughout the code. Second, because of the drastic change introduced by such mutants, they are often killed by any test case that covers them. Combining these two observations, many test cases will remain in the test suite when minimising. At the same time, BlockStatement alone is responsible for 14% of test case executions, which is the second-highest. However, just

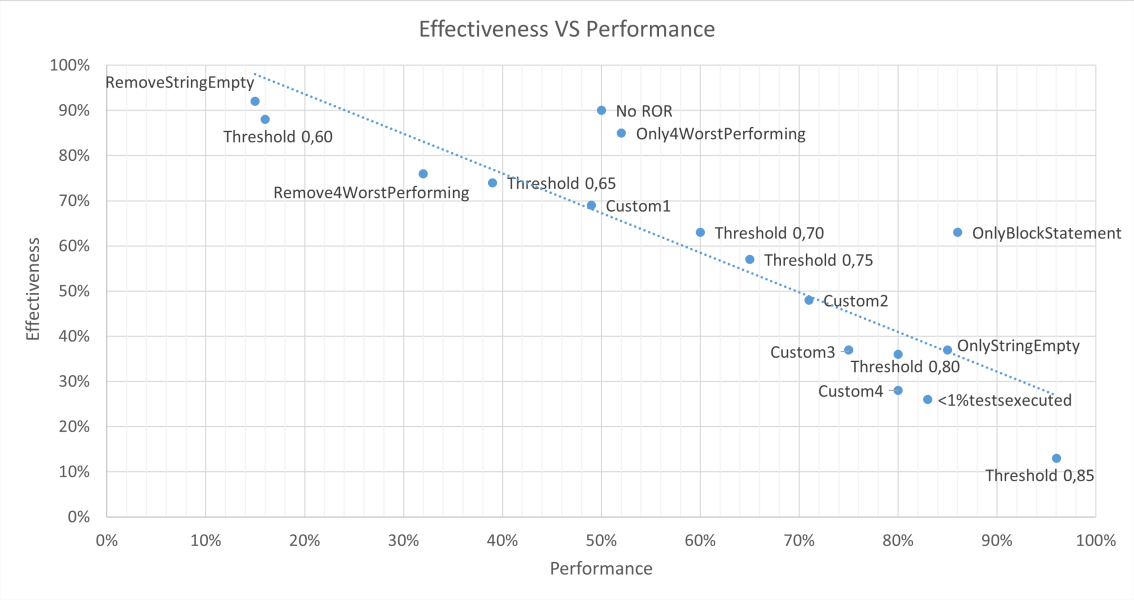

Figure 6.1: Effectiveness and performance of mutation levels visualised. A trend line is added to showcase outliers: the closer to the top-right corner of the graph, the better a level, and vice versa.

using BlockStatement will therefore lead to a decrease in test case executions of 86%.

BlockStatement may retain many test cases, but these will often be basic test cases. BlockStatement mutations do not often lead to well-designed test cases, as they can be easily killed by any test case using the concerned code block. In that sense, BlockStatement can give a tester a good overview of which code has not been covered yet.

Another factor that may have caused the OnlyBlockStatement mutation level to do so well, is the decision to use inadequate test suites for the nine example programs. It could be that on average, the test cases contained in the accompanying test suites are relatively simple. Perhaps more advanced and hard-to-design test cases would have been added if the surviving mutants were investigated. Such test cases would remain in the main minimal test suite, as they are needed to kill specific (non-BlockStatement) mutants. But if only BlockStatement mutants are used, then these are generally removed when minimising, resulting in a smaller test suite, and thus a lower effectiveness for such a level. Because test suites were not made adequate, such test cases are not present, and thus this could explain why the effectiveness of the OnlyBlockStatement level is higher.

The Only4WorstPerforming mutation level most likely did well for the same reasons as mentioned above for OnlyBlockStatement. After all, one of the mutation operators present in it is BlockStatement.

The threshold mutation levels mostly performed average. Following the trend line from top to bottom, the threshold levels can be found nearby in the order of the threshold set. This order is logical, as a lower threshold means more mutants are included, which raises the effectiveness but lowers the performance of a level.

Finally the four custom mutation levels achieved mixed results. Custom 1 and 2 perform average and are placed almost exactly on the trend line. Custom 3 and 4 are very close to custom 2,

but experience a drop in effectiveness. In the end custom 1 and 2 are favourable mutation levels, as they use a consistent list of mutation operators, and have a decent performance and effectiveness. Both levels can be added to StrykerJS, such that a user can choose how much effectiveness they want to lose to gain performance. Custom 1 retains a high 69% effectiveness, while removing 49% of test case executions, effectively doubling performance. Custom 2 flips these numbers and has an effectiveness of 48%, while removing 71% of test case executions, between three and four times faster.

What is apparent from these results is that there is a clear benefit to using mutation levels. All trial levels are placed in the top-right half of the graph in Figure 6.1[2]. This means that the performance percentage achieved is always higher than the effectiveness percentage lost compared to 100% effectiveness. This is a strong indication that mutation levels are a valid means to speed up mutation testing.

This experiment showed once again that when it comes to the performance of Callisto, minimising the test suite takes the most time. Testing the effectiveness and performance of a mutation level often takes longer than calculating the quality, as first the main test suite needs to be minimised, and afterwards the test suite corresponding to the mutation level must be as well. Callisto can only test one mutation level at a time. The main test suite of the example program must be minimised each time, which is the exact same process regardless of the mutation level. Therefore testing several mutation levels in a row is quite inefficient. A possible solution for this problem could be a function where Callisto can export the minimisation of a test suite, so that it can be reused later. Alternatively an option could be added for users to provide the size of the main minimal test suite for the process, after it has been determined once. After all, the only property of the main minimal test suite needed is its size when analysing a mutation level.

[2]I.e. they are above the line from the top-left to the bottom-right corners.

7. Conclusion

This chapter concludes the project. First the project is summarised and the research questions of Section 1.2 are revisited and answered, after which conclusions are drawn from the project. Next several aspects of the project and the threats to validity are discussed. Finally options for future work concerning Callisto and the project are described.

7.1 Conclusion

This project has proposed and evaluated the use of mutation levels as a technique to speed up mutation testing. A mutation level consists of a subset of mutation operators such that fewer mutants are generated without losing too much resolution. Thus the focus lies on keeping subtle, hard-to-kill mutants, such that the creation of a high-quality test suite is encouraged. In order to design mutation levels the resolution and performance impact of mutation operators is determined. Resolution is quantified using an existing quality metric from literature: the coverage-based quality by Delgado-Pérez et al. [6], which in turn is an improvement upon the quality metric by Estero-Botaro et al. [12]. Quality must be determined in an experimental setting for an example program and its accompanying test suite. Performance is measured in the same setting by counting the number of test case executions for each mutant. To calculate these two metrics the tool Callisto is designed and implemented, which automates large parts of the analysis of mutation operators.

To evaluate the effectiveness of designing and using mutation levels, the technique is applied to the mutation testing framework Stryker. Nine example programs written in JavaScript and/or TypeScript are mutated using StrykerJS. Callisto is then used to analyse all the involved mutation operators and determine their resolution and performance impact. These results were subsequently used to design several trial mutation levels for StrykerJS. The effectiveness and performance of these mutation levels were evaluated by Callisto using the decrease in minimal test suite size, and number of test case executions saved, respectively. The mutation levels show that there is indeed potential for an increase in performance for mutation testing, without losing too much resolution. The performed evaluation also showed that in the process for designing mutation levels, finding and mutating suitable example programs takes the most effort. After this, Callisto can be easily applied, so long as the used mutation testing framework provides the required information in a usable format.

Looking back at the five research sub-questions described in Section 1.2, several have clear answers now. Sub-question 1 and 2 asked how to determine the resolution and performance impact of mutation operators, respectively. Resolution is quantified by the coverage-based quality metric of Delgado-Pérez et al. [6], while performance impact is measured by counting the number of test case executions during mutation testing for the mutants of a mutation operator. Callisto was designed to calculate these metrics, based on the JSON mutation reports that Stryker creates.

Sub-question 3 asked how consistent the resolution of mutation operators was across multiple programs. This question was addressed through Callisto by determining the mean absolute deviation of quality for each mutation operator. The deviations in Table 5.2 and Figure 5.5 show that not every mutation operator is consistently given the same quality across the nine example programs. More on this in the discussion in Section 7.2.

Sub-question 4 concerned finding techniques to design suitable mutation levels, given the resolution and performance impact of mutation operators. Section 6.2 showed several quality thresholds being used for trial mutation levels. Based on Figure 6.1, thresholds set at 0.70 and 0.75 resulted in mutation levels with a high effectiveness to performance ratio. Another technique that was used is the removal of mutation operators with the highest performance impact. In Section 6.2 this was done by removing only one operator, four operators, or all operators with a relative performance impact of above 1%. Because the above techniques do not take the consistent removal of mutation operators into account, custom designed mutation levels were made as well, such that paired mutation operators are removed or kept together. These custom levels are linked and incrementally remove more mutation operators, such that more performance is gained at the cost of more effectiveness with further levels.

Finally, the effectiveness and performance of a mutation level are determined by measuring the decrease in minimal test suite size ($\mathcal{E}_L$), and number of test case executions saved ($\mathcal{P}_L$) respectively, which provides the answer to Sub-question 5.

For an answer to the main research question, "*How to partition a set of mutation operators into mutation levels such that these levels balance performance with resolution?*", the process described in Chapters 4 to 6 can be followed. First the resolution and performance impact of mutation operators is determined with Callisto, using the workflow in Figure 4.1. For this a set of example programs is used which should meet the criteria as described in Section 5.2. Then with the results of Callisto mutation levels can be designed by establishing thresholds or hand-picking mutation operators based on their quality and performance impact, as demonstrated in Section 6.2. Finally Callisto can be used to test the effectiveness and performance of designed mutation levels using the metrics of Section 6.1.

For the mutation levels designed in Chapter 6, Custom 1 and 2 provide an adequate solution for StrykerJS. These two levels have demonstrated a decent performance and effectiveness, where Custom 1 retains 69% of the test suite while removing 49% of test case executions, and Custom 3 retains 48% of the test suite while removing 71% of test case executions. Other mutation levels did better, but these two are chosen as they use a consistent set of mutation operators, which ensures a consistent mutation testing experience, as explained in Section 6.2. Multiple levels are chosen as this allows users of Stryker the option to choose how much resolution they wish to lose to gain performance. Of course the third option of using all mutation operators is always available. This will guarantee the highest resolution possible, as using the mutants generated by all operators has the highest potential to lead to the design of new test cases. This conversely also has the worst performance compared to any mutation level. However, should a user of Stryker disregard

performance, for example because the program they are mutating is quite small or time is not a factor, then using all mutation operators is the best choice.

As was mentioned in Section 1.3, the set of mutation operators that Stryker uses is always subject to change. Consequently, the mutation levels should evolve with these changes, and therefore the mutation levels as designed right now in this project are temporary. This is where Callisto shows its usefulness, as it can be (re)used to determine the resolution and performance of new mutation operators to facilitate the design of mutation levels in the future as well. Furthermore, Callisto itself is language-agnostic: it can calculate quality and performance impact for any language, as long as a mutation report with the required information complying to the Stryker mutation report standard is given. This enables the possibility for the design of mutation levels for Stryker.NET and Stryker4s[1], and even other mutation testing frameworks as well. For example, PIT for Java [4] has the capability to generate Stryker mutation reports, and therefore could be compatible with Callisto, given that the right kill- and coverage information is present in the report. As was mentioned at the end of Section 4.1, Callisto can also be extended to accept other forms of mutation reports, as the workflow on which Callisto is based, Figure 4.1, can easily be adapted to other frameworks. In conclusion, a more robust answer to the main research question is therefore Callisto itself.

Besides designing mutation levels, several other interesting and useful results were found during the project. First of all, a thorough analysis of several mutation reports was never done for StrykerJS. In addition to the quality and performance impact the results presented in Section 5.5 are therefore novel. An overview of how many mutants which mutation operator generates as in Figure 5.3 is new information that provides a better perspective of the applicability of mutation operators in JavaScript and/or TypeScript.

Second of all, comparing the third and fourth column in Table 5.2 provides an example of how many test case executions are caused by static mutants. Moreover, as is stated in Section 5.5 as well, a majority of 81.70% of test case executions are due to static mutants. This gives a good indication of the performance impact that static mutants have during mutation testing, and shows that in the case of the nine example programs used, Stryker could run approximately six times faster during the mutant execution phase when ignoring static mutants.

Third and last of all, Callisto has implemented the minimising of test suites in order to calculate quality. It is possible that software testers, such as users of Stryker, would like to use this directly to analyse their test suite and remove redundant test cases. When that is the case, Callisto will already have a working algorithm and implementation to minimise test suites. Callisto can then easily be expanded to provide the result of minimising directly to the user.

7.2 Discussion

One of the most influential decisions for this project was to forego the expansion of test suites to adequacy for example programs. This has allowed more and larger example programs to be used to determine the resolution and performance impact of mutation operators in StrykerJS. Section 5.3 discussed whether this decision did not harm the reliability of the results, and showed with an experiment that there is a small deviation in the quality when removing mutants. The deviation was deemed safe enough to proceed, as the benefits of being able to use more mutants outweighed the risk of the deviation. Programs such as Freecodecamp were included, despite their low mutation

[1] Once they provide the needed information in their mutation reports.

score, as they still provided a large amount of mutants due to their size.

However, when designing and evaluating the mutation levels, there was an unexpected result as some of the unintuitive mutation levels ('No ROR', 'Only4WorstPerforming', 'OnlyBlockStatement') achieved the best results. As was stated in Section 6.2, this could be explained by the use of inadequate test suites. Furthermore, any survived mutants in an example program might be harder to kill than those that were already killed, exactly because they were not killed by the initial test suite. These harder to kill mutants would be given a higher quality when killed. Therefore using adequate test suites may result in a significant increase in quality for mutation operators, especially when the current mutation score for a program is low, such that many more mutants can be killed. For these reasons one of the avenues for future work would be to gather a set of example programs with (near) adequate test suites, such that their effect can be studied. Despite this possible effect, the core methodology for calculating quality, and using it to design mutation levels is still sound.

Research sub-question 3 asked how consistent the resolution of mutation operators is over multiple programs. To answer this question Callisto calculates the mean absolute deviation in quality for each mutation operator. In Table 5.2 and Figure 5.5 it can be seen that the deviation for many mutation operators is high, especially when their quality is relatively low. In other words, the found quality of individual mutants under such mutation operators lies in a broad range. This then indicates that there could be some inconsistency in quality among the nine individual programs used. Appendix C shows the results that Callisto produced for each example program individually. Comparing the results of the same operator across different programs shows the found quality can indeed vary. For example, the BlockStatement operator achieves a quality of 0.031 in the program BigMath (412 mutants), but in Express it achieves a quality of 0.716 (143 mutants). This can be explained by looking at the differences in code between these programs. BigMath is written such that it contains many BlockStatement mutants that are easily killed, which has as result that this operator is given a low quality. In Express BlockStatement generates mutants that are harder to kill, which results in a higher quality. In conclusion, this suggests that some mutation operators by their nature generate mutants of diverse quality. This makes them harder to place in a mutation level.

In the process to calculate the quality of a mutation operator, as described by Estero-Botaro et al. [12], the test suite of an example program is minimised twice. In Figure 4.1 this can be seen in Task 7 to 9: first the initial test suite is minimised, then to calculate the quality of an operator only its mutants are selected, after which the test suite must be minimised again to account for the reduced number of mutants. Estero-Botaro et al. give no direct reason for this approach, but one advantage is that it saves significant performance when calculating quality for multiple mutation operators. By minimising the initial test suite beforehand, determining the mutation operator specific minimal test suites has become easier, as fewer test cases and thus decision variables are present at that point. However, this introduces the possibility for mutation operators to influence each other's quality. As minimising the main test suite is done with the mutants of all mutation operators, there exists the possibility that certain test cases are removed which would have been retained when just one mutation operator is considered for minimising. When the minimal test suite for that operator is then determined, such test cases cannot be selected as they have been removed already in the main minimal test suite. A different number of test cases, or test cases that kill fewer or more mutants will change the quality outcome. An example of this effect is given in Table 7.1.

	m_1	m_2	m_3	m_4
t_1	×	×		
t_2			×	×
t_3		×	×	

Table 7.1: Example kill matrix where minimising for all mutants or just m_2 and m_3 has different results.

If the test suite is minimised with regard to all mutants, then retaining t_1 and t_2, and removing t_3 is the clear solution. However, if mutants m_2 and m_3 are from the same mutation operator O, and the test suite is minimised only for O, then t_3 forms the minimal test suite, as it kills both mutants. If the workflow described above is followed, then the minimal test suite for mutation operator O will be t_1 and t_2, as t_3 was already removed when determining the main minimal test suite. Alternatively, if the mutation operator specific minimal test suite of O is determined directly from the initial test suite, then only t_3 will be retained. This difference in minimal test suites will cause a difference in quality for operator O as well.

One way to avoid this effect is thus to minimise only once for each mutation operator, directly from the initial test suite. This way every mutation operator can choose from all test cases in the initial test suite to form an operator specific minimal test suite. The downside is that Callisto will require more time for minimising, as more decision variables will be present. For small programs this performance hit will be negligible, but for large programs with several hundred test cases and thousands of mutants this can significantly impact performance. Furthermore, the difference in quality between the two strategies is quite small, as situations such as in Table 7.1 do not occur often, and minimising once or twice will frequently have near-equal results.

Threats to Validity

External Validity

One threat to external validity for the experiments of Chapters 5 and 6 is the restriction to the StrykerJS flavour of Stryker, and thus all experimental results apply to JavaScript and TypeScript only. There is no guarantee that the same mutation operators will obtain the same quality and performance impact when applied to another programming language, for example C# with Stryker.NET. However, this only concerns the experimental results of this project. As mentioned above in Section 7.1, the methodology and use of Callisto is sound for any mutation testing framework and program which can produce a suitable mutation report.

Another threat to external validity is the unreliability of some of the results in Table 5.2, caused by low mutant counts. Because the mutators of StrykerJS are split up into 59 mutation operators, several of those operators account for very few mutants, as they are rarely applied. By merging the generated mutants of multiple example programs it was hoped that enough mutants would be generated for each mutation operator to produce reliable results. However, despite the use of around 11,000 mutants, some mutation operators still have a mutant count of below ten. For example, mutating `==` and `!=` to `true` and `false` using the ConditionalExpression mutator is exceedingly rare, as those comparison operators are seldom used in JavaScript and TypeScript.

Internal Validity

One of the key elements for calculating quality is the test suite that comes with an example program. It directly determines the kill- and coverage information that Callisto uses. One threat to internal validity here is that the structure and characteristics of the initial test suite can influence the found quality and performance impact of mutation operators. The initial test suite is created by the developers of the nine example programs, therefore their skill and methodology for writing test cases influences the experimental results of this project. Callisto minimises test suites to ensure that their size does not form a bias, but there are other problems possible. For example, one test case might contain too many assertions and should have been written as two separate cases. This will influence the kill- and coverage information, and subsequently the results of Callisto. There is no guarantee that such test cases are not present in the nine example programs used in this project.

7.3 Future Work

There are multiple options for future work with this project, regarding both Callisto itself, and the experiments performed with it.

The method used for quantifying the performance impact of mutation operators can be further improved. Currently the number of test executions needed to test the generated mutants of an operator are counted. This relies on the assumption that each test execution takes an equal amount of time for each mutant. As was explained in Section 4.1, this assumption does not always hold, as test cases can have significant differences in execution time based on how they test the program. For the scope of this project the assumption is deemed valid, as no further information regarding the execution time of test cases is currently available for Callisto. However, a future project can investigate this further by looking at whether test cases that take a relatively long time to execute are commonplace or not. Alternatively an improvement can be made to Stryker to measure the time taken for executing each test case. This information can then be included in the mutation report, so that Callisto can use it as well.

Equivalent mutants do not play a significant role in this project. Because of the decision to not use adequate test suites for the nine example programs, no equivalent mutants were identified for them. It is assumed that equivalent mutants rarely occur in StrykerJS, as could be seen with the program Robobar in Section 5.3. A possibility for future work is therefore to conduct a study to find the frequency of occurrence of equivalent mutants in Stryker. This could be done using a variety of existing techniques to find equivalent mutants [25]. When done on a mutation operator basis, problematic mutation operators that generate equivalent mutants often can be identified and re-evaluated. The results of such a study could also apply to other mutation testing frameworks or languages where similar mutation operators are used.

In the related work, Section 2.3, the fault hierarchies of Kaminski et al. [22] were discussed. They provide the opportunity to generate fewer mutants using the ROR mutation operator, based on subsumption relations identified using detection conditions and logic. Future work can consist of applying this technique on Stryker to evaluate its usefulness. Lindström and Márki [24] have shown that the fault hierarchies only hold if the concerned mutant is executed only once. Stryker should therefore be augmented with the ability to count this. This can be achieved using code coverage tools.

As was explained above in Section 7.2, more suitable example programs with high mutation

scores should be found. This way the effect of (near) adequate test suites on calculating quality can be further investigated. It could also have an effect on the effectiveness calculation for mutation levels, as explained in Section 6.2.

This project has not done an in-depth analysis for the individual mutation operators present for regular expressions. As is explained in Section 4.2, regular expressions are written in their own language and their analysis was therefore outside the scope of this project. A future project could therefore investigate their mutation operators using Callisto. For this a sufficiently sized set of regular expressions should be gathered, so that enough mutants are generated for each operator. For mutation one of the flavours of Stryker can be used, so that the regular expressions are used and tested in one of those languages.

Due to the scope of the project a limited number of methods is used in Chapter 6 to design trial mutation levels for StrykerJS. An option for future work could therefore be to further investigate how the resolution and performance impact of mutation operators can be used for the design of mutation levels. This way techniques that consistently produce effective mutation levels can be identified.

Although mutation levels were designed for StrykerJS, they have not been implemented yet. Not all mutation levels present in Section 6.2 are usable, but as was said in the conclusion, Custom 1 and 2 are suitable. They can therefore be provided to the users of StrykerJS as an option to use when mutation testing. The developers of StrykerJS have expressed that they would like to use mutation levels. For this documentation should also be placed on the Stryker website, to explain the purpose of mutation levels.

As of writing, Callisto is not yet released to the public[2]. One of the first tasks is therefore to publish it. First, the open-source repository for Callisto can be hosted under the Stryker organisation on GitHub[3]. This will also include a comprehensive readme to explain the purpose of Callisto, and how to use it. Second, Callisto can be published to a package manager so that it can be easily downloaded by anyone.

Because Callisto is currently private, it lacks several practical and quality-of-life features. Callisto is quite naive when it comes to the given JSON file and assumes it is in good order. Several mistakes could be present in the supplied data however. Callisto should therefore be extended to verify the integrity of the data it is being given, and report any peculiarities to the user as warnings. Furthermore, the current list of CLI options that Callisto accepts can be improved, see Appendix A. Several options require the use of another option, without specifying this. For example, when merging JSON reports, the option `-m` is used to indicate this, but also options `-i` and `-o` need to be used to specify input reports and the output report path, respectively. This can be improved by not using an option to indicate a merge, but a command word, followed by options related to that command. Then merging with Callisto can be done by invoking `callisto merge -i input-report.json -o output-report.json`. A similar approach can be followed for determining the effectiveness and performance of a given mutation level.

The developers of Stryker would like to have the use of Callisto automated in a pipeline. Then all steps in Figure 4.1 will be done automatically. For this a set of example programs should be set up so that Stryker can run and collect the resulting JSON reports. This can make use of the example programs used in this project, Table 5.1. Then Callisto can merge the reports and subsequently use

[2]To access (the source code of) Callisto, please contact the author.

[3]Found at `https://github.com/stryker-mutator`

the merged report as input. This will allow Callisto to be more easily used when for example new mutation operators are considered for Stryker.

Another possibility for Callisto is to integrate parts of it into the HTML mutation report of Stryker. Currently there are two pages in these reports, one for the generated mutants and achieved mutation score, and one for the test cases and their coverage of mutants. A third page could be added for information regarding the used mutation operators. This could give an overview with the results of Callisto as seen in Table 5.2. This will allow users to more easily see which mutation operators have a high performance impact for their own program, so that they can choose to exclude them for a performance increase.

Once Stryker4s and Stryker.NET support the inclusion of test case information into their mutation reports, Callisto can be used to determine the resolution and performance impact of their mutation operators. This can be used in the first place to design mutation levels for these flavours of Stryker. Additionally, Stryker shares many mutation operators between its flavours: it will be interesting to see if there is a significant difference between the results for the same operator but across two languages.

References

[1] P. Ammann, M. E. Delamaro, and J. Offutt, "Establishing theoretical minimal sets of mutants," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, 2014, pp. 21–30. DOI: 10.1109/ICST.2014.13 (cit. on p. 21).

[2] Bartosz Leoniak. "BigMath." (n.d.), [Online]. Available: https://github.com/kmdrGroch/BigMath (visited on 11/2021) (cit. on pp. 40, 73).

[3] M. Cachia, M. Micallef, and C. Colombo, "Towards incremental mutation testing," *Electronic Notes in Theoretical Computer Science*, vol. 294, pp. 2–11, 2013. DOI: 10.1016/j.entcs.2013.02.012 (cit. on p. 9).

[4] H. Coles. "PIT Mutation Testing for Java." (n.d.), [Online]. Available: http://pitest.org/ (visited on 04/2021) (cit. on pp. 9, 62).

[5] Cucumber Team. "Cucumber for JavaScript." (n.d.), [Online]. Available: https://github.com/cucumber/cucumber-js (visited on 11/2021) (cit. on p. 40).

[6] P. Delgado-Pérez, L. Rose, and I. Medina-Bulo, "Coverage-based quality metric of mutation operators for test suite improvement," *Software Quality Journal*, vol. 27, pp. 823–859, 2019. DOI: 10.1007/s11219-018-9425-7 (cit. on pp. 2, 3, 16, 18, 23, 24, 27, 53, 60, 61).

[7] R. DeMillo, D. Guindi, W. McCracken, A. Offutt, and K. King, "An extended overview of the mothra software testing environment," in *Workshop on Software Testing, Verification, and Analysis*, IEEE Computer Society, 1988, pp. 142–151. DOI: 10.1109/WST.1988.5369 (cit. on pp. 8, 13, 56).

[8] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978. DOI: 10.1109/C-M.1978.218136 (cit. on p. 6).

[9] R. DeMillo, E. Krauser, and A. Mathur, "Compiler-integrated program mutation," in *Proceedings The Fifteenth Annual International Computer Software & Applications Conference*, 1991, pp. 351–356. DOI: 10.1109/CMPSAC.1991.170202 (cit. on p. 9).

[10] A. Derezinska, "Quality assessment of mutation operators dedicated for c# programs," in *2006 Sixth International Conference on Quality Software (QSIC'06)*, 2006, pp. 227–234. DOI: 10.1109/QSIC.2006.51 (cit. on pp. 11, 14, 15).

[11] A. Estero-Botaro, F. Palomo-Lozano, and I. Medina-Bulo, "Quantitative evaluation of mutation operators for ws-bpel compositions," in *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, 2010, pp. 142–150. DOI: 10.1109/ICSTW.2010.36 (cit. on pp. 11, 14).

[12] A. Estero-Botaro, F. Palomo-Lozano, I. Medina-Bulo, J. J. Domínguez-Jiménez, and A. García-Domínguez, "Quality metrics for mutation testing with applications to ws-bpel compositions," *Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 536–571, 2015. DOI: 10.1002/stvr.1528 (cit. on pp. 2, 3, 7, 11, 15, 16, 18, 20, 21, 27, 41, 42, 60, 63).

[13] ExpressJS Team. "Express." (n.d.), [Online]. Available: https://github.com/expressjs/express (visited on 11/2021) (cit. on p. 40).

[14] Freecodecamp Team. "Freecodecamp." (n.d.), [Online]. Available: https://github.com/freeCodeCamp/freeCodeCamp (visited on 11/2021) (cit. on p. 40).

[15] Google. "Google OR-tools." (n.d.), [Online]. Available: https://developers.google.com/optimization (visited on 11/2021) (cit. on p. 33).

[16] J. Hu, N. Li, and J. Offutt, "An analysis of oo mutation operators," in *IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, 2011, pp. 334–341. DOI: 10.1109/ICSTW.2011.47 (cit. on pp. 11, 14, 15).

[17] Info Support B.V. "Info Support B.V." (n.d.), [Online]. Available: https://www.infosupport.com (visited on 04/2021) (cit. on p. 9).

[18] N. Jansen, S. de Lang, and A. van Assem. "Stryker Mutator." (n.d.), [Online]. Available: https://stryker-mutator.io (visited on 04/2021) (cit. on pp. 2, 9).

[19] ——, "Stryker Mutator - GitHub." (n.d.), [Online]. Available: https://github.com/stryker-mutator (visited on 04/2021) (cit. on p. 9).

[20] ——, "Stryker Mutator - Supported Mutators." (n.d.), [Online]. Available: https://stryker-mutator.io/docs/mutation-testing-elements/supported-mutators (visited on 04/2021) (cit. on p. 9).

[21] R. Just and F. Schweiggert, "Higher accuracy and lower run time: Efficient mutation analysis using non-redundant mutation operators," *Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 490–507, 2015. DOI: 10.1002/stvr.1561 (cit. on pp. 8, 12, 56).

[22] G. Kaminski, P. Ammann, and J. Offutt, "Improving logic-based testing," *Journal of Systems and Software*, vol. 86, no. 8, pp. 2002–2012, 2013. DOI: 10.1016/j.jss.2012.08.024 (cit. on pp. 8, 11, 56, 65).

[23] B. Kurtz, P. Ammann, M. E. Delamaro, J. Offutt, and L. Deng, "Mutant subsumption graphs," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, 2014, pp. 176–185. DOI: 10.1109/ICSTW.2014.20 (cit. on p. 12).

[24] B. Lindström and A. Márki, "On strong mutation and the theory of subsuming logic-based mutants," *Software Testing, Verification and Reliability*, vol. 29, no. 1-2, e1667, 2019. DOI: 10.1002/stvr.1667 (cit. on pp. 12, 65).

[25] L. Madeyski, W. Orzeszyna, R. Torkar, and M. Józala, "Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation," *IEEE Transactions on Software Engineering*, vol. 40, no. 1, pp. 23–42, 2014. DOI: 10.1109/TSE.2013.44 (cit. on p. 65).

[26] A. Mathur, "Performance, effectiveness, and reliability issues in software testing," *Proceedings The Fifteenth Annual International Computer Software & Applications Conference*, pp. 604–605, 1991. DOI: 10.1109/CMPSAC.1991.170248 (cit. on pp. 1, 8).

[27] E. Mresa and L. Bottaci, "Efficiency of mutation operators and selective mutation strategies: An empirical study," *Software Testing Verification and Reliability*, vol. 9, no. 4, pp. 205–232, 1999. DOI: 10.1002/(SICI)1099-1689(199912)9:4%3C205::AID-STVR186%3E3.0.CO;2-X (cit. on pp. 1, 8, 11, 13, 53, 56).

[28] Nest Team. "Nest." (n.d.), [Online]. Available: https://github.com/nestjs/nest (visited on 11/2021) (cit. on p. 40).

[29] Nikita Ovchinnikov. "Minesweeper." (n.d.), [Online]. Available: https://github.com/nickovchinnikov/minesweeper (visited on 11/2021) (cit. on p. 40).

[30] npm-stat.com. "Download statistics for package @stryker-mutator/core." (n.d.), [Online]. Available: https://npm-stat.com/charts.html?package=%40stryker-mutator%2Fcore&from=2018-01-01&to=2021-12-31 (visited on 01/2022) (cit. on p. 9).

[31] OASIS. "Web Services Business Process Execution Language Version 2.0." (2007), [Online]. Available: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html (visited on 12/2021) (cit. on p. 15).

[32] A. J. Offutt and J. Pan, "Automatically detecting equivalent mutants and infeasible paths," *Software Testing, Verification and Reliability*, vol. 7, no. 3, pp. 165–192, 1997. DOI: 10.1002/(SICI)1099-1689(199709)7:3<165::AID-STVR143>3.0.CO;2-U (cit. on p. 7).

[33] A. J. Offutt and R. H. Untch, "Mutation 2000: Uniting the orthogonal," in *Mutation Testing for the New Century*, W. E. Wong, Ed. Springer US, 2001, pp. 34–44. DOI: 10.1007/978-1-4757-5939-6_7 (cit. on p. 8).

[34] A. J. Offutt and J. M. Voas, "Subsumption of condition coverage techniques by mutation testing," 1996. [Online]. Available: https://cs.gmu.edu/media/techreports/ISSE-TR-96-01.pdf (cit. on p. 8).

[35] A. Offutt, G. Rothermel, and C. Zapf, "An experimental evaluation of selective mutation," in *Proceedings of 1993 15th International Conference on Software Engineering*, 1993, pp. 100–107. DOI: 10.1109/ICSE.1993.346062 (cit. on pp. 1, 8).

[36] F. Palomo-Lozano, A. Estero-Botaro, I. Medina-Bulo, and M. Núñez, "Test suite minimization for mutation testing of ws-bpel compositions," in *Proceedings of the Genetic and Evolutionary Computation Conference*, ser. GECCO '18, Kyoto, Japan: Association for Computing Machinery, 2018, pp. 1427–1434. DOI: 10.1145/3205455.3205533 (cit. on p. 31).

[37] A. Pizzoleto, F. Ferrari, J. Offutt, L. Fernandes, and M. Ribeiro, "A systematic literature review of techniques and metrics to reduce the cost of mutation testing," *Journal of Systems and Software*, vol. 157, p. 110 388, 2019. DOI: 10.1016/j.jss.2019.07.100 (cit. on pp. 1, 8).

[38] M. Rafalko. "PHP Mutation Testing Framework." (n.d.), [Online]. Available: https://infection.github.io/ (visited on 06/2021) (cit. on p. 11).

[39] B. Smith and L. Williams, "On guiding the augmentation of an automated test suite via mutation analysis," *Empirical Software Engineering*, vol. 14, pp. 341–369, 2008. DOI: 10.1007/s10664-008-9083-7 (cit. on pp. 11, 13).

[40] J. Smits. “Mutation Testing Elements GitHub issue: JSON Schema: Change type of testsCompleted from number to integer.” (2021), [Online]. Available: https://github.com/stryker-mutator/mutation-testing-elements/issues/1289 (visited on 01/2022) (cit. on p. 5).

[41] ——, “Mutation Testing Elements GitHub issue: Merge ConditionalExpression with EqualityOperator and move mutation from BooleanLiteral to LogicalOperator.” (2021), [Online]. Available: https://github.com/stryker-mutator/mutation-testing-elements/issues/1432 (visited on 01/2022) (cit. on p. 5).

[42] ——, “StrykerJS GitHub issue: Add configuration option to turn off bail.” (2021), [Online]. Available: https://github.com/stryker-mutator/stryker-js/issues/2996 (visited on 01/2022) (cit. on p. 5).

[43] ——, “StrykerJS GitHub issue: Json reporter: wrong test id under killedBy.” (2021), [Online]. Available: https://github.com/stryker-mutator/stryker-js/issues/3168 (visited on 01/2022) (cit. on p. 5).

[44] Stryker Team. “Mutation Testing Elements.” (n.d.), [Online]. Available: https://github.com/stryker-mutator/mutation-testing-elements/tree/master/packages/elements (visited on 11/2021) (cit. on p. 40).

[45] ——, “Robobar: How code coverage of 100% could mean only 60% is tested.” (n.d.), [Online]. Available: https://stryker-mutator.io/docs/General/example (visited on 11/2021) (cit. on p. 41).

[46] ——, “StrykerJS Core.” (n.d.), [Online]. Available: https://github.com/stryker-mutator/stryker-js/tree/master/packages/core (visited on 11/2021) (cit. on pp. 40, 73).

[47] ——, “StrykerJS Instrumenter.” (n.d.), [Online]. Available: https://github.com/stryker-mutator/stryker-js/tree/master/packages/instrumenter (visited on 11/2021) (cit. on pp. 40, 73).

[48] R. H. Untch, A. J. Offutt, and M. J. Harrold, “Mutation analysis using mutant schemata,” *SIGSOFT Software Engineering Notes*, vol. 18, no. 3, pp. 139–148, 1993. DOI: 10.1145/174146.154265 (cit. on p. 9).

A. Callisto Usage

This appendix gives an overview of the usage of Callisto. Table A.1 shows all the command-line options that Callisto can be configured with.

Name	Option	Arguments	Description
Input	`-i`	Path(s) to JSON reports to be used	Specify which JSON reports are to be used in an operation.
Output	`-o`	Path to output file	Specify where to store the results of an operation.
Merge	`-m`	-	Specify that Callisto will merge the input reports and store the result in the output file.
Level	`-l`	File containing a mutation level	Specify that Callisto will determine the effectiveness and performance of the given level, using the input JSON files as example program. Result is stored in the output file.
Killed only	`-k`	-	Only use killed mutants in the operation, for when inadequate test suites are used.
Static	`-t`	-	Include static mutants in the operation
Solver	`-s`	Name of the solver	Specify which solver to use when minimising test suites.
Verbose	`-v`	-	Enable detailed status updates during an operation.

Table A.1: Overview of all the command-line options that Callisto can be used with.

Callisto has three different modes of operation. By default Callisto will calculate the quality (and its deviation) and count the test case executions for each mutation operator used in the given example program(s), and save the result to the output file. When the merge option is used, it will merge the input JSON files into one and store the result in the output file. When the level option is used, Callisto will determine the effectiveness and performance of a given mutation level, according to the metrics in Section 6.1. The level is described in a simple text file, which contains all the mutation operators present in the level on separate lines. The calculated results are stored in a small text report in the output file. Therefore the options `-m` and `-l` cannot be used simultaneously. Furthermore, options `-k`, `-t` and `-s` are only relevant when not merging reports using `-m`.

B. Deviation Experiment Tables

This appendix shows the results of the deviation experiment described in Section 5.3 per program. Tables B.1, B.2 and B.3 show what the deviation in quality is when removing 10%, 20% or 30% of mutants using different RNG seeds for the programs BigMath [2], StrykerJS Core [46] and StrykerJs Instrumenter [47], respectively. Note that each value in the tables is already an average of the deviations in quality of individual mutators for that particular RNG seed and removal percentage.

RNG Seed	10% Removed	20% Removed	30% Removed
1	0.024	0.033	0.047
2	0.018	0.022	0.039
3	0.020	0.031	0.044
4	0.013	0.025	0.030
Average	0.019	0.028	0.040

Table B.1: Deviations in quality when removing mutants from BigMath.

RNG Seed	10% Removed	20% Removed	30% Removed
1	0.019	0.025	0.034
2	0.017	0.025	0.035
3	0.021	0.029	0.036
4	0.017	0.034	0.040
Average	0.018	0.028	0.036

Table B.2: Deviations in quality when removing mutants from StrykerJS Core.

RNG Seed	10% Removed	20% Removed	30% Removed
1	0.020	0.031	0.050
2	0.028	0.042	0.060
3	0.021	0.046	0.048
4	0.013	0.048	0.056
Average	0.021	0.042	0.054

Table B.3: Deviations in quality when removing mutants from StrykerJS Instrumenter.

C. Callisto Individual Program Results

This appendix shows the results that Callisto produced for each of the nine example programs from Table 5.1. Here the mutant count, quality and quality deviation were determined using non-static mutants only. The test execution counts do include static mutants. Note that these results cannot be directly combined to form the results of Table 5.2. The results there are calculated from the merged JSON report of all nine example programs, and therefore the found quality was determined using all mutants of an operator across all those programs. The results here are isolated from other programs. The quality of an operator is influenced by the number of mutants used for the calculation, and thus the results here cannot be averaged to find the results of Table 5.2.

Several results show a mutant count of 0. This occurs when all the mutants a mutation operator generated were not usable to calculate quality with, i.e. they were not killed, or static mutants. The result is still shown, as test executions could be counted for those mutation operators, since that is done for survived or static mutants as well.

The tables do not show the results for the complete set of 59 mutation operators, as not every operator could be applied to every example program. Furthermore, operator results where no mutants *or* test executions were counted have been removed as well. This occurs when a mutation operator generates mutants that cannot be tested, for example because they are not covered by any test cases, or caused an error. They therefore cannot be used to calculate quality, and also did not cause any test case executions, making them irrelevant for the design of mutation levels.

Mutation Operator	Count	Test Executions	Quality	Quality Deviation
ArithmeticOperator%To*	10	407	0.906	0.036
ArithmeticOperator*To/	24	392	0.543	0.418
ArithmeticOperator/To*	10	294	0.718	0.144
ArithmeticOperator+To-	16	567	0.756	0.189
ArithmeticOperator-To+	20	991	0.238	0.358
ArrayDeclarationEmpty	3	11	0.667	0.000
ArrowFunction	0	8,370	0.000	0.000
BlockStatement	412	4,652	0.031	0.049
BooleanLiteralfalseTotrue	35	3,741	0.643	0.354
BooleanLiteralRemoveNegation	16	70	0.926	0.022
BooleanLiteraltrueTofalse	14	684	0.688	0.310

ConditionalExpression!==Tofalse	18	869	0.525	0.363
ConditionalExpression!==Totrue	13	370	0.406	0.311
ConditionalExpression<=Tofalse	8	57	0.750	0.094
ConditionalExpression<=Totrue	7	29	0.694	0.105
ConditionalExpression<Tofalse	15	1,154	0.800	0.132
ConditionalExpression<Totrue	13	897	0.563	0.317
ConditionalExpression===Tofalse	58	1,503	0.728	0.261
ConditionalExpression===Totrue	58	896	0.376	0.330
ConditionalExpression>=Tofalse	3	126	0.667	0.000
ConditionalExpression>=Totrue	3	126	0.667	0.000
ConditionalExpression>Tofalse	18	1,168	0.664	0.262
ConditionalExpression>Totrue	20	1,168	0.378	0.287
ConditionalExpressionConditionTofalse	32	1,090	0.708	0.252
ConditionalExpressionConditionTotrue	30	821	0.345	0.386
ConditionalExpressionEmptyCase	6	946	0.333	0.333
EqualityOperator!==To===	9	341	0.709	0.135
EqualityOperator<=To<	5	41	0.800	0.000
EqualityOperator<=To>	5	20	0.800	0.000
EqualityOperator<To<=	11	1,002	0.780	0.137
EqualityOperator<To>=	15	798	0.588	0.350
EqualityOperator===To!==	45	635	0.628	0.336
EqualityOperator==To!=	5	24	0.720	0.096
EqualityOperator>=To<	3	126	0.667	0.000
EqualityOperator>=To>	3	126	0.667	0.000
EqualityOperator>To<=	19	1,162	0.297	0.372
EqualityOperator>To>=	11	1,181	0.794	0.137
LogicalOperator&&To||	10	84	0.788	0.115
LogicalOperator||To&&	14	191	0.898	0.035
ObjectLiteral	70	7,411	0.313	0.349
StringLiteralEmpty	94	4,095	0.434	0.126
StringLiteralFill	6	554	0.333	0.222
UnaryOperator+To-	8	163	0.844	0.047
UnaryOperatorRemove~	1	1	0.000	0.000
UnaryOperator-To+	26	3,787	0.220	0.218
UpdateOperatorPost++To--	4	47	0.750	0.000

Table C.1: Callisto results for BigMath.

Mutation Operator	**Count**	**Test Executions**	**Quality**	**Quality Deviation**
ArithmeticOperator%To*	1	12	0.000	0.000
ArithmeticOperator*To/	3	26	0.444	0.148
ArithmeticOperator/To*	1	2	0.000	0.000
ArithmeticOperator+To-	9	226	0.328	0.381
ArithmeticOperator-To+	5	125	0.790	0.080
ArrayDeclarationEmpty	26	6,548	0.474	0.405
ArrayDeclarationFill	29	1,287	0.560	0.257
ArrowFunction	73	4,215	0.606	0.244

BlockStatement	415	15,900	0.367	0.214
BooleanLiteralfalseTotrue	9	346	0.099	0.176
BooleanLiteralRemoveNegation	9	439	0.892	0.032
BooleanLiteraltrueTofalse	7	280	0.607	0.173
ConditionalExpression!==Tofalse	10	599	0.818	0.055
ConditionalExpression!==Totrue	12	594	0.665	0.255
ConditionalExpression!=Tofalse	5	381	0.320	0.192
ConditionalExpression!=Totrue	3	381	0.750	0.000
ConditionalExpression<Tofalse	4	170	0.458	0.146
ConditionalExpression<Totrue	2	93	0.000	0.000
ConditionalExpression===Tofalse	73	1,735	0.642	0.218
ConditionalExpression===Totrue	55	4,071	0.704	0.249
ConditionalExpression==Tofalse	3	31	0.667	0.000
ConditionalExpression==Totrue	1	297	0.000	0.000
ConditionalExpression>=Tofalse	1	12	0.000	0.000
ConditionalExpression>=Totrue	2	18	0.500	0.000
ConditionalExpression>Tofalse	15	233	0.632	0.274
ConditionalExpression>Totrue	14	233	0.755	0.136
ConditionalExpressionConditionTofalse	103	4,460	0.746	0.167
ConditionalExpressionConditionTotrue	99	3,611	0.529	0.292
ConditionalExpressionEmptyCase	17	143	0.581	0.377
EqualityOperator!==To===	11	407	0.824	0.127
EqualityOperator!=To==	7	489	0.095	0.163
EqualityOperator<To<=	1	93	0.000	0.000
EqualityOperator<To>=	3	93	0.333	0.222
EqualityOperator===To!==	60	1,945	0.530	0.315
EqualityOperator==To!=	3	31	0.667	0.000
EqualityOperator>=To<	1	12	0.000	0.000
EqualityOperator>=To>	0	16	0.000	0.000
EqualityOperator>To<=	14	210	0.451	0.407
EqualityOperator>To>=	10	210	0.763	0.121
LogicalOperator&&To||	15	547	0.623	0.264
LogicalOperator||To&&	15	275	0.789	0.141
ObjectLiteral	157	8,926	0.432	0.248
OptionalChaining	1	16	0.000	0.000
Regex	3	2,216	0.444	0.148
StringLiteralEmpty	201	15,284	0.502	0.280
StringLiteralFill	49	727	0.299	0.304
UnaryOperator-To+	2	2	0.500	0.000
UpdateOperatorPost++To--	0	8	0.000	0.000

Table C.2: Callisto results for CucumberJS.

Mutation Operator	**Count**	**Test Executions**	**Quality**	**Quality Deviation**
ArithmeticOperator+To-	6	239	0.550	0.272
ArithmeticOperator-To+	1	71	0.000	0.000
ArrayDeclarationEmpty	2	1,157	0.500	0.000

ArrayDeclarationEmptyConstructor	1	11	0.000	0.000
ArrayDeclarationFill	2	2,135	0.667	0.000
BlockStatement	143	32,830	0.716	0.186
BooleanLiteralfalseTotrue	8	6,185	0.833	0.056
BooleanLiteralRemoveNegation	31	5,374	0.705	0.204
BooleanLiteraltrueTofalse	2	16,249	0.500	0.000
ConditionalExpression!==Tofalse	25	16,219	0.883	0.102
ConditionalExpression!==Totrue	28	10,417	0.773	0.160
ConditionalExpression!=Tofalse	4	15	0.855	0.037
ConditionalExpression!=Totrue	4	15	0.688	0.094
ConditionalExpression<Tofalse	4	4,447	0.625	0.125
ConditionalExpression<Totrue	3	1,953	0.889	0.000
ConditionalExpression===Tofalse	45	23,703	0.847	0.109
ConditionalExpression===Totrue	45	22,985	0.771	0.158
ConditionalExpression==Tofalse	2	1,077	0.500	0.000
ConditionalExpression==Totrue	2	93	0.500	0.000
ConditionalExpression>=Tofalse	5	2,195	0.551	0.140
ConditionalExpression>=Totrue	4	1,211	0.688	0.094
ConditionalExpression>Tofalse	3	1,234	0.444	0.148
ConditionalExpression>Totrue	2	1,234	0.500	0.000
ConditionalExpressionConditionTofalse	68	23,483	0.927	0.059
ConditionalExpressionConditionTotrue	69	18,071	0.809	0.125
ConditionalExpressionEmptyCase	6	1,303	0.611	0.111
EqualityOperator!==To===	17	4,510	0.840	0.153
EqualityOperator!=To==	6	63	0.806	0.093
EqualityOperator<To<=	4	1,373	0.786	0.071
EqualityOperator<To>=	4	2,644	0.625	0.125
EqualityOperator===To!==	30	11,512	0.692	0.226
EqualityOperator==To!=	2	1,243	0.500	0.000
EqualityOperator>=To<	2	476	0.750	0.000
EqualityOperator>=To>	2	1,459	0.000	0.000
EqualityOperator>To<=	2	81	0.500	0.000
EqualityOperator>To>=	1	81	0.000	0.000
LogicalOperator&&To||	24	6,003	0.813	0.183
LogicalOperator||To&&	29	10,390	0.809	0.109
ObjectLiteral	6	10,599	0.889	0.030
Regex	3	9,220	0.000	0.000
StringLiteralEmpty	168	120,987	0.815	0.129
StringLiteralFill	8	2,021	0.929	0.023
UnaryOperator-To+	4	384	0.625	0.125
UpdateOperatorPost++To--	5	1,128	0.545	0.168
UpdateOperatorPre++To--	1	1	0.000	0.000

Table C.3: Callisto results for Express.

Mutation Operator	Count	Test Executions	Quality	Quality Deviation
ArithmeticOperator*To/	2	16	0.500	0.000
ArithmeticOperator/To*	2	11	0.500	0.000
ArithmeticOperator+To-	0	5	0.000	0.000
ArithmeticOperator-To+	5	19	0.320	0.192
ArrayDeclarationEmpty	3	1,117	0.667	0.000
ArrayDeclarationFill	3	337	0.667	0.000
ArrowFunction	25	5,723	0.800	0.165
BlockStatement	149	1,820	0.343	0.411
BooleanLiteralfalseTotrue	1	804	0.000	0.000
BooleanLiteralRemoveNegation	16	130	0.871	0.071
BooleanLiteraltrueTofalse	8	476	0.833	0.056
ConditionalExpression!==Tofalse	9	206	0.716	0.143
ConditionalExpression!==Totrue	4	206	0.625	0.125
ConditionalExpression<=Tofalse	0	6	0.000	0.000
ConditionalExpression<=Totrue	0	4	0.000	0.000
ConditionalExpression<Tofalse	2	30	0.000	0.000
ConditionalExpression<Totrue	2	18	0.500	0.000
ConditionalExpression===Tofalse	25	354	0.708	0.231
ConditionalExpression===Totrue	26	354	0.903	0.068
ConditionalExpression>=Tofalse	1	5	0.000	0.000
ConditionalExpression>=Totrue	1	5	0.000	0.000
ConditionalExpression>Tofalse	12	68	0.803	0.092
ConditionalExpression>Totrue	14	68	0.491	0.421
ConditionalExpressionConditionTofalse	33	414	0.815	0.150
ConditionalExpressionConditionTotrue	34	394	0.731	0.262
ConditionalExpressionEmptyCase	1	12	0.000	0.000
EqualityOperator!==To===	9	144	0.765	0.066
EqualityOperator<=To<	0	4	0.000	0.000
EqualityOperator<=To>	0	4	0.000	0.000
EqualityOperator<To<=	1	17	0.000	0.000
EqualityOperator<To>=	2	17	0.500	0.000
EqualityOperator===To!==	21	254	0.909	0.030
EqualityOperator==To!=	2	20	0.000	0.000
EqualityOperator>=To<	1	5	0.000	0.000
EqualityOperator>=To>	1	5	0.000	0.000
EqualityOperator>To<=	7	26	0.816	0.058
EqualityOperator>To>=	3	26	0.444	0.148
LogicalOperator&&To||	22	311	0.655	0.284
LogicalOperator??To&&	1	8	0.000	0.000
LogicalOperator||To&&	10	104	0.840	0.048
ObjectLiteral	20	1,920	0.856	0.091
OptionalChaining	3	243	0.667	0.000
Regex	18	288	0.503	0.391
StringLiteralEmpty	211	4,891	0.281	0.340
StringLiteralFill	9	311	0.741	0.165
UpdateOperatorPost++To--	1	14	0.000	0.000
UpdateOperatorPost--To++	1	10	0.000	0.000

Table C.4: Callisto results for Freecodecamp.

Mutation Operator	Count	Test Executions	Quality	Quality Deviation
ArithmeticOperator*To/	4	50	0.000	0.000
ArithmeticOperator/To*	3	38	0.583	0.111
ArithmeticOperator+To-	7	180	0.122	0.210
ArithmeticOperator-To+	9	245	0.766	0.035
ArrayDeclarationEmpty	33	730	0.767	0.138
ArrayDeclarationEmptyConstructor	2	30	0.000	0.000
ArrayDeclarationFill	0	20	0.000	0.000
ArrowFunction	15	1,048	0.887	0.065
BlockStatement	52	862	0.487	0.221
BooleanLiteralfalseTotrue	8	221	0.677	0.212
BooleanLiteralRemoveNegation	3	26	0.800	0.000
BooleanLiteraltrueTofalse	2	36	0.500	0.000
ConditionalExpression!==Tofalse	1	3	0.000	0.000
ConditionalExpression!==Totrue	1	3	0.000	0.000
ConditionalExpression<Tofalse	5	68	0.627	0.152
ConditionalExpression<Totrue	3	44	0.444	0.148
ConditionalExpression===Tofalse	18	303	0.546	0.364
ConditionalExpression===Totrue	18	303	0.468	0.312
ConditionalExpression>=Tofalse	5	170	0.000	0.000
ConditionalExpression>=Totrue	5	170	0.600	0.080
ConditionalExpression>Tofalse	8	116	0.829	0.059
ConditionalExpression>Totrue	8	116	0.590	0.244
ConditionalExpressionConditionTofalse	11	206	0.883	0.048
ConditionalExpressionConditionTotrue	11	206	0.649	0.157
ConditionalExpressionEmptyCase	8	233	0.234	0.293
EqualityOperator!==To===	1	3	0.000	0.000
EqualityOperator<To<=	4	56	0.829	0.029
EqualityOperator<To>=	4	56	0.375	0.188
EqualityOperator===To!==	12	210	0.491	0.234
EqualityOperator==To!=	1	3	0.000	0.000
EqualityOperator>=To<	2	68	0.000	0.000
EqualityOperator>=To>	2	68	0.000	0.000
EqualityOperator>To<=	3	76	0.000	0.000
EqualityOperator>To>=	3	76	0.467	0.178
LogicalOperator&&To||	14	214	0.425	0.373
LogicalOperator??To&&	1	42	0.000	0.000
LogicalOperator||To&&	1	13	0.000	0.000
ObjectLiteral	6	362	0.444	0.148
StringLiteralEmpty	17	1,563	0.261	0.328
UpdateOperatorPost++To--	3	66	0.000	0.000
UpdateOperatorPost--To++	2	22	0.000	0.000

Table C.5: Callisto results for Minesweeper.

Mutation Operator	Count	Test Executions	Quality	Quality Deviation
ArithmeticOperator+To-	1	68	0.000	0.000
ArithmeticOperator-To+	4	9	0.750	0.000
ArrayDeclarationEmpty	1	3,719	0.000	0.000
ArrayDeclarationFill	4	137	0.700	0.100
ArrowFunction	33	828	0.849	0.093
BlockStatement	119	2,156	0.838	0.151
BooleanLiteralfalseTotrue	6	162	0.875	0.000
BooleanLiteralRemoveNegation	13	231	0.908	0.042
BooleanLiteraltrueTofalse	6	139	0.880	0.015
ConditionalExpression<=Tofalse	2	20	0.000	0.000
ConditionalExpression<=Totrue	1	20	0.000	0.000
ConditionalExpression<Tofalse	0	5	0.000	0.000
ConditionalExpression===Tofalse	21	571	0.716	0.191
ConditionalExpression===Totrue	20	589	0.711	0.266
ConditionalExpression==Tofalse	1	27	0.000	0.000
ConditionalExpression==Totrue	1	27	0.000	0.000
ConditionalExpression>=Tofalse	4	81	0.000	0.000
ConditionalExpression>=Totrue	4	81	0.000	0.000
ConditionalExpression>Tofalse	11	256	0.862	0.099
ConditionalExpression>Totrue	9	241	0.296	0.395
ConditionalExpressionConditionTofalse	22	614	0.831	0.143
ConditionalExpressionConditionTotrue	22	708	0.825	0.141
ConditionalExpressionEmptyCase	27	1,049	0.756	0.288
EqualityOperator<=To<	1	10	0.000	0.000
EqualityOperator<=To>	1	10	0.000	0.000
EqualityOperator<To<=	0	5	0.000	0.000
EqualityOperator<To>=	0	5	0.000	0.000
EqualityOperator===To!==	21	423	0.706	0.189
EqualityOperator==To!=	1	27	0.000	0.000
EqualityOperator>=To<	2	33	0.000	0.000
EqualityOperator>=To>	2	33	0.667	0.000
EqualityOperator>To<=	11	194	0.413	0.451
EqualityOperator>To>=	7	156	0.122	0.210
LogicalOperator&&To||	8	266	0.787	0.148
LogicalOperator??To&&	1	34	0.000	0.000
LogicalOperator||To&&	10	284	0.764	0.153
ObjectLiteral	1	75	0.000	0.000
OptionalChaining	0	23	0.000	0.000
StringLiteralEmpty	161	10,722	0.880	0.107
StringLiteralFill	7	353	0.661	0.209
UnaryOperator-To+	1	20	0.000	0.000
UpdateOperatorPost++To--	2	37	0.000	0.000

Table C.6: Callisto results for Mutation Testing Elements.

Mutation Operator	Count	Test Executions	Quality	Quality Deviation
ArithmeticOperator%To*	1	2	0.000	0.000
ArithmeticOperator*To/	0	5,392	0.000	0.000
ArithmeticOperator+To-	6	1,497	0.778	0.074
ArithmeticOperator-To+	11	57	0.793	0.114
ArrayDeclarationEmpty	53	34,085	0.929	0.087
ArrayDeclarationEmptyConstructor	0	1,352	0.000	0.000
ArrayDeclarationFill	37	14,042	0.868	0.151
ArrowFunction	167	68,506	0.825	0.234
BlockStatement	1,197	103,614	0.605	0.279
BooleanLiteralfalseTotrue	23	11,150	0.951	0.012
BooleanLiteralRemoveNegation	253	15,503	0.689	0.299
BooleanLiteraltrueTofalse	33	11,181	0.933	0.067
ConditionalExpression!==Tofalse	37	284	0.792	0.250
ConditionalExpression!==Totrue	28	304	0.803	0.175
ConditionalExpression<=Tofalse	7	51	0.900	0.000
ConditionalExpression<=Totrue	4	51	0.625	0.125
ConditionalExpression<Tofalse	2	6	0.500	0.000
ConditionalExpression===Tofalse	120	10,710	0.647	0.382
ConditionalExpression===Totrue	123	12,043	0.635	0.361
ConditionalExpression==Totrue	1	20	0.000	0.000
ConditionalExpression>=Tofalse	4	16	0.625	0.125
ConditionalExpression>=Totrue	5	16	0.720	0.096
ConditionalExpression>Tofalse	21	2,961	0.877	0.093
ConditionalExpression>Totrue	19	13,752	0.935	0.022
ConditionalExpressionConditionTofalse	365	60,401	0.786	0.206
ConditionalExpressionConditionTotrue	442	49,570	0.717	0.248
ConditionalExpressionEmptyCase	47	102	0.791	0.303
EqualityOperator!==To===	28	156	0.764	0.292
EqualityOperator!=To==	1	2	0.000	0.000
EqualityOperator<=To<	7	51	0.900	0.000
EqualityOperator<=To>	8	51	0.875	0.037
EqualityOperator<To<=	1	2	0.000	0.000
EqualityOperator<To>=	2	6	0.500	0.000
EqualityOperator===To!==	93	7,450	0.738	0.290
EqualityOperator==To!=	2	12	0.500	0.000
EqualityOperator>=To<	4	12	0.750	0.000
EqualityOperator>=To>	3	12	0.667	0.000
EqualityOperator>To<=	9	68	0.864	0.038
EqualityOperator>To>=	4	75	0.625	0.125
LogicalOperator&&To||	144	22,065	0.813	0.190
LogicalOperator??To&&	11	1,495	0.909	0.023
LogicalOperator||To&&	132	23,486	0.783	0.244
ObjectLiteral	153	76,484	0.852	0.216
OptionalChaining	18	1,834	0.925	0.031
Regex	12	72,918	0.781	0.069
StringLiteralEmpty	160	396,086	0.603	0.413
StringLiteralFill	14	302	0.910	0.036

UnaryOperator+To-	2	4	0.500	0.000
UnaryOperator-To+	4	10,816	0.750	0.000
UpdateOperatorPost++To--	2	9	0.500	0.000
UpdateOperatorPre++To--	0	1	0.000	0.000

Table C.7: Callisto results for Nest.

Mutation Operator	Count	Test Executions	Quality	Quality Deviation
ArithmeticOperator%To*	2	19	0.500	0.000
ArithmeticOperator*To/	6	65	0.762	0.063
ArithmeticOperator/To*	8	722	0.611	0.194
ArithmeticOperator+To-	16	183	0.697	0.228
ArithmeticOperator-To+	14	163	0.815	0.150
ArrayDeclarationEmpty	23	8,356	0.888	0.104
ArrayDeclarationEmptyConstructor	2	40	0.667	0.000
ArrayDeclarationFill	7	1,380	0.816	0.058
ArrowFunction	46	1,375	0.820	0.213
BlockStatement	319	4,989	0.569	0.309
BooleanLiteralfalseTotrue	6	641	0.833	0.000
BooleanLiteralRemoveNegation	36	429	0.914	0.064
BooleanLiteraltrueTofalse	13	2,678	0.889	0.073
ConditionalExpression!==Tofalse	5	106	0.833	0.000
ConditionalExpression!==Totrue	6	119	0.810	0.063
ConditionalExpression<Tofalse	10	124	0.903	0.028
ConditionalExpression<Totrue	9	123	0.747	0.175
ConditionalExpression===Tofalse	53	855	0.823	0.178
ConditionalExpression===Totrue	43	854	0.861	0.151
ConditionalExpression>=Tofalse	2	47	0.000	0.000
ConditionalExpression>=Totrue	1	47	0.000	0.000
ConditionalExpression>Tofalse	26	365	0.840	0.151
ConditionalExpression>Totrue	19	339	0.868	0.099
ConditionalExpressionConditionTofalse	69	2,453	0.886	0.123
ConditionalExpressionConditionTotrue	63	1,852	0.818	0.225
ConditionalExpressionEmptyCase	16	82	0.877	0.101
EqualityOperator!==To===	4	194	0.575	0.188
EqualityOperator!=To==	2	19	0.500	0.000
EqualityOperator<To<=	5	121	0.800	0.000
EqualityOperator<To>=	9	121	0.778	0.173
EqualityOperator===To!==	49	746	0.815	0.207
EqualityOperator==To!=	1	9	0.000	0.000
EqualityOperator>=To<	1	42	0.000	0.000
EqualityOperator>=To>	1	42	0.000	0.000
EqualityOperator>To<=	23	297	0.897	0.034
EqualityOperator>To>=	14	297	0.916	0.029
LogicalOperator&&To||	15	274	0.908	0.074
LogicalOperator??To&&	10	69	0.860	0.048
LogicalOperator||To&&	10	126	0.643	0.235

ObjectLiteral	28	7,067	0.881	0.115
OptionalChaining	1	103	0.000	0.000
Regex	0	7,440	0.000	0.000
StringLiteralEmpty	351	76,931	0.687	0.303
StringLiteralFill	12	847	0.826	0.096
UnaryOperator-To+	3	27	0.667	0.000
UpdateOperatorPost++To--	5	45	0.667	0.133
UpdateOperatorPost--To++	1	1	0.000	0.000

Table C.8: Callisto results for StrykerJS Core.

Mutation Operator	**Count**	**Test Executions**	**Quality**	**Quality Deviation**
ArithmeticOperator+To-	9	80	0.716	0.143
ArithmeticOperator-To+	4	28	0.800	0.000
ArrayDeclarationEmpty	20	2,495	0.622	0.282
ArrayDeclarationFill	1	2	0.000	0.000
ArrowFunction	12	443	0.525	0.231
BlockStatement	106	2,223	0.601	0.273
BooleanLiteralfalseTotrue	16	649	0.365	0.320
BooleanLiteralRemoveNegation	20	491	0.733	0.165
BooleanLiteraltrueTofalse	15	445	0.635	0.201
ConditionalExpression!==Tofalse	1	43	0.000	0.000
ConditionalExpression!==Totrue	1	43	0.000	0.000
ConditionalExpression<=Tofalse	6	40	0.000	0.000
ConditionalExpression<=Totrue	4	40	0.000	0.000
ConditionalExpression<Tofalse	2	14	0.750	0.000
ConditionalExpression<Totrue	2	14	0.000	0.000
ConditionalExpression===Tofalse	50	1,108	0.681	0.194
ConditionalExpression===Totrue	50	1,160	0.606	0.248
ConditionalExpression>=Tofalse	6	41	0.000	0.000
ConditionalExpression>=Totrue	4	41	0.438	0.188
ConditionalExpression>Tofalse	9	255	0.289	0.385
ConditionalExpression>Totrue	9	258	0.821	0.071
ConditionalExpressionConditionTofalse	52	1,742	0.831	0.117
ConditionalExpressionConditionTotrue	36	1,414	0.639	0.166
ConditionalExpressionEmptyCase	14	213	0.612	0.301
EqualityOperator!==To===	1	43	0.000	0.000
EqualityOperator<=To<	2	12	0.000	0.000
EqualityOperator<=To>	2	12	0.000	0.000
EqualityOperator<To<=	2	14	0.750	0.000
EqualityOperator<To>=	2	14	0.000	0.000
EqualityOperator===To!==	33	533	0.461	0.310
EqualityOperator>=To<	2	13	0.000	0.000
EqualityOperator>=To>	2	13	0.750	0.000
EqualityOperator>To<=	5	24	0.857	0.000
EqualityOperator>To>=	3	24	0.667	0.000
LogicalOperator&&To||	15	456	0.851	0.074

LogicalOperator??To&&	3	113	0.444	0.148
LogicalOperator||To&&	25	965	0.789	0.101
ObjectLiteral	15	915	0.688	0.184
OptionalChaining	1	58	0.000	0.000
Regex	9	5,227	0.679	0.066
StringLiteralEmpty	47	13,938	0.757	0.214
StringLiteralFill	6	75	0.611	0.111

Table C.9: Callisto results for StrykerJS Instrumenter.

D. Custom Mutation Level Contents

This appendix shows the contents of the four custom mutation levels designed in Chapter 6. Table D.1 shows a list of all the 59 mutation operators present in StrykerJS. A cross marks that they have been included in the custom mutation level corresponding to that column. Because the custom mutation levels progressively remove more mutation operators, any mutation operator not present in Custom 1 is also removed from subsequent custom levels. Any operator removed in Custom 2 is also removed from Custom 3 and 4, and so on.

Mutation Operator	Custom 1	Custom 2	Custom 3	Custom 4
ArithmeticOperator%To*	×	×	×	×
ArithmeticOperator*To/	×	×	×	×
ArithmeticOperator/To*	×	×	×	×
ArithmeticOperator+To-	×	×	×	×
ArithmeticOperator-To+	×	×	×	×
ArrayDeclarationEmpty	×	×		
ArrayDeclarationEmptyConstructor	×	×		
ArrayDeclarationFill	×	×		
ArrowFunction	×	×		
BlockStatement				
BooleanLiteralfalseTotrue	×	×	×	×
BooleanLiteralRemoveNegation	×			
BooleanLiteraltrueTofalse	×	×	×	×
ConditionalExpression!==Tofalse	×	×	×	×
ConditionalExpression!==Totrue	×	×	×	×
ConditionalExpression!=Tofalse	×	×	×	×
ConditionalExpression!=Totrue	×	×	×	×
ConditionalExpression<=Tofalse	×	×	×	×
ConditionalExpression<=Totrue	×	×	×	×
ConditionalExpression<Tofalse	×	×	×	×
ConditionalExpression<Totrue	×	×	×	×
ConditionalExpression===Tofalse				
ConditionalExpression===Totrue				
ConditionalExpression==Tofalse	×	×	×	×
ConditionalExpression==Totrue	×	×	×	×

ConditionalExpression>=Tofalse	×	×	×	×
ConditionalExpression>=Totrue	×	×	×	×
ConditionalExpression>Tofalse	×	×	×	×
ConditionalExpression>Totrue	×	×	×	×
ConditionalExpressionConditionTofalse	×			
ConditionalExpressionConditionTotrue	×			
ConditionalExpressionEmptyCase	×			
EqualityOperator!==To===	×	×	×	×
EqualityOperator!=To==	×	×	×	×
EqualityOperator<=To<	×	×	×	×
EqualityOperator<=To>	×	×	×	×
EqualityOperator<To<=	×	×	×	×
EqualityOperator<To>=	×	×	×	×
EqualityOperator===To!==				
EqualityOperator==To!=	×	×	×	×
EqualityOperator>=To<	×	×	×	×
EqualityOperator>=To>	×	×	×	×
EqualityOperator>To<=	×	×	×	×
EqualityOperator>To>=	×	×	×	×
LogicalOperator&&To||	×	×	×	
LogicalOperator??To&&	×	×	×	×
LogicalOperator||To&&	×	×	×	
ObjectLiteral				
OptionalChaining	×	×	×	×
Regex				
StringLiteralEmpty				
StringLiteralFill				
UnaryOperator+To-				
UnaryOperatorRemove~				
UnaryOperator-To+				
UpdateOperatorPost++To--	×	×	×	×
UpdateOperatorPost--To++	×	×	×	×
UpdateOperatorPre++To--	×	×	×	×
UpdateOperatorPre--To++	×	×	×	×

Table D.1: Contents of the four custom mutation levels designed in Chapter 6.