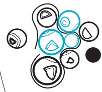
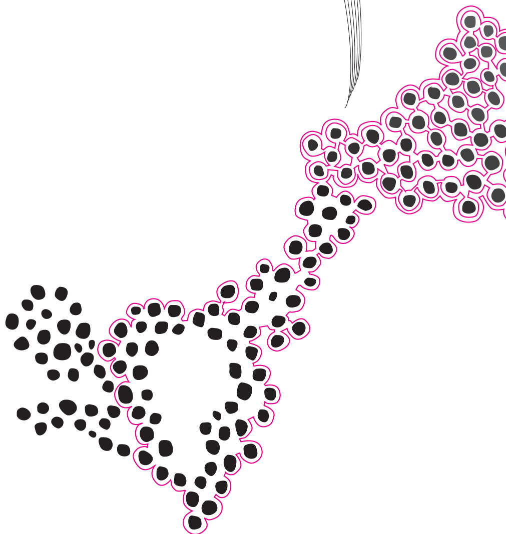
Faster Mutation Testing through Simultaneous Mutation Testing

Mart de Roos



Supervisors: Arend Rensink, Rinse van Hees

May , 2024



UNIVERSITY OF TWENTE.

Abstract

Mutation testing is a powerful fault-based testing technique, which can be used to evaluate the quality of a test suite for a specific program. This is done by injecting faults into the program, called mutations. Programs injected with mutations are called mutants and will typically behave differently from the original program, and hence (given that the original program supposedly implements the desired functionality) are faulty by construction. The idea is that an adequate test suite of high quality would be able to detect these mutants by means of a failing test, whereas a low quality test suite would not.

Unfortunately, mutation testing is costly in terms of time as it needs to run the test suite for each generated mutant. Although many optimisations already exist, most of them come with a trade-off between improving performance and reducing quality. This project attempts to improve performance while retaining quality, by testing mutants as a group, as opposed to individually, which is called *simultaneous mutation testing*. For this purpose, a theoretical view on the cost of mutation testing is presented alongside the theoretical performance benefit of using simultaneous mutation testing over regular mutation testing. Additionally, two algorithms for grouping mutants were explored, one of which is rather pragmatic while the other uses a constraint solver.

We implemented simultaneous mutation testing in StrykerJS. To validate the theory and explore the difference in performance between various groups, an experiment was set up. The experiment shows that simultaneous mutation testing can improve the performance of the mutation testing tool StrykerJS by 3%. This minor improvement can be explained by the fact that the time spent creating test sessions in StrykerJS is relatively small. The follow-up experiment, in which we increase the time spent creating test sessions, shows a performance improvement of 27%. Simultaneous mutation testing retains over 99.9% of the quality of mutation testing. Overall, the pragmatic algorithm performs better than the solver algorithm.

Contents

1 Introduction **4**

2 Related work **7**
2.1 Equivalent Mutants 7
2.2 Second-order Mutation 7
2.3 Higher-order Mutation 8
2.4 Mutant Subsumption 9
2.5 Selective Mutation 9
2.6 Test Prioritization and Reduction 10
2.7 Summary 10

3 Background **12**
3.1 Mutation Testing 12
3.1.1 Process 12
3.1.2 Equivalent and Redundant Mutants 13
3.1.3 Mutation Results 14
3.1.4 Mutation Operators 15
3.1.5 Strong And Weak Mutation Testing 15
3.1.6 Source-code and Byte-code Mutators 16
3.1.7 Cost Optimisations 16
3.2 Stryker 20
3.2.1 Mutation Operators 20
3.2.2 Mutation Results 21
3.2.3 Static Mutants 22
3.2.4 Cost Optimisations 22

4 Simultaneous Mutation Testing **24**
4.1 Concept 24
4.2 Preliminary Research 25
4.3 Research Questions 26

5 Theory **27**
5.1 Mutation Testing 27

5.2 Simultaneous Mutation Testing . . . 30
5.2.1 Performance Benefit . . . 30
5.2.2 Timeouts with Mutant Groups . . . 30
5.2.3 Amdahl's Law . . . 31
5.2.4 Smart Bail . . . 31
5.3 Runtime Specification: StrykerJS . . . 32

6 Formation of Mutant Groups **34**
6.1 Constraints . . . 34
6.2 Objective . . . 34
6.3 Concept . . . 35
6.4 Example Scenario . . . 36
6.5 Pragmatic Algorithm . . . 38
6.6 Constraint Solver Algorithm . . . 40
6.6.1 Input . . . 40
6.6.2 Axioms . . . 40
6.6.3 Variables . . . 40
6.6.4 Constraints . . . 41
6.6.5 Objective . . . 43
6.6.6 Output . . . 43
6.6.7 Implementation Particularities . . . 43

7 Implementation of Simultaneous Mutation Testing for StrykerJS **45**
7.1 Simultaneous Mutant Schemata . . . 45
7.2 Simultaneous Infinite Loop Detection . . . 46
7.3 Timeouts & Live Reporting . . . 46
7.4 Smart Bail . . . 48
7.5 Configuration Options . . . 49

8 Validation **50**
8.1 Gathering test subjects . . . 50
8.2 Evaluating performance . . . 51
8.3 Evaluating quality . . . 52
8.4 Evaluating Grouping Algorithms . . . 52

9 Experimental Setup **53**

10 Results **57**
10.1 Performance . . . 57
10.2 Quality . . . 61
10.3 Strategy for Grouping Mutants . . . 63
10.4 Summary of Findings . . . 64

11 Discussion **66**
11.1 Threats to Validity . . . 66
11.1.1 Threats to Internal Validity . . . 66

11.1.2 Threats to External Validity 67
11.2 Future Work 67
11.2.1 Timeouts and Simultaneous Testing 67
11.2.2 Simultaneous Mutation Testing for Other StrykerJS Test-runners 68
11.2.3 Smart Mutation Switching 68
11.2.4 Comparison with Stryker.NET 68
11.3 Recommendation 69
11.4 Conclusion 69

A Google Form **74**

B Automated Scripts **77**
B.1 automate-stryker.sh 77
B.2 automate-solver.sh 80
B.3 automate-both.sh 82

Chapter 1

Introduction

The typical software development lifecycle starts with an idea and then cycles through the following phases: requirements analysis & planning, design, development, testing, deployment and maintenance, as illustrated in Figure 1.1. In order to guarantee a certain level of quality within each phase, software quality policies, processes and standards are determined that describe how certain tasks should be done and how quality should be measured [21].

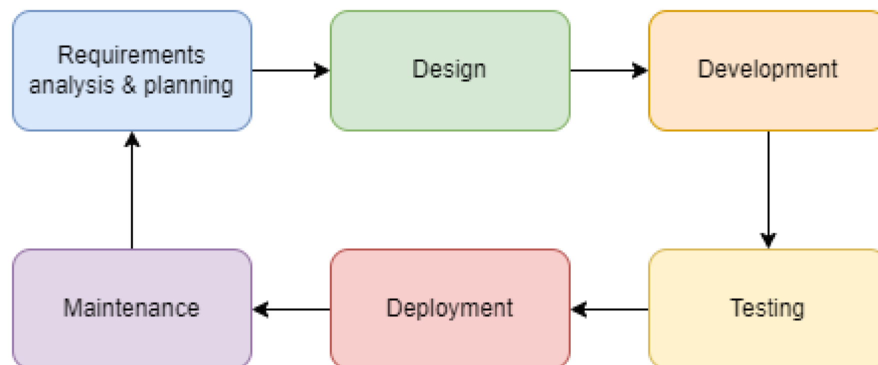


Figure 1.1: Software development lifecycle.

There exist many definitions as to what it means for software to be of good quality [7], but for the remainder of this report software quality is defined as *the degree to which a software product behaves as intended under certain conditions*, where the intended behaviour (under the specified condition, if any) is imposed by the set requirements. Conditions are a description of the state of the environment in which the system runs, such as high or low load. Of the software development lifecycle, the test phase exists to assess the correctness, and thus

the quality, of a program. The test phase may consist of numerous test suites, which are a collection of tests, and can be used to assess the quality of a piece of software. Hence, it is desirable to have high-quality test suites. Quality of a test suite is defined as *the degree to which a test suite is able to find deviations from the software's intended behavior.* Writing high-quality test suites is hard. Many testing techniques, like equivalence class partitioning and boundary value analysis [11], could be enforced in a tester's workflow in order to improve the quality of test suites. However, these techniques do not guarantee high-quality test suites as they are prone to human error.

It is impossible to determine the quality of a test suite simply by looking at it. Instead, quality is quantified through metrics derived from an execution of the test suite. The most common metrics used are method, branch, statement and line coverage [22]. However, a better but less common metric is *mutation score.* Mutation score is derived through mutation analysis, which is an extension of the test phase in order to properly assess the quality of a test suite.

Mutation testing, or *mutation analysis*, is a fault-based testing technique used to evaluate the quality of a test suite of a specified program. It does so by making small changes to the source code of the original program, called *mutations*, which may cause the program to behave differently than intended, essentially simulating a possible fault. These *mutant* programs are generated by so-called *mutation operators* which map certain syntactic tokens to others, introducing a possible fault. The newly mutated program will then be tested by the original test suite. A mutant is considered *killed* whenever at least one test fails. Conversely, that same mutant is considered to have *survived* when the mutant passes all the tests. Practitioners should consider specifying new tests whenever a mutant survives. In order to assess the quality of a test suite, a *mutation score* is calculated. The mutation score is the ratio of killed mutants to the total number of mutants generated. A high mutation score indicates that the test suite is of high quality, capable of detecting many bugs in the code, whereas low scores may indicate that the test suite is not adequate for detecting most bugs in the code. In practice, the aim is to have a mutation score of approximately 80% since it is often impossible to get a score of 100% due to *equivalent mutants*, which will be elaborated on in section 3.1.2.

Mutation score is the superior metric compared to traditional statement and branch coverage [32]. This is because the traditional metrics do not give any indication on whether the semantics or behaviour of a piece of code has been tested for correctness, it only indicates that the piece of code was executed. A surviving mutant could be an indication that the behaviour of a piece of code was not tested for correctness. To illustrate this, a (rather useless) test suite that executes about half of the source code but has no assertions of any kind will have approximately 50% statement and branch coverage. The mutation score, on the other hand, would be 0%.

Mutation testing is deemed to be one of the best strategies when it comes to program validation and assessing test quality [14]. Unfortunately, mutation

testing is also one of the most, if not *the* most, expensive form of test quality assurance compared to traditional (coverage-based) forms of test quality assurance in terms of resources. As a result, it has not been widely adopted by the industry. A project may generate thousands of mutants and running the entire test suite for every single mutant is time consuming. A project with about 7000 lines of code (which is not especially large) may easily generate 2500 mutants[1]. If it takes one second to test a single mutant on average, then it will take over 40 minutes for mutation testing to be finished. Even then, after running all mutants, time needs to be invested into analysing the surviving mutants. Consequently, a lot of research has been done with the goal being to reduce the cost of mutation testing [35].

Most of the research aiming to improve the performance of mutation testing comes with a trade-off, namely sacrificing mutation testing quality for a reduction of costs. Here, mutation testing quality means *how well the mutation score reflects the actual quality of a test suite*. Hence, sacrificing mutation testing quality is undesired as it decreases the trust one can have in a system. Moreover, it is fairly hard to quantify to what extent this sacrifice to quality will have on the overall quality of one's test suites [34] because the creation of new tests is driven by the survival of mutants.

The solution to the resource-demanding nature of mutation testing investigated in this report, called *simultaneous mutation testing*, attempts to reduce the time it takes to perform mutation testing *without loss of quality*. However, the correctness of simultaneous testing could be too complex to guarantee as it relies on imperfect coverage analysis, which will be elaborated on in chapter 4.

This report is structured in the following way. The next chapter is dedicated to related work, which is mostly about other optimisations in mutation testing (chapter 2). Then, background information is provided such as the (technical) details of mutation testing (chapter 3). Afterwards, the concept of simultaneous mutation testing is explained and includes the research questions we wish to answer (chapter 4). This is followed by an extensive theoretical view on the cost of both regular and simultaneous mutation testing (chapter 5). Then, an investigation into the formation of simultaneous mutant groups is presented (chapter 6), followed by: the implementation of simultaneous testing specifically for StrykerJS (chapter 7), how simultaneous mutation testing is evaluated (chapter 8) and describing the experimental setup (chapter 9). Finally, conclusions of the results are presented (chapter 10) followed by a discussion (chapter 11).

[1]Derived from StrykerJS's core package from Github.

Chapter 2

Related work

This chapter discusses other works that may prove useful in finding ways to reduce the cost of mutation testing. Note that the works described do not necessarily belong to only one category (indicated by the title of the subsection), it is possible that there is some overlap between categories in some of the works.

2.1 Equivalent Mutants

Kintis et al. [17] developed a method that can classify whether survived mutants are equivalent or not. They propose the *HOM Classifier* that can categorize mutants based on the impact they have on each other. The idea is that since equivalent mutants have little to no impact on the state of the program, they should also not have an effect on the state of another mutant. The classifier they have created is called I-EQM and has a precision of 71% and a recall of 82% for identifying whether mutants are equivalent and is a combination of the *Coverage Impact Classifier*, which classifies equivalence based on the number of methods that deviate from execution frequency compared to the original program's execution frequency, proposed by Schuler and Zeller [38] and the new *HOM Classifier*.

2.2 Second-order Mutation

Second-order mutants are mutants that are formed of two first-order mutants. Papadakis et al. [31] have made an empirical evaluation of first and second-order testing strategies. In this study, the strength (fault rate), cost and benefit are compared between 14 variants of first and second-order mutation strategies. The study only tests on programs that have been identified to be faulty in order to properly determine strength, being the ability to detect *real faults*. There are three strategies in particular that are interesting, namely: (i) *StrongMutation*,

which means no special strategy; (ii) *SameNode*, which combines two mutants from the same basic block to form a single mutant (2nd-order strategy); and (iii) *Rand60%*, which means that only 60% of the mutants are executed (1st-order strategy). They find that StrongMutation has the highest fault rate, as expected, and that Rand60% scores similar. For all second-order strategies, SameNode performs best. In terms of cost SameNode and all first-order strategies require many more tests than the second-order strategies. Additionally, all but one second-order strategy created 80-90% fewer equivalent mutants, the exception being SameNode with 60% fewer equivalent mutants. They conclude that first-order strategies are better at detecting faults and that second-order strategies are less costly.

Polo et al. [36] have introduced three second-order mutation strategies, namely: (i) *LastToFirst*, which combines the first and last first-order mutant (FOM) in the list of mutants to form a second-order mutant (SOM); (ii) *DifferentOperators*, which combines two FOMs from different mutation operators with each other; and (iii) *RandomMix*, which combines any two FOMs to form a SOM. They find that the number of equivalent mutants is reduced by approximately 73% and that there is a risk of 21% that killing a SOM does not kill both FOMs from which they were formed.

2.3 Higher-order Mutation

Jia and Harman [12, 13] have introduced the concept of subsuming higher-order mutants (SHOM). A HOM is considered subsuming whenever the ratio of fragility between the HOM and the FOMs from which it is created is less than 1, where fragility of a mutant is equal to the number of test cases that kill that mutant. They find that about 15% of HOMs are also subsuming. Additionally, they find that the genetic algorithm used finds the most SHOMs, that the hill climbing algorithm finds the highest fitness SHOMs and that the greedy algorithm finds the highest order SHOMs. Unfortunately, in order to find SHOMs it is necessary to analyse the mutation results, which means it is required to have performed complete mutation testing on all FOMs before one can make use of the established SHOMs.

Ghiduk et al. [6] have done a systematic literature review on higher order mutation testing. They conclude that there are three tactics for reducing the number of HOMs, namely: (i) reduce the number of mutation operators as less FOMs mean less HOMs, (ii) selecting a subset of HOMs instead of all mutants such as subtle set and (iii) reduce mutated locations in original programs using techniques such as data flow analysis. Research is still needed to quantify the realism of the generated HOMs.

2.4 Mutant Subsumption

Kurtz et al. [18] define mutant subsumption as: when all tests that kill mutant a also kill mutant b, then a subsumes b. This is represented in so-called mutant subsumption graphs (MSG). Subsumption relationships identify redundancy of subsumed mutants. Subsumed mutants need not be tested for as they are covered by the subsuming mutant, reducing the number of mutants to test. In the paper, they distinguish between dynamic and static mutant subsumption graphs (DMSG and SMSG). In a subsumption graph, nodes represent maximal sets of indistinguishable mutants and edges represent a subsumption relation in which the source subsumes the sink. Subsumption relations of DMSGs are identified by executing all mutants and analysing the results, whereas subsumption relations are identified *by hand* for SMSGs. In the example program used, they find that 7 mutants subsume all other 127 mutants generated by *muJava* [23] based on only the DMSG. They conclude that DMSGs tend to be optimistic about subsumption relationships whereas SMSGs tend to be pessimistic. Another work by Kurtz et al. [19] extends on this by automating the generation of SMSGs with directed incremental symbolic execution (DiSe). The results show that SMSG has higher precision and accuracy than DMSG but a low recall. It appears that static analysis tools are not strong enough to find subsumption relationships on their own which is unfortunate as being able to find subsumption relationships without running test cases, as with the dynamic approach, is highly desirable. Having to perform dynamic analysis in order to find subsumption relationships defeats the purpose of finding them in the first place, as the work has then already been done.

2.5 Selective Mutation

A paper from 1996 by Offutt et al. [26] presents an empirical evaluation of selective mutation for Fortran. They conclude that only five operands (ABS, AOR, LCR, ROR and UOI) out of all 22 MOTHRA [5] operands are sufficient for effective mutation testing. Although the results of this research cannot be applied to modern programming languages, it is a good indicator that selective mutation can be efficient without loss of too much quality.

A recent study by Smits [39] defines mutation levels for the JavaScript flavour of Stryker. For this, the tool Callisto was created which automates the analysis of mutation operators' quality. The output of Callisto is used to determine the quality metric *resolution*. *Resolution* is a new metric introduced in the paper that is defined as follows: "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". Resolution is then used to create mutation levels. The mutation levels were chosen by intuition and compared with each other. In order to properly compare mutation levels, effectiveness and performance are compared. Effectiveness is the proportion of the minimal test suite that is required to kill all mutants of a mutation level to the minimal test suite that

is required to kill all mutants when no mutation level is used. Performance is defined as the percentage of test case executions that were removed. Of interest are the created mutation levels *custom 1* and *custom 2*. Compared to the default set of mutation operators in StrykerJS [41], custom 1 excludes the following mutators: BlockStatement, StringLiteral, ObjectLiteral, Regex, Unary and all mutators that work on the token '==='. Custom 1 has an effectiveness of 69% and a performance of 49%. Custom 2 excludes all mutators that custom 1 excludes and additionally excludes the following mutators: ConditionalExpressionEmptyCase, ConditionalExpressionConditionTofalse, ConditionalExpressionConditionTotrue and BooleanLiteralRemoveNegation. Custom 2 has an effectiveness of 48% and removed 71% of all test case executions.

2.6 Test Prioritization and Reduction

Zhang et al. [48] introduce a technique that consists of both test prioritization and reduction, called 'Faster Mutation Testing' (FaMT). For test prioritization they have two phases, an initial and a dynamic one. The initial phase is based on coverage. They consider the odds of killing a mutant by a certain test to be higher when the test executes the mutated statement more often and when the test executes the mutated statement more closely to the test's exit statement. The dynamic phase reorders tests during mutation testing, favouring tests that tend to kill neighbours of the mutant under test. There are four levels for determining whether two mutants are considered neighbours, namely: statement-level (mutants share the statement), method-level (mutants share the method), class-level (mutants share the class) and global-level (any two mutants are considered neighbours, globally well-performing tests are prioritized). Only one level is selected during the dynamic phase. For reduction they simply halt testing when a certain percentage of tests are run. The idea is that the more powerful tests have already been executed and thus should have had a higher chance to kill them. This does mean that it is possible that we do not kill a mutant because the test that would have killed the mutant is not executed. Reduction techniques can reduce all executions for all mutants by around 50.0% while only causing error rates around 0.50%, and some FaMT reduction techniques can reduce all executions for all mutants by more than 63.0% while causing error rates smaller than 1.22%. Unfortunately, no data has been provided on improvements of total execution time. The number of test executions is not a clear indicator for more efficient mutation testing, it is possible that the initial phase favours integration tests, which tend to run longer than unit tests.

2.7 Summary

A lot of research has been put into decreasing the cost of mutation testing. Unfortunately, a decrease in cost is often paired with a decrease in quality, especially with second and higher order mutation. Additionally, a lot of the optimi-

sations can only be used after having already done the work of mutation testing (post-optimisations) in its entirety by analysing the mutation results, defeating the purpose of optimising in the first place. Furthermore, post-optimisers can mostly be replaced by incremental mutation testing which skips certain mutants completely instead of re-evaluating with a more optimised set of mutants.

Chapter 3

Background

3.1 Mutation Testing

Mutation testing was first introduced in 1978 by DeMillo et al. [4] and is a means to evaluate the effectiveness of one's test suite, and as such is a way to evaluate the behaviour of the system under test. In their paper, the empirical principle of *the coupling effect* is introduced. This principle hypothesizes that complex faults are often the cause of relatively simple faults. Consequently, the detection of complex faults is often assured by the detection of simple faults. The coupling effect is supported in the paper by the idea of the *competent programmer*, which states that programmers write programs that are close to correct, indicating that many bugs are caused by trivial syntactic mistakes.

3.1.1 Process

Mutation testing, or *mutation analysis*, is a white-box testing technique which works by injecting simple faults, or bugs, into a program, which are called mutations. The mutated programs that result from the injection are called mutants. Once all mutants have been created, the test suite will be run on the original non-mutated program to ensure that the original program actually abides to the test specification. Next, the mutants are tested by the same test suite and afterwards the test results are analysed. This process is illustrated by Figure 3.1. In the case that at least one test fails for a specific mutant, then we say that this mutant has been *killed* (or *detected*). When no tests fail for a specific mutant, we say that this mutant has *survived* and is still alive (or has gone *undetected*).

Listing 3.1 shows a Java program which finds the minimum of two integers and includes two tests. A possible mutation is to replace the less-than operator ('<') on line 2 by a greater-than operator ('>'). This injected fault changes the output of the program for many inputs and can, in this case, be caught by a

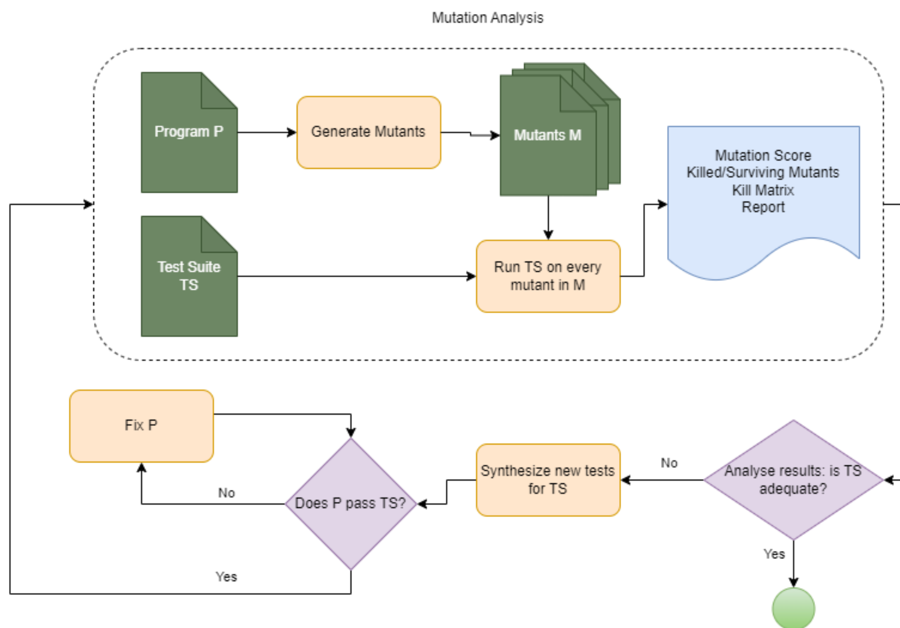


Figure 3.1: Mutation analysis workflow.

single test. For example, the test t_1 where $x = -1$ and $y = 2$ which should return -1 will fail. We say that t_1 has killed the mutant. A test case in which x is equal to y (t_2) would never be able to detect or kill this mutant.

```java
int min(int x, int y) {
    if (x < y) // mutant: if (x > y)
        return x;
    return y;
}

void testMin() {
    assertEquals(-1, min(-1, 2)); // t1: x=-1, y=2
    assertEquals( 4, min( 4, 4)); // t2: x= 4, y=4
}
```

Listing 3.1: Example Java program for finding the minimum of two numbers accompanied by a test method.

3.1.2 Equivalent and Redundant Mutants

Whenever a mutant survives, there are two possibilities. The first is that the test suite is not effective enough in order to detect this specific mutant and that new tests should be created that will kill the mutant. The second is that the mutant is

an *equivalent mutant*, meaning that the mutant behaves semantically equivalent to the original program. It is *impossible* to kill equivalent mutants and no effort should be put into creating new tests. In Listing 3.1, the mutant where '<' is replaced with '<=' is an example of such an equivalent mutant. There exists no test that can distinguish between this mutant and the original program as the output is the same for all values of x and y. The detection of equivalent mutants is in general impossible as determining whether two programs are equivalent is undecidable [27]. The proportion of equivalent mutants is hard to determine but it is estimated to be about 8.6% on average. This value is derived from the work by Schuler and Zeller [38] and is close to the 9.1% equivalence reported by Offutt and Pan [28]. Note, however, that the proportion of equivalent mutants depends heavily on the *mutation operators* (see section 3.1.4) used and thus could vary per programming language.

Some mutants are *redundant*. Redundant mutants are semantically equivalent to other mutants, but not to the original program. If two mutants are considered to be redundant with respect to each other, then it is sufficient to only kill one of them, as killing one guarantees killing the other. In Listing 3.1, the mutants where '<' is replaced by '>' and '>=' are redundant with respect to each other. The proportion of redundant mutants is expected to be similar to the proportion of equivalent mutants, so approximately 9%.

3.1.3 Mutation Results

Once all mutants have been tested against the test suite, a mutation score is calculated. The mutation score S is the percentage of mutants that were killed, which is equivalent to the the number of killed mutants over the number of non-equivalent mutants generated times a hundred percent, as shown in Equation 3.1.

$$S = \frac{\#\text{killed-mutants}}{\#\text{generated-mutants} - \#\text{equivalent-mutants}} * 100\% \tag{3.1}$$

$$S' = \frac{\#\text{killed-mutants}}{\#\text{generated-mutants}} * 100\% \tag{3.2}$$

Since determining whether a survived mutant is equivalent is undecidable, calculations of mutation score in mutation testing frameworks do not take mutant equivalence into account, as indicated by Equation 3.2. This means that the actual mutation score S might be higher than the calculated score S'. It also means that the maximum achievable mutation score (S') is probably less than a 100%.

In addition to calculating the mutation score, mutation testing frameworks often generate a *mutation report*. With mutation reports, one can find information such as the status (e.g., *survived* and *killed*) of a mutant, reason for the associated mutant status and which test(s) were responsible for killing a specific

mutant in the form of a *kill matrix*. Note, however, that the information available in mutation reports can be different between mutation testing tools.

3.1.4 Mutation Operators

Mutation operators are responsible for generating mutants. They work by removing, replacing or injecting one or more syntax tokens in a program. A common mutation operator is the Relational Operator Replacement (ROR) mutator which mutates a relational operator, which can be any of '<', '<=', '>', '>=', '==' and '! =', to all other relational operators. This means that for every relational operator used, there are five more mutants to kill. It is possible that a mutation operator makes for an *invalid mutant*. An invalid mutant is a mutant that cannot be tested, for instance a mutant that produces a compile time error, and is not included into the mutation score.

Many defined mutation operators can be applied to multiple programming languages, such as the ROR mutator since most programming languages contain relational operators. There also exist language-specific mutation operators, which only apply to a few or a single programming language and sometimes may even be tailored to the domain. An example would be a mutation operator that removes a call to 'Distinct()' or 'Unique()' on a collection, possibly resulting in a collection that contains duplicate values. Although less commonly used, there also exist interface mutation operators, which evaluate how well the interactions between two units are tested [3, 2].

3.1.5 Strong And Weak Mutation Testing

The original idea of mutation testing as defined by DeMillo et al. [4] only considered *strong mutation*. That is, a form of mutation testing where only the output of a mutant is compared to the output of the original program. In order to be able to detect a fault, it is required that the fault causes an abnormal state within the program that propagates through the entire program, affecting the final output. The output of the program can only be tested with *integration tests*. Integration tests test the behaviour of two or more modules that should work together and tend to run longer than *unit tests*.

Only four years after the inception of mutation testing, *weak mutation* testing was introduced [9]. During weak mutation, the internal program state is immediately compared to the expected program state after the execution of a mutated statement. *Weak mutants* do not necessarily propagate through the entire program. Weak mutants are killed mostly by *unit tests*, which test the individual components within a module. Weak mutation is, in general, faster than strong mutation because unit tests execute a smaller part of the entire program.

Weak mutation testing is used more often in the industry than strong mutation testing because it is faster. Hence, the remainder of this report should be viewed

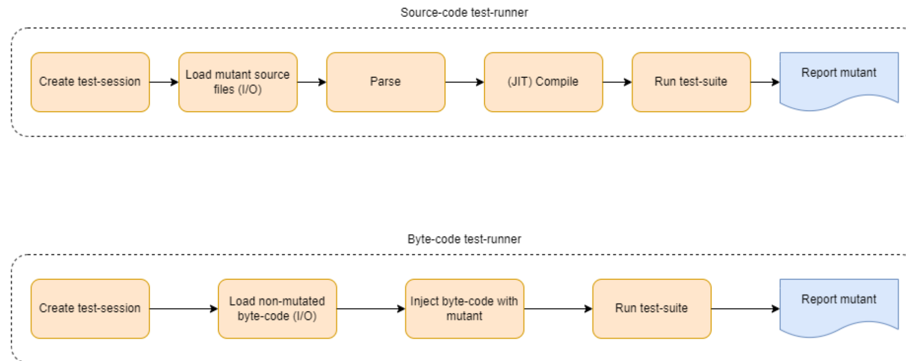


Figure 3.2: Control flow diagram of source-code and byte-code test-runners.

in context of weak mutation testing.

3.1.6 Source-code and Byte-code Mutators

There exist two ways by which mutation testing can be done. One is called a source-code mutator, the other is called byte-code mutator. The difference between the two is that a source-code mutator mutates the source files and then recompiles the mutants whereas a byte-code mutator mutates the compiled byte-code or machine code of the original program. Byte-code mutators are faster in performing mutation testing compared to non-optimised source-code mutators. On the other hand, source-code mutators tend to generate more suitable or representative mutants than byte-code mutators. This is due to the fact that compiled code may be optimised by the compiler, blurring the relation with the original source code. Even more importantly, it can be hard to explain where a mutant originates from in byte-code mutators, which makes it harder to kill surviving mutants [33]. Figure 3.2 illustrates the course of action of (non-optimised) source-code and byte-code test-runners within the action "Run TS on every mutant in M" from the mutation analysis process diagram (see Figure 3.1). From this illustration it should be clear that source-code mutators are generally slower due to the parsing and (JIT) compilation phase, which byte-code mutators do not have.

3.1.7 Cost Optimisations

As briefly mentioned in the introduction, mutation testing is a relatively costly technique compared to traditional testing. Many mutants could be derived from a single source file. It is not uncommon to have thousands of mutants in an entire project. The entire test suite needs to be run against every single mutant, which is a lengthy process. For example, if a project contains 5000 mutants and a test suite that requires 1 second to finish on average, it will take approximately 83

minutes to finish the entire mutant execution process. After the execution, the testers need to analyse the result to determine whether they need to synthesize new tests in order to kill the surviving mutants. This iterative process can also be quite time-consuming as many surviving mutants may need to be analysed for equivalence.

Consequently, a lot of research has been put into reducing the cost of mutation testing, summarized by Pizzoleto et al. [35]. Offutt et al. [29] have identified three main categories by which most research can be characterized, namely "do fewer", "do smarter" and "do faster". The following optimisations may be applied to speed up mutation testing:

- "Do fewer" tries to reduce the total number of mutants that are being generated and executed. Optimisations in this category usually come with a trade-off between cost and quality.
  - **Selective mutation** is a technique where only a subset of mutation operators is used to generate mutations. Selective mutation has varying impact on performance and quality as it is completely dependent on the selected subset of mutators. Additionally, performance and quality measurements for a specific subset of mutation operators may not be consistent between different programming languages or even domains within a programming language [39, 26].
  - **Higher-order Mutation** makes for fewer mutants by combining two or more mutations into a single higher-order mutant (HOM) [6, 12, 13, 20, 25]. Instead of executing all first-order mutants (FOMs), one could decide to execute higher-order mutants, reducing the total number of mutants to test. Preferably, the formed HOMs are representative for all the FOMs in a program but this cannot be guaranteed because any HOM may behave drastically different from the behaviours of its constituent FOMs. **Second-order Mutation** is a form of *higher-order mutation*, but it restricts itself to only form mutants from two first-order mutations [31, 16]. Many algorithms exist for creating higher-order mutants and the performance and quality differs for each. See section 2.2 for more details on some of the algorithms used.
  - **Mutant Subsumption** makes for fewer mutants by identifying subsumption relations that signify redundancy between mutants [18, 19]. The idea is that subsumed mutants need not to be tested for since testing the subsuming mutant should be sufficient. Note the use of 'should': killing the subsuming mutant does not always guarantee that the test suite also kills the subsumed mutant [15].

    Take for example the C++ program defined in Listing 3.2. The commented lines 2 and 3 are both valid mutants, respectively named m_1 and m_2, that may be generated to replace the original statement in line 4. m_1 is generated by the 'FunctionBodyRemoval' mutator,

which serves as a form of method coverage in mutation testing by emptying the body of a function. m_2 is generated by the 'Addition-Negation' mutator. In this case mutant m_2 subsumes mutant m_1 because all tests that kill m_2 are guaranteed to also kill the mutant on line m_1. In fact, any mutant within the body subsumes the mutant generated by the 'FunctionBodyRemoval' mutator.

```
1 void increment(int &i) {
2     // m1: return;
3     // m2: i = i - 1;
4     i = i + 1;
5 }
```

Listing 3.2: Example C++ function that increments an integer passed by reference with two mutants that have a subsumption relation.

Subsumption can be combined with higher-order mutation [12, 13] to automatically detect subsumption relations. For more details, see section 2.3 and 2.4.

- "Do smarter" attempts to reduce the cost by skipping certain operations through clever deductions and by splitting the workload. Optimisations in this category *usually* retain the quality of mutation testing.
  - **Parallelization** is a common technique in software where certain calculations are done in parallel in order to reduce the total time it takes to perform a certain task. Parallelization in mutation testing means to execute the generated mutants in parallel [10, 24]. This can be done locally on one's computer by creating more test runners, where every runner only executes a (proper) subset of the mutants to test. More optimal would be to allow the mutation analysis to run distributively in the cloud if such resources are available.
  - **Test Prioritization and Reduction** is a means to reduce the number of test executions for each mutant [48]. Test prioritization comes down to reordering test cases in such a way that tests that have a higher likelihood to kill a mutant are executed first. Test reduction involves executing only a subset of the entire test suite based on some rules. A trivial example of such a reduction rule is to stop testing after a certain percentage of test-cases of a test suite have been executed. This works well in combination with test prioritization. A consequence of test reduction is that the calculated mutation score is an underestimation of the actual mutation score.
  - **Mutant Coverage Analysis** is a technique where only a subset of tests is executed. Instead of executing every mutant against the entire test suite, a coverage analysis is performed in order to trace reachability of mutants by all test cases. Through this analysis, one can run only those tests that can reach a specific mutant since the

execution of tests that can never reach this mutant is useless. This allows mutation testing tools to skip most unit tests for many mutants. This technique could be considered a form of *test reduction*.

  - **Bail** or fail-fast is a technique where test runners abort execution of additional tests whenever any test has failed. Since the goal of mutation testing is to kill a mutant by failing at least 1 test, it is not necessary to continue running tests when a mutant has been killed by another test. There do exist other optimisation techniques, such as *dynamic test prioritization* where the continuation of testing may be desired in order to properly evaluate the likelihood of a test killing a specific mutant.
  - **Incremental Mutation Testing** or testing with baseline is a technique similar to traditional incremental testing but for mutations [1, 49]. Mutation testing tools will take information from a previous *mutation report* and deduce whether mutants or tests have changed. If a change has been detected for a certain mutant, it will be tested again, otherwise the results for that mutant are copied from the previous report. Note that the definition of 'change' may be different among mutation testing tools. This optimisation is a big improvement for practitioners as they now can write tests to kill mutants that previously survived without the need to wait for all mutants of which the results are already known.
  - **Automated Equivalence Detection** decreases the time spent by developers deciding whether mutants are equivalent. Deciding equivalence takes about 15 minutes per mutant [38], which can be quite time-consuming for large projects. Being able to automatically detect equivalence improves productivity. Detection of equivalence can be integrated into the compiler [30] or can be done through techniques such as second-order mutation [17]. Once equivalence has been detected for a mutant, it can then be excluded from mutation testing for future runs.

- "Do faster" comes down to reducing the time it takes to generate and execute mutants individually. Optimisations in this category should *never* impact the quality of mutation testing.
  - **Mutant Schemata**, meta-mutants or mutation-switching is a technique where all found mutants are compiled into a single program, the 'meta-mutant' [46]. The different mutants can be enabled by dynamic environment variables or flags. This technique eliminates the need to recompile the entire project for every single mutant generated. This is a significant improvement as larger projects may take a relatively long time to compile. This optimisation is really only applicable to source-code mutators, since byte-code mutators work with the compiled byte-code directly.

- **Hot Reload** is an optimisation where certain test-runners are adapted in such a way that sequentially executing multiple mutants (or programs for that matter) does not require a full restart of the environment. In Java, the initialization of the JVM, which takes about 70 milliseconds [47], dominates the total runtime for small programs, adding 70 seconds to mutation testing for every thousand mutants. For large programs, the JIT compilation may take even more time than the initialization of the JVM. With hot reload, one can eliminate startup times as well as retain JIT compiled code.

  For hot reload to take full effect, it is best that one has also implemented *mutant schemata*. Retaining JIT compiled code is near impossible when there is no mutant schemata. When a source-code mutator has both optimisations, it means that the source-code test-runner (see Figure 3.2) can skip the file I/O, parsing and (JIT) compilation phases for subsequent mutant runs. For byte-code mutators, this optimisation is relatively simple to implement since it does not require mutant schemata. With hot reload, the file I/O phase can be skipped in byte-code test-runners for subsequent mutant runs.

3.2 Stryker

Stryker is an open-source mutation testing framework that originates from a graduation project, mutation testing in JavaScript, at Info Support [40, 43]. Due to its enormous success, Info Support has been sponsoring the development for Stryker for years now and it has become a well-known mutation testing framework. Stryker is a source-code mutator and as such tends to create more realistic faults than byte-code competitors, however at the cost of some performance as previously stated in section 3.1.6. At this moment in time, Stryker consists of three 'flavours', namely: StrykerJS for JavaScript and TypeScript [42], Stryker.NET for C# [45] and Stryker4s for Scala [44].

3.2.1 Mutation Operators

Stryker uses a common set of mutation operators between their flavours in order to make it easy to switch and compare between them. Because every programming language is different, some mutators are not supported or cannot be supported at all because they are not applicable in the language. For example, the operator 'optional chaining' that mutates `foo?.bar`[1] to `foo.bar` is implemented in Stryker4JS, not implemented (yet) in Stryker.NET and not implemented in Stryker4s because Scala does not support optional chaining in their language definition (in Scala, optional chaining is supported with the `Option[T]` type). The full list of supported mutation operators (16 in total) can be found on the Stryker website [41].

[1] This code snippet illustrates optional chaining, where `bar` is only referenced from `foo` whenever `foo` is `not null`.

Interesting to note is that Stryker has made a few pragmatic choices when it comes to the mutation operators they have defined. The original ROR mutator, as mentioned in section 3.1.4, generates five more mutants to kill for every relational operator used. Most of these mutations do not make sense when considering the principle of *the competent programmer*. It is unlikely that when a developer was supposed to use the less-than operator, that they used the not-equal operator instead. As such, Stryker only mutates to likely mistakes. For example, '<' is only mapped to '<=' and '>=', and '==' is mapped to '! =' and vice versa.

3.2.2 Mutation Results

Besides the states *killed* and *survived*, Stryker has some other states for specifying the result of a mutant [2]. The state *no coverage* indicates that no tests can reach the mutant and is as such considered to be a survivor. The state *timeout* indicates that the mutant's execution duration exceeded the configured timeout duration or that the amount of times the mutated statement was executed exceeded the configured amount of executions. Timed out mutants are considered *detected* mutants. The idea here is that the mutant most likely caused an infinite loop, which can be detected by a CI build since the tests will never complete. The states *runtime* and *compile error* speak for themselves; these mutants are excluded from mutation score calculations. The final state *ignored* indicates that the mutant was ignored, either by configuration or some other reason, and is not included for calculating the mutation score. These mutant states are also summarised in Table 3.1 for convenience. For clarity, Stryker thus calculates the following metrics[3]:

$$\#\text{detected} = \#\text{killed} + \#\text{timeout} \tag{3.3}$$

$$\#\text{undetected} = \#\text{survived} + \#\text{no-coverage} \tag{3.4}$$

$$S_{stryker} = \frac{\#\text{detected}}{\#\text{detected} + \#\text{undetected}} * 100\% \tag{3.5}$$

Stryker will aggregate the results from the mutation testing session into a *mutation report*. This report includes the information described in section 3.1.3. Additionally, Stryker will upload the report to the Stryker Dashboard[4] when configured to do so.

[2]For further reference on mutant states, see `https://stryker-mutator.io/docs/mutation-testing-elements/mutant-states-and-metrics/#mutant-states`.

[3]For further reference on metrics, see `https://stryker-mutator.io/docs/mutation-testing-elements/mutant-states-and-metrics/#metrics`

[4]Stryker reports are uploaded to `https://dashboard.stryker-mutator.io/`.

Status	**Description**	**Included in mutation score**
Killed	At least 1 test failed for this mutant	Yes
Survived	All tests passed for this mutant	Yes
No coverage	No tests cover this mutant and is as such considered a survivor	Yes
Timeout	This mutant exceeded the configured timeout limit, or this mutant exceeded the maximum number of executions allowed	Yes
Runtime error	Attempts to run tests for this mutant resulted in an error	No
Compile error	This mutant could not compile	No
Ignored	This mutant was ignored, either by configuration or some other reason	No

Table 3.1: Overview of Stryker's mutant states.

3.2.3 Static Mutants

Aside from specifying the status of a mutant, Stryker also differentiates between whether a mutant is a *regular (non-static) mutant* or a *static mutant.* Static mutants are mutants that are only executed once, namely during startup. Usually, static mutants are derived from global constants and singleton classes. Once a static mutant is loaded, it is not possible to enable or disable them later during testing. Static mutants have a big impact on performance since (i) coverage analysis is inaccurate (or not possible at all) for static mutants, resulting in Stryker executing the complete test suite against static mutants; and (ii) in order to enable or disable a static mutant, a reset of the entire environment is required, which takes time.

3.2.4 Cost Optimisations

Stryker has employed many optimisation techniques already into their tool, namely: selective mutation, local concurrent test-runners (local parallelization), mutant coverage analysis, bail, incremental mutation testing, mutant schemata and hot reload. StrykerJS in particular has also implemented an optimisation that we will call *static reload*, which was not presented before as part of the list of optimisations in section 3.1.7 because it exists specifically for static mutants. Static reload allows a test-runner to unload a previously loaded static mutant, which means that a test-runner can execute multiple static mutants in succession without the need to reset the entire environment. Table 3.2 provides an overview of the optimisations implemented by Stryker per flavour. In addition to excluding mutation operators, Stryker's implementation of selective mutation is extended with the ability to exclude entire files from mutation as well as being able to exclude certain mutations by annotations in the source file with comments.

[5] The nature of JavaScript allows for this out of the box, no explicit implementation was necessary

[6] Only supported for some test-runners.

optimisation	Stryker4JS	Stryker.NET	Stryker4s
Selective Mutation	Y	Y	Y
Local Concurrent Test Runners	Y	Y	Y
Mutant Coverage Analysis	Y	Y	Y
Bail	Y	Y	Y
Incremental Mutation Testing	Y	Y	N
Mutant Schemata	Y	Y	Y
Hot Reload	Y[5]	N	N
Static Reload	Y[6]	N	N

Table 3.2: Overview of employed optimisations by Stryker per flavour.

Note that there may be differences in how (well) an optimisation is implemented across the different flavours. For example, with incremental mutation testing we have that StrykerJS simply compares with the previous mutation report that resides in the project without regard as to what produced it. In Stryker.NET there are options for selecting the baseline mutation report that will be used for incremental mutation testing. The baseline can even be selected based on the version control branch currently worked on, making it easier to perform mutation analysis for different branches.

Chapter 4

Simultaneous Mutation Testing

4.1 Concept

In order to reduce the cost of mutation testing, the solution we propose will decrease the time it takes to perform mutation analysis *without loss of quality*. In comparison to higher-order mutation, this optimisation should not affect the mutation score. The general idea is that we enable multiple mutants within a meta-mutant (recall mutant schemata from section 3.1.7), henceforth to be referred to as *simultaneous (mutation) testing*. This allows testing multiple mutants during a single test session. This could, in theory, reduce the number of test sessions by quite an amount. Most of the performance gain would come from not having to re-instantiate the entire environment for every mutant.

For purposes of consistent terminology, we will define a *mutant group* to be a set of mutants that will be enabled simultaneously, where the size of the mutant group is the number of mutants within that mutant group. A mutant that is a member of a mutant group is a *simultaneous mutant*. A mutant group that contains only 1 simultaneous mutant is called a *singleton mutant group*. Similarly, a simultaneous mutant within a singleton mutant group is called a singleton mutant. Additionally, we say that a simultaneous mutant has a properly defined result/status within its group when it is possible to derive a mutant status from the test results gathered up until the time of evaluation. For example, if at least 1 test from a simultaneous mutant has failed then it is considered killed and if all tests that cover a simultaneous mutant have been executed (and all these tests passed) then it is considered to be a survivor. Simultaneous mutants who have a properly defined status may also be referred to as being complete.

Only mutants that are *disjoint* in terms of test coverage will be grouped together

during simultaneous testing. Two mutants are considered disjoint when no single test executes at least 1 mutated statement of both mutants. For convenience, if some test t executes any of the mutated statements that belong to some mutant m, we say that t can reach m or that t covers m. Note that there can be different meanings to reachability, but in this case, we only consider a mutated statement to be reachable by a given test when the original statement was executed by that test. Enabling mutants that are *not* disjoint from each other may result in a *collateral*, killing a mutant only because the other mutant was enabled, which reduces the quality of mutation testing.

There is a conceptual difference between simultaneous testing and higher-order mutation. Higher-order mutation combines first-order mutations into a single higher-order mutant and is treated as if it were a single mutant. Killing any of the first-order mutations will mean that the higher-order mutant was killed. With simultaneous testing, the mutant group is executed as if it were a single mutant, but the individual mutants from which the group was formed have their own status.

4.2 Preliminary Research

Simultaneous testing already exists in Stryker.NET since 2020[1]. However, no scientific research has been reported on what it does exactly, how well it works in terms of performance and correctness, nor how it is implemented. Since Stryker.NET already contains simultaneous testing, we would like to research simultaneous testing for StrykerJS. Although there could be minor improvements to be made to Stryker.NET's variant of simultaneous testing, there is more to be gained from designing and implementing it for a Stryker flavour that does not have it yet.

Preliminary investigation revealed the following problems regarding simultaneous mutation testing for StrykerJS:

- When multiple mutants are enabled, we need to figure out which test killed which mutant. It is unclear to what extent this will impact the testing result. It may be that the result requires post-processing in order to properly identify which test killed which mutant;
- There also exist *static mutants* in StrykerJS as explained in section 3.2.3. Coverage analysis is not accurate for static mutants, meaning that determining whether a static mutant and another (non-)static mutant are disjoint cannot be done reliably. It is unclear whether it is possible to include static mutants for simultaneous testing.

[1]Created by dubdob, see also `https://github.com/stryker-mutator/stryker-net/pull/936`.

4.3 Research Questions

The main research question is formulated as follows: How can simultaneous testing be implemented into StrykerJS such that the cost of mutation testing, in terms of time, is reduced without loss of quality? The proposed solution will be validated and supported by finding answers to the following research questions:

1. How much impact will *simultaneous testing* have on the performance of Stryker in terms of execution time?

   The total runtime durations of StrykerJS will be measured with and without simultaneous testing enabled. This will be compared in order to derive a performance metric.

2. How much impact will *simultaneous testing* have on the quality of mutation testing?

   In order to validate the correctness of simultaneous testing, the results of simultaneous mutation testing and regular mutation testing should be compared. If the results differ for a specific mutant, then it means that there is a complex underlying issue with simultaneous testing which could be problematic for practitioners. These issues could arise due to poor coverage analysis, static mutants and (hidden) test dependencies. It could even be the case that two mutants that were originally considered disjoint in terms of test coverage actually have overlapping tests when grouped together. This is possible because one mutant might influence the control flow of the original program entirely. If the mutation scores of regular and simultaneous mutation testing differ too much, then the impact on the quality of mutation testing could be considered too drastic.

3. What is the best strategy for grouping mutants together in terms of performance?

   There are many ways by which disjoint mutants can be combined to form mutant groups. This is a computationally hard problem and may require a search strategy. A simple approach would be to just group the first found disjoint mutants together. A more advanced approach would be to make use of, for example, a SAT solver. The results of performing simultaneous mutation testing with the groups formed by each algorithm should be captured and then compared to figure out which strategy works best for increasing performance.

The validation as well as the performance and quality comparisons will be done by testing the implementation on multiple real-world applications that make use of StrykerJS. The Stryker community has a lot of members which should make it possible to gather numerous projects of different sizes.

Chapter 5

Theory

This chapter provides a theoretical view on the cost of (simultaneous) mutation testing. It provides a way to explain where the cost of mutation testing comes from. This is done for both regular and simultaneous mutation testing. The derivations for the cost of mutation testing that are provided in the following sections assume that *coverage analysis* and *bail* have been implemented. This is assumed because i) coverage analysis is required for the formation of mutant groups during simultaneous testing and because ii) bail is relatively simple to implement when compared to simultaneous mutation testing. Additionally, the existence of static mutants will be ignored completely because coverage analysis is inaccurate for static mutants.

This chapter is structured, as follows. The cost derivations for simultaneous mutation testing in section 5.2 build upon the foundation provided in section 5.1. Finally, a runtime specification is made for StrykerJS in section 5.3.

5.1 Mutation Testing

We can identify three stages in the general process of mutation testing as described in section 3.1.1, namely: program validation, mutant generation and mutation testing. The program validation stage entails running the test suite on the original program before continuing the process. If the test suite does not pass for the program, then it makes no sense to perform mutation testing. The generation stage includes the generation of mutant programs by the mutation operators. In the mutation testing stage, the test suite is run on all mutant programs.

In terms of cost, meaning the time it takes for a stage to complete, mutation testing is by far the most demanding stage. The cost of program validation and mutant generation is negligible compared to the cost of mutation testing. The cost of program validation can be thought of as being similar to performing

mutation testing for a single (surviving) mutant. The generation of mutants may take some time, depending on the efficiency of the algorithms used. Efficient algorithms should have linear complexity in the size of the program as well as in the number of mutation operators to be applied, meaning that the entire program only needs to be traversed once in order to generate all mutants.

There are many aspects to determining the cost of mutation testing. For convenience, the legend below is provided. For any set $\mathcal{S}$, cardinality is denoted as $|\mathcal{S}|$.

TS: the entire test suite, which is a set of tests. Members of TS are typically denoted t

T_t: time it takes to finish test t (for the original program, assumed to be the same for all mutants)

M: the set of mutants. Members of M are typically denoted m

G: the set of simultaneous mutant groups. A Member of G is typically denoted g, which is a set of mutants that are contained within the group

TS_m: the set of tests that is used to test mutant m, which is a subset of TS

T_m: time it takes for executing the test suite against mutant m

$T_{timeout}$: the maximum amount of time a mutant may run before being considered timed out

T_{TR}: time it takes to create a new test-runner

T_{CTS}: time it takes to create a new test session within a test-runner (time spent up until the first execution of any test, which includes setting up the entire environment)

Note that $T_{timeout}$ is a variable that should be determined by the mutation testing tool. It should *always* be larger than the duration it takes to execute the entire test suite against the original program. This duration can, for example, be measured during the check "Does P pass TS" illustrated in Figure 3.1.

With coverage analysis, we can determine reachability of a mutant by a test, which can be used to only execute a subset of the tests in the entire test suite for that mutant. Coverage analysis has an impact on the variable TS_m. It should be clear that TS_m is equal to TS when no coverage analysis has been done. With bail, we can stop executing tests whenever a test has failed. Bail only influences the time it takes to execute mutants that will be killed. The following runtime cost derivation can be made:

$$T_m = \begin{cases} \frac{1}{2} * \sum_{t \in TS_m} T_t, & m_i \text{ has been killed} \\ \sum_{t \in TS_m} T_t, & m_i \text{ has survived} \\ T_{timeout}, & m_i \text{ has timed out} \end{cases} \tag{5.1}$$

The derivation for mutants that will be killed is a worst case average scenario. Since the mutant is assumed to be killed, there should be *at least* 1 test in TS_{m_i} that can kill the mutant. When there exists *exactly* 1 test that can kill the mutant, the fraction of tests that need to be run on average is half the number of tests in TS_{m_i} (assuming random test execution order). However, it is likely that more than 1 of the tests in TS_{m_i} can actually kill the mutant, reducing the total number of test executions even further.

Mutants are tested by test-runners, as described in section 3.1.6, which need to create a unique test session for every mutant. The variables time spent creating test sessions (T_{CTS}) and test-runners (T_{TR}) are called process variables. Without any optimisations to the process, killed, surviving and timed-out mutants will all require a newly created test session. Additionally, for timed-out mutants, a new test-runner must be created. The following derivation can be made;

With process variables:

$$T_m = \begin{cases} T_{CTS} + \frac{1}{2} * \sum_{t \in TS_m} T_t, & m_i \text{ has been killed} \\ T_{CTS} + \sum_{t \in TS_m} T_t, & m_i \text{ has survived} \\ T_{CTS} + T_{timeout} + T_{TR}, & m_i \text{ has timed out} \end{cases} \quad (5.2)$$

Since all options in the system of equations just provided contain T_{CTS} exactly once, and because this system of equations holds for any mutant, it means that the total amount of time spent creating test sessions is equal to $|M| * T_{CTS}$.

For completeness let us also define the system of equations for when the optimisation *hot reload* is present. Hot reload reduces the time it takes to set up the environment for a mutant. It has a direct impact on T_{CTS}. How much impact the optimisation has on T_{CTS} may differ drastically between implementations. When a timeout occurs, hot reload does not provide any benefit since the entire environment will be purged and therefore cannot be reused. Hence, T_{CTS} will be divided into two special groups, namely T_{reset} and T_{reload}, where T_{reset} is essentially equal to the original definition of T_{CTS} and T_{reload} is the improved variant of T_{CTS}, meaning that $T_{reload} \leq T_{reset}$. In a system of equations that would be;

With hot reload and updated process variables:

$$T_m = \begin{cases} T_{reload} + \frac{1}{2} * \sum_{t \in TS_m} T_t, & m_i \text{ has been killed} \\ T_{reload} + \sum_{t \in TS_m} T_t, & m_i \text{ has survived} \\ T_{reset} + T_{timeout} + T_{TR}, & m_i \text{ has timed out} \end{cases} \quad (5.3)$$

It should be clear from this system of equations that the total time spent creating test sessions is dependent on the proportion of mutants that will lead to a timeout. It should also be clear that the total time spent creating test sessions is less than or equal to $|M| * T_{CTS}$ as opposed to being equal to $|M| * T_{CTS}$ as with Equation 5.2.

5.2 Simultaneous Mutation Testing

For simultaneous mutation testing, the duration of running a group of mutants is roughly equal to the sum of the time it takes to create a single test session and the time it takes to execute (a part of) the test suite for every simultaneous mutant from that group individually. With simultaneous testing, the total time spent creating test sessions is dependent on the number of groups $|G|$ that were created. So, instead of creating $|M|$ test sessions, you would only need to create $|G|$ test sessions. Hence, simultaneous testing should always be as fast or faster than regular mutation testing, since $|G| \leq |M|$. Note, however, that this is not always the case when the implementation of simultaneous testing comes with too much overhead, or when it conflicts with other optimisations, as will be shown later.

5.2.1 Performance Benefit

The duration of running a group of mutants g is equal to the sum of the time it takes to create 1 test session, the sum of testing simultaneous mutants individually (excluding process variables) and the overhead $\mathcal{O}$, which is an unknown function subject to g, which simultaneous testing will introduce. The overhead of simultaneous mutation testing may vary per group and has a direct impact on the time it takes to create a new test session and the time it takes to formulate a result based on the outcome of the test session. In formula[1]:

$$T_g = T_{CTS} + \sum_{m \in g} (T_m - T_{CTS}) + \mathcal{O}(g) \tag{5.4}$$

Note that T_m in this formula relates to the cost derivation without hot reload (see Equation 5.2), hence the use of T_{CTS} over T_{reload}. Also note the subtraction of T_{CTS} within the summation. This is done because T_m contains T_{CTS} exactly once for all paths in the system of equations.

It should be clear from this formula that, in terms of time, the maximum possible gain of running a group of mutants with size $|g|$, compared to running the simultaneous mutants from which this group is comprised of individually, is equal to $(|g| - 1) * T_{CTS}$. This implies that when hot reload is available, simultaneous mutation testing will have a lower possible gain since T_{CTS} will be smaller. Additionally, the actual gain is partially lowered by the overhead.

5.2.2 Timeouts with Mutant Groups

Actually, the formula just provided is only correct when the group will *not* time out. The moment a group times out, all the uncompleted simultaneous mutants from which it was comprised must be rerun individually. Every rerun will require a new test session, which partially diminishes the benefit of simultaneous

[1] This formula is correct when the simultaneous mutants in g do not lead to a time-out. This will be elaborated on in section 5.2.2.

mutation testing. For example, take the group $g = \{m_1, t_1, m_2\}$, where m_1 and m_2 are mutants that will *not* time out and t_1 is a mutant that will time out. Here, the maximum possible gain would be $2 * T_{CTS}$ if there were no timeouts. However, this particular group will actually require 2 test sessions, as opposed to just one, because the timeout of t_1 makes it impossible to determine the status of m_2 (as it will never complete its tests) and thus will need to be rerun in an individual test session. As a consequence, it may be beneficial to set a limit on the maximum size of the groups being formed, especially when there exists many mutants that will time out.

5.2.3 Amdahl's Law

The maximum possible gain from not having to create as many test sessions in terms of percentages of the overall duration of the process can be approximated by using Amdahl's law. Amdahl's law states the following: "the overall performance improvement gained by optimizing a single part of a system is limited by the fraction of time that the improved part is actually used"[37]. The theoretical possible gain from simultaneous mutation testing for a certain group can be computed with the following formula:

$$S_g = \frac{1}{(1-p) + \frac{p}{|g|}} \tag{5.5}$$

Where:

S_g = the theoretical speedup by testing the simultaneous mutants in g as a group as opposed to running them separately;
p = the proportion of time spent creating test sessions for the simultaneous mutants of g if they were tested separately;
$|g|$ = the size of the group.

For example, if the average time spent creating test sessions is 20% of the total duration of the average test session duration ($p = 0.2$), then the maximum possible performance gain of a group with 2 mutants ($|g| = 2$) is equal to $\frac{1}{1-0.2+\frac{0.2}{2}} \approx 1.111$. Meaning a theoretical performance improvement of 11.1%. See Table 5.1 for some more calculations with different values of p and $|g|$.

5.2.4 Smart Bail

Smart bail is an adaptation to the optimisation bail, specifically meant for simultaneous testing, which only bails the tests associated with a *killed* simultaneous mutant. The effect of smart bail on the duration of testing simultaneous mutants in a group should have the same effect that bail has on the duration of testing singleton mutants. If an implementation of simultaneous mutation testing does not support smart bail, a trade-off must be made between the performance gained from creating fewer test sessions and performance lost from

p	$\|g\|$	Performance increase
0.2	2	11.1%
0.5	2	33.3%
0.8	2	66.7%
p	$\|g\|$	**Performance increase**
0.2	3	15.4%
0.5	3	50.0%
0.8	3	114.3%
p	$\|g\|$	**Performance increase**
0.2	$\rightarrow \infty$	25%
0.5	$\rightarrow \infty$	100%
0.8	$\rightarrow \infty$	400%

Table 5.1: Calculation of the theoretical performance gain of using groups of certain sizes, based on Amdahl's law

not being able to bail any mutant after failing a test. The idea is that the odds of killing a mutant are higher when also more tests can reach that mutant. As a result, not being able to bail on a mutant that is reachable by a relatively large proportion of the test suite will likely increase the duration of mutation testing. When T_{CTS} is larger than the duration of running the tests for a mutant, then the benefit of performing simultaneous testing is higher than losing the ability to bail.

5.3 Runtime Specification: StrykerJS

StrykerJS consists of 4 stages, namely:

1. Prepare, which includes loading/validating configuration options, loading plugins, resolving input files, starting the logging server and initialising reporters;
2. Instrument, which includes parsing the input source files, finding possible mutations, instrumenting the code (injecting mutations), pre-processing the meta-mutant (recall the optimisation mutant schemata from section 3.1.7) formed (disables type checking, rewrites configuration files) and writing the output files to a sandboxed environment;
3. Dry-run, which includes testing validity of the meta-mutant without enabled mutants and performing a configurable coverage analysis strategy;
4. Mutation test, which includes creating mutation test plans (based on an incremental mutation test report), executing configured checkers and executing test suites of mutants that are covered by at least 1 test.

Compared to the generalized theory's stages described in section 5.1, the prepare

stage is new, the instrument stage is similar to mutant generation stage, the dry-run stage does program validation as well as coverage analysis and the mutation test stage does some checks, if configured to do so, before the actual mutation testing.

In terms of cost, the prepare, instrument and dry-run stages are negligible compared to the mutation test stage. The prepare stage does not depend on the size of the project, and is relatively small. Just as with the generalized mutant generation stage, the instrumentation stage's cost depends on the efficiency of the algorithms used. StrykerJS uses an algorithm of linear complexity. The cost of the dry-run stage can be thought of as being similar to performing mutation testing for a single (surviving) mutant. The mutation test stage has the highest cost and consists of two phases, namely:

1. Checker phase: consists of verifying the validity of a generated mutant. If a configured checker decides that a generated mutant is invalid, then the mutant will not be executed during the testing phase.

   There exists one official StrykerJS checker, namely the TypeScript checker. This checker verifies the validity of a mutant by enforcing the type constraints imposed by the typescript code. For example, the mutant generated by the 'FunctionBodyRemoval' mutator, which empties the body of a function, in a function that must return a concrete value (i.e. not `void` nor `undefined`) will not pass the TypeScript checker. This mutant will show up as 'compile error' and is not included in the calculation of mutation score.

2. Testing phase: consists of running the tests for a specific mutant, i.e. actual mutation testing. Mutants that reach this phase will have passed all configured checkers.

The optional checker phase, or more specifically the TypeScript checker, usually takes more time than the testing phase. The TypeScript checker will not be used during experiments since the goal of simultaneous mutation testing is to reduce the time needed to perform the testing phase.

StrykerJS may choose to short-circuit certain mutants, like skipping mutants based on an incremental mutation report as well as not executing mutants that are not reachable by any test at all. Only mutants that are not being short-circuited are used during simultaneous testing and behave the same as provided by Equation 5.3.

Chapter 6

Formation of Mutant Groups

The formation of mutant groups can be done in many ways. This chapter explains how to form acceptable groups in terms of constraints and provides two implementations that are able to form proper groups from a set of mutants. It also describes the main objective during the formation of mutant groups, which will all have an impact on the performance of simultaneous testing. For consistency, some symbols from the legend of section 5.1 are reused throughout this chapter.

6.1 Constraints

All algorithms that intend to form groups to be used during simultaneous mutation testing must satisfy the following constraints:

1. Invariant: each mutant must be present in *exactly* 1 group;
2. Invariant: for each mutant in a group, no tests may overlap with another mutant in that group.

The first constraint makes sure that every mutant is assigned to a group and not assigned to multiple groups. This is necessary because we wish to test each mutant at least once in order to retain the quality of mutation testing. However, we should not use any mutant twice as that would be a waste of time. The second constraint makes sure that only mutants that are disjoint in terms of test coverage can be assigned to the same group, which is also required to retain the quality of mutation testing.

6.2 Objective

Based on the theory of simultaneous mutation testing (see section 5.2), it is expected that some feasible set of mutant groups G_1 will perform better than

another feasible set of mutant groups G_2 if $|G_1| < |G_2|$. Hence, for an optimal solution, algorithms should consider prioritizing solutions with the least number of mutant groups to be the main objective.

6.3 Concept

This problem can be simplified by first introducing the *reachability matrix*. A reachability matrix is an $|M| \times |TS|$ Boolean matrix (each cell can only have the value 0 or 1). Consider an $|M| \times |TS|$ reachability matrix $\mathcal{R}$, then each cell $\mathcal{R}_{m,t}$ in $\mathcal{R}$ indicates that mutant m is reachable by test t when $\mathcal{R}_{m,t}$ is 1. Mutant m is *not* reachable by test t when $\mathcal{R}_{m,t}$ is 0. The reachability matrix is provided to the solver as factual input. For readability purposes, the binary relation $overlaps(m_1, m_2)$ is derived from $\mathcal{R}$, which returns true when m_1 has *any* overlapping tests with m_2, false otherwise.

The formed groups can be interpreted as a $|G| \times |M|$ Boolean matrix, which is called a *group matrix*. Consider a $|G| \times |M|$ group matrix $\mathcal{G}$, then each cell $\mathcal{G}_{g,m}$ in the matrix indicates that mutant m is contained within group g whenever the cell has the value 1. Any group matrix represents a feasible solution when it adheres to the invariants provided before. In the worst case scenario, the solution to the group matrix is equivalent to the identity matrix $I_{|M|}$.

To ensure that every mutant is in *exactly* 1 group, the following propositions must be satisfied:

$$\forall m \in M. \exists g \in G. \mathcal{G}_{g,m} \tag{6.1}$$

$$\forall m \in M. \forall g_1, g_2 \in G. \mathcal{G}_{g_1,m} \wedge \mathcal{G}_{g_2,m} \implies g_1 = g_2 \tag{6.2}$$

Where the first proposition (Equation 6.1) ensures that every mutant is grouped and the second proposition (Equation 6.2) ensures that every grouped mutant is not in any other group. To ensure that no tests overlap by mutants within a group, the following proposition must hold:

$$\forall g \in G. \forall m_1, m_2 \in M. \mathcal{G}_{g,m_1} \wedge \mathcal{G}_{g,m_2} \wedge overlaps(m_1, m_2) \implies m_1 = m_2 \tag{6.3}$$

Additional constraints can be imposed to have more control over the size of the formed groups. As was explained in section 5.2.2, setting a limit on the maximum size of mutation groups could be beneficial if many mutants lead to timeouts. Limiting the size of mutation groups to L could be enforced by including the following proposition:

$$\forall g \in G. |\{m \in M | \mathcal{G}_{g,m}\}| \leq L \tag{6.4}$$

It is also possible to fix the size of mutation groups. Fixing the size of the mutant groups is only useful for research purposes, as it allows the researcher to

Mutant	**Tests**	**Mutant**	**Tests**
m_0	0,1,2	m_1	3,4,5
m_2	6,7,8	m_3	0,1,9
m_4	2,4,6	m_5	3,5,7
m_6	7,8,9	m_7	1,2,9
m_8	9	m_9	4,7
m_{10}	5,7	m_{11}	1,2,4,7,9
m_{12}	2,3	m_{13}	0

Table 6.1: Scenario coverage data.

figure out which groups of certain sizes perform the best. Unlike the size limit just provided, this forces the groups to be of a certain size S. Fixing the size of a group to just S is not that useful as it is unlikely that such a solution exists (unless $S = 1$). As such, we also allow the size of a group to equal 1. If combined with the main objective provided in the introduction of this chapter, it will try to find as many groups of size S as possible and form singleton groups for all the remaining mutants. This can be expressed by the following proposition:

$$\forall g \in G. |\{m \in M | \mathcal{G}_{g,m}\}| \in \{1, S\} \tag{6.5}$$

6.4 Example Scenario

A system is being tested with a mutation testing tool and 14 mutants (m_0 to m_{13}) were generated. The test suite contains a total of 10 tests (t_0 to t_9) and all tests in the suite are useful, in the sense that each test can reach at least 1 mutant. Coverage data is shown in Table 6.1. In the table, mutants are paired with a comma-separated list of test identifiers that indicate which tests can reach that specific mutant. For example, mutant 9 (m_9) is reachable by tests 4 and 7. Based on the coverage data, the 14×10 reachability matrix $\mathcal{R}$ is determined:

$$
\mathcal{R} = \begin{bmatrix}
 & t_0 & t_1 & t_2 & t_3 & t_4 & t_5 & t_6 & t_7 & t_8 & t_9 \\
m_0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
m_1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
m_2 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\
m_3 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
m_4 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
m_5 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\
m_6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
m_7 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
m_8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
m_9 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
m_{10} & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
m_{11} & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\
m_{12} & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
m_{13} & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
$$

Then, the 14×14 group matrix $\mathcal{G}$ is initialized. Starting with all zeros, indicating that no mutants are assigned to any group yet[1]:

$$
\mathcal{G} = \begin{bmatrix}
 & m_0 & \cdots & m_{13} \\
g_0 & 0 & \cdots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
g_{13} & 0 & \cdots & 0
\end{bmatrix}
$$

Finally, the solver is executed to solve this problem, obeying the constraints. The following solution can be found:

$$
\mathcal{G} = \begin{bmatrix}
 & m_0 & m_1 & m_2 & m_3 & m_4 & m_5 & m_6 & m_7 & m_8 & m_9 & m_{10} & m_{11} & m_{12} & m_{13} \\
g_0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
g_1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
g_2 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
g_3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
g_4 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
g_5 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
g_6 & 0 & & & & & & \cdots & & & & & & & 0 \\
\vdots & \vdots & & & & & & \ddots & & & & & & & \vdots \\
g_{13} & 0 & & & & & & \cdots & & & & & & & 0
\end{bmatrix}
$$

The solution consists of six non-empty groups and eight empty groups. As it turns out, this solution is actually optimal in terms of the number of groups that were formed, due the following observation. Based on the reachability matrix, one can find the test that can reach the most mutants. In this case, t_7 can reach the most mutants, namely 6, meaning that the minimum number of groups that we must form is six, otherwise it is not possible to abide to the second invariant

[1] The group matrix can also be initialized to be equivalent to the identity matrix (I_{14}), indicating that every mutant is assigned to a group of its own.

(no overlapping tests).

6.5 Pragmatic Algorithm

The first algorithm is relatively simple. The basic idea is to group the first found disjoint mutants together until all mutants have been grouped. However, before doing so, the mutants are sorted based on the number of tests that can reach the mutants such that the process of finding disjoint mutants is more efficient.

We have implemented the pragmatic algorithm in TypeScript for StrykerJS. It is an adaptation from Stryker.NET's grouping algorithm. The pseudocode is shown in algorithm 1. First, the input M, a set of mutants, is split into two partitions, namely: `mutantsToGroup` and `singletonMutants` (lines 6-7). The conditions for a mutant to be grouped in one of the partitions may differ between implementations for different mutation testing tools. For StrykerJS specifically, the condition for a mutant to be part of `mutantsToGroup` is that coverage data is available for the mutant (this is not reflected in the pseudocode). A new group is created for each singleton mutant, which is stored in `singletonMutantGroups`. The partition of mutants to group is first sorted, in increasing order, based on the number of tests that can reach that mutant. Then, we keep iterating over all mutants to group and put them in the same group (`nextGroup`) as long as they have no overlapping tests with the other mutants' test suites already in the group (`simultaneousTestSet`). An iteration is terminated early when the size of the currently formed group's test suite plus the size of a candidate mutant's test suite is greater than the size of the entire test suite, since that would mean that the candidate must have overlapping tests with the formed group so far. It also means that all subsequent candidates must have overlapping tests, hence we can halt this iteration (lines 17-18). Once a mutant is assigned to a group, they are removed from `mutantsToGroup`, ensuring that every mutant belongs to exactly one group. The groups formed from both partitions are then merged and returned.

Algorithm 1 Pragmatic algorithm for forming simultaneous mutant groups.

```
1: M ← set of mutants    ▷ Mutants within the set contain the property ts, indicating the test suite that can reach the mutant (TS_m). For some mutant m, this property is accessed by the dot notation m.ts
2: |TS| ← the size of the entire test suite
3: satisfiesConditionsForGrouping ← a function that determines whether a specific mutant satisfies the conditions for it to be grouped with other mutants
4: sortBasedOnTestSize ← a function that is able to sort a set of mutants in increasing order based on the size of the tests that can reach the mutants
5: function formSimultaneousGroups(M, |TS|)
6:     mutantsToGroup ← {m ∈ M| satisfiesConditionsForGrouping(m)}
7:     singletonMutants ← {m ∈ M| ¬satisfiesConditionsForGrouping(m)}
8:     singletonMutantGroups ← {{m}|m ∈ singletonMutants}
9:     mutantsToGroup ← sortBasedOnTestSize(mutantsToGroup)
10:    G ← φ
11:    while |mutantsToGroup| > 0 do
12:        firstSimultaneousMutant ← mutantsToGroup[0]
13:        simultaneousTestSet ← φ ∪ firstSimultaneousMutant.ts
14:        nextGroup ← {firstSimultaneousMutant}
15:        mutantsToGroup ← mutantsToGroup\firstSimultaneousMutant
16:        for candidate ∈ mutantsToGroup do
17:            if |simultaneousTestSet| + |candidate.ts| > |TS| then
18:                break
19:            else if ∃t_1 ∈ candidate.ts, t_2 ∈ simultaneousTestSet.t_1 = t_2
   then
20:                continue
21:            end if
22:            nextGroup ← nextGroup ∪ {candidate}
23:            mutantsToGroup ← mutantsToGroup \ candidate
24:            simultaneousTestSet ← simultaneousTestSet ∪ candidate.ts
25:        end for
26:        G ← G ∪ {nextGroup}
27:    end while
28:    return G ∪ singletonMutantGroups
29: end function
```

6.6 Constraint Solver Algorithm

Finding the optimal solution based on a set of mutants with their coverage data is (in general) a computationally hard problem. To do that we need a tool that can work with the constraints as provided in section 6.3. For this purpose we chose Google OR-Tools' [8] constraint programming solver. This solver finds feasible solutions to problems that are expressed by a set of constraints. The source code of the implementation can be found here[2]. Unfortunately, the constraint solver used does not support universal quantifiers directly. As such, the propositions provided in section 6.3 need to be rewritten slightly, which is done in section 6.6.4.

6.6.1 Input

The input to the solver may contain the fields 'mutants', 'fixedGroupSize' and 'maximumGroupSize'. The field 'mutants' is an array of Mutant objects, which contains the fields 'id' and 'tests'. See Table 6.2 and Table 6.3 for a description of these fields.

6.6.2 Axioms

The solver cannot work with the provided input directly, it must first be converted to axioms with which it can work. To do this, the reachability matrix $\mathcal{R}$ is constructed from the input, as described in section 6.3. Then, another matrix is pre-calculated which serves as the binary relation *overlaps*. This *overlaps* relation is provided to the solver as a fact.

6.6.3 Variables

The variables, which are also the solution to the problem, are encoded through the group matrix $\mathcal{G}$, as described in section 6.3. Note that the dimension of the group matrix is chosen to be $|M| \times |M|$ because it covers the worst case scenario, one where no disjoint mutants exist.

[2] `https://github.com/mcdr2k/simultaneous-mutant-grouping`

Field	**Description**	**Optional**
Mutants	Array of Mutants (see Table 6.3)	False
FixedGroupSize	If provided, forces the solver to only accept groups of the provided size	True
MaximumGroupSize	If provided, adds a constraint that enforces a maximum number of mutants in a group	True

Table 6.2: Description of the SMT expected input JSON object.

Field	Description	Optional
Id	Unique identifier of the mutant (string)	False
Tests	The set of tests that can reach this specific mutant as an array of test ids (strings)	False

Table 6.3: Description of the Mutant object.

6.6.4 Constraints

Constraints in this solver are expressed by a model (`CpModel` in this case). The model contains methods for creating specific constraints that take an expression, in the form of a Java object. For example, the method `addLessOrEqual(`e_1, e_2`)` adds the constraint that the value of the first expression must be less or equal to the second expression, or simply $e_1 \leq e_2$. The propositions as defined in section 6.3 make use of universal quantifiers, which the `CpModel` does not support directly, but can be imitated with programming constructs. In particular, the forall universal quantifier's behavior is imitated with (nested) for-loops. The propositions are rewritten slightly to make it work, as follows:

- Recall from section 6.3 that each row is a group and each column is a mutant in the matrix $\mathcal{G}$. To ensure that every mutant belongs to exactly one group a constraint can be made on each column of the group matrix. For every column in $\mathcal{G}$, the sum of all values in the column should equal 1. In code:

```
1 for (int m = 0; m < M; m++) {
2     List<Literal> mutant = new ArrayList<>(M);
3     for (int g = 0; g < M; g++) {
4         mutant.add(groupMatrix[g][m]);
5     }
6     model.addExactlyOne(mutant);
7 }
```

  This essentially means that the propositions from Equation 6.1 and Equation 6.2 have been rewritten to:

$$\forall m \in M.\forall g \in G.|\{g|\mathcal{G}_{g,m}\}| = 1 \tag{6.6}$$

- To ensure that every mutant that is part of the same group does not have overlapping tests a constraint is added between every 2 different mutants for every group. The constraint expresses that the sum of any two mutants within the same group must be less or equal to 1, *unless* they have no overlapping tests. Whether two mutants have no overlap is determined through the pre-calculated *overlaps* matrix. In code:

```
1 for (int g = 0; g < M; g++) {
2     for (int m1 = 0; m1 < M; m1++) {
```

```
3         var l1 = groupMatrix[g][m1];
4         for (int m2 = 0; m2 < M; m2++) {
5             if (m1 == m2) continue;
6             var l2 = groupMatrix[g][m2];
7             var hasOverlap = overlaps[m1][m2];
8             // if two mutants have overlapping tests,
9             // then at most 1 of them can be true
10            model.addLessOrEqual(
11                    LinearExpr.sum(
12                        new LinearArgument[]{l1, l2}),
13                        1
14                    )
15                .onlyEnforceIf(hasOverlap);
16        }
17    }
18 }
```

This essentially means that the proposition from Equation 6.3 has been rewritten to:

$$\forall g \in G.\forall m_1, m_2 \in M.m_1 \neq m_2 \wedge \mathit{overlaps}(m_1, m_2) \implies \mathcal{G}_{g,m_1} + \mathcal{G}_{g,m_2} \leq 1 \tag{6.7}$$

- Setting a limit on the maximum size of the groups can be achieved through the less or equal constraint, where the value to test is the sum of a row of $\mathcal{G}$. This is equivalent to the proposition described by Equation 6.4. In code:

```
1 for (int g = 0; g < M; g++) {
2     model.addLessOrEqual(
3         LinearExpr.sum(groupMatrix[g]),
4         input.maxGroupSize
5     );
6 }
```

- Fixing the size of the groups can also be achieved by a test on the sum of a row of $\mathcal{G}$. This is done through half-reified linear constraints, with any row g and an independent Boolean variable b, in code:

```
1 for (int g = 0; g < M; g++) {
2     var groupSize = LinearExpr.sum(groupMatrix[g]);
3     var b = model.newBoolVar("reifyFixedGroup" + g);
4     model.addLessOrEqual(groupSize, 1).onlyEnforceIf(b);
5     model.addEquality(groupSize, input.fixedGroupSize)
6         .onlyEnforceIf(b.not());
7 }
```

This essentially means that the proposition described by Equation 6.5 has been rewritten to:

$$\begin{aligned}
&\forall g \in G : \\
&\qquad b_g \implies (\sum g) = 1, \text{ and} \\
&\qquad \neg b_g \implies (\sum g) = fixedSize
\end{aligned} \tag{6.8}$$

6.6.5 Objective

The primary goal of this optimization problem is to group mutants together in such a way that we have as few groups as possible. The objective function is defined over the group matrix $\mathcal{G}$. For every group in $\mathcal{G}$, a new Boolean variable $gHasMutants$ is introduced that states whether group g contains any mutants. The sum over all these variables indicates how many groups were formed. This sum is then added to the model with the minimization objective, forcing the model to search for solutions with as few non-empty groups as possible.

6.6.6 Output

The output of the constraint solver consists of the result and a timer. The result consists of a status and the actual solution found. The status indicates whether the solution is simply any feasible solution or an optimal solution. If any other status was returned then there is no solution, likely due to malformed input. The solution is a 2 dimensional array of mutant groups, derived from the group matrix $\mathcal{G}$. Where the outer array is an array of groups and the inner arrays are arrays of mutant ids that belong to the same group. For example, the solution `[[1, 2], [3]]` indicates that two groups were found, where g_1 contains m_1 and m_2 and g_2 contains m_3. Empty groups are not included in the solution. See Table 6.4 for an overview of the data included in the result. Finally, the timer consists of numerous fields that indicate how much time certain tasks take, including the total time spent creating a model of the problem and solving the problem. See Table 6.5 for an overview of the durations included in the output.

6.6.7 Implementation Particularities

Finding the optimal solution to large problems with a constraint solver is infeasible as there exist many solutions, meaning that it will simply take too much time. Because of that, an arbitrarily chosen timeout of 90 seconds was set on the solver, which causes the solver to return the best solution that it could find within 90 seconds.

After testing it was found that large problems causes the program to run out of memory. When the JVM was given 8GB of memory, it would crash on inputs greater than 225 mutants and a test suite size of 100. When it was given 50GB of memory, the maximum input size could be increased to about 325 mutants

Field	**Description**
Status	Status indication of the solution. Can be any of the following: feasible, infeasible, optimal, unknown, invalid
Solution	The formed mutant groups presented as a 2-dimensional array of mutant ids

Table 6.4: Description of the SolverResult object.

Field	**Description**
SetupAxiomsDurationNanos	Time spent setting up the model's axioms
SetupVariablesDurationNanos	Time spent setting up the model's variables
SetupConstraintsDurationNanos	Time spent setting up the model's constraints
CreateObjectiveDurationNanos	Time spent creating the objective function
SolveDurationNanos	Time spent solving the model
TotalDurationNanos	Total time spent on all operations

Table 6.5: Description of the Timer object.

and a test suite size of 140. This shows that the space complexity of the solver is highly exponential. This is problematic as many projects generate much larger input than the solver can handle. As a consequence, the solver was modified to split the provided input into multiple parts. After solving for each part of the input, the results will be combined. Even though all the individual results of the different parts may be optimal, the result may not be optimal. Also note that the solver will be given at most 90 seconds for each part of the input as opposed to being given 90 seconds for the whole input.

Chapter 7

Implementation of Simultaneous Mutation Testing for StrykerJS

This section discusses some aspects of how we implemented simultaneous mutation testing for the mocha test-runner for StrykerJS. Understanding the implementation is important for some parts of the process. The implementation details should help in understanding the semantic differences with regards to regular mutation testing. It should also help in clarifying why simultaneous testing would perform better or worse than regular mutation testing. The implementation can be found here[1].

7.1 Simultaneous Mutant Schemata

StrykerJS supports *mutant schemata*, which means that StrykerJS is able to dynamically change whether a certain mutant is active. The original implementation (simplified) would store the so-called active mutant in a global variable. Every time a mutated statement would be executed by the original program, it instead checks first whether the global active mutant variable is equal to the mutant's identifier. If true, it would continue executing the mutated statement, otherwise it would execute the original non-mutated statement. Since we need to enable multiple mutants at the same time, this global variable has been changed to be a set of active mutant ids.

[1] `https://github.com/mcdr2k/stryker-js/tree/simultaneous-mutation-testing-experiment1`. This link points to the branch used during the experiment in chapter 9.

7.2 Simultaneous Infinite Loop Detection

StrykerJS also supports infinite loop detection. This allows for Stryker to terminate, for example, `while (true){}` loops early. Although it cannot detect all such cases, it works most of the time. This loop detection is supported by yet two more global variables, namely hitcount and hitlimit. If the hitcount exceeds the hitlimit, then Stryker terminates early by throwing an error. Hitcount is increased by 1 every time the mutated statement is executed, hitlimit on the other hand is determined during the dry-run and will be fixed throughout the test run. To support this for simultaneous testing, both hitlimit and hitcount have been converted to a map, which maps a mutant's identifier to the current hitcount or hitlimit.

7.3 Timeouts & Live Reporting

During implementation it was found that simultaneous mutation testing was completely useless if it were unable to detect which mutant of a group caused a timeout. A timeout would mean that we should rerun all simultaneous mutants from the timed-out group individually to produce a proper result. That would include the mutant that caused the timeout in the first place, which would impact the performance greatly. However, the current implementation of StrykerJS had nothing in place to detect which test caused a timeout. The original implementation would just terminate the child process that is running the tests for a certain mutant if it did not return a result after the configured timeout. In order to be able to detect which test caused the timeout, it was necessary to implement 'live reporting'. Live reporting entails that child-processes report back all results of all tests that are about to be executed and have been executed to the parent process. Upon timing out, it is possible to determine which test never finished executing by the updates sent by the child-process and is assumed to be the timed-out test. The process of live reporting is illustrated in Figure 7.1.

This does bring us to another major change revolving timeouts, which is when does one decide a mutant (group) has timed out? As of now, simultaneous testing considers a group to be timed out when the parent process has not received any updates within configured timeout duration. The configured timeout durations are the same for regular and simultaneous testing. This means that simultaneous testing allows each test to run for the duration of the configured timeout, whereas the original implementation only allows the entire test run to run for the duration of the configured timeout. As a result of live reporting and the differences in timeout, comparisons between regular mutation testing and simultaneous mutation would be unfair. It is expected that simultaneous mutation testing takes longer because of this change. These differences are taken into account during the experiments in chapter 9.

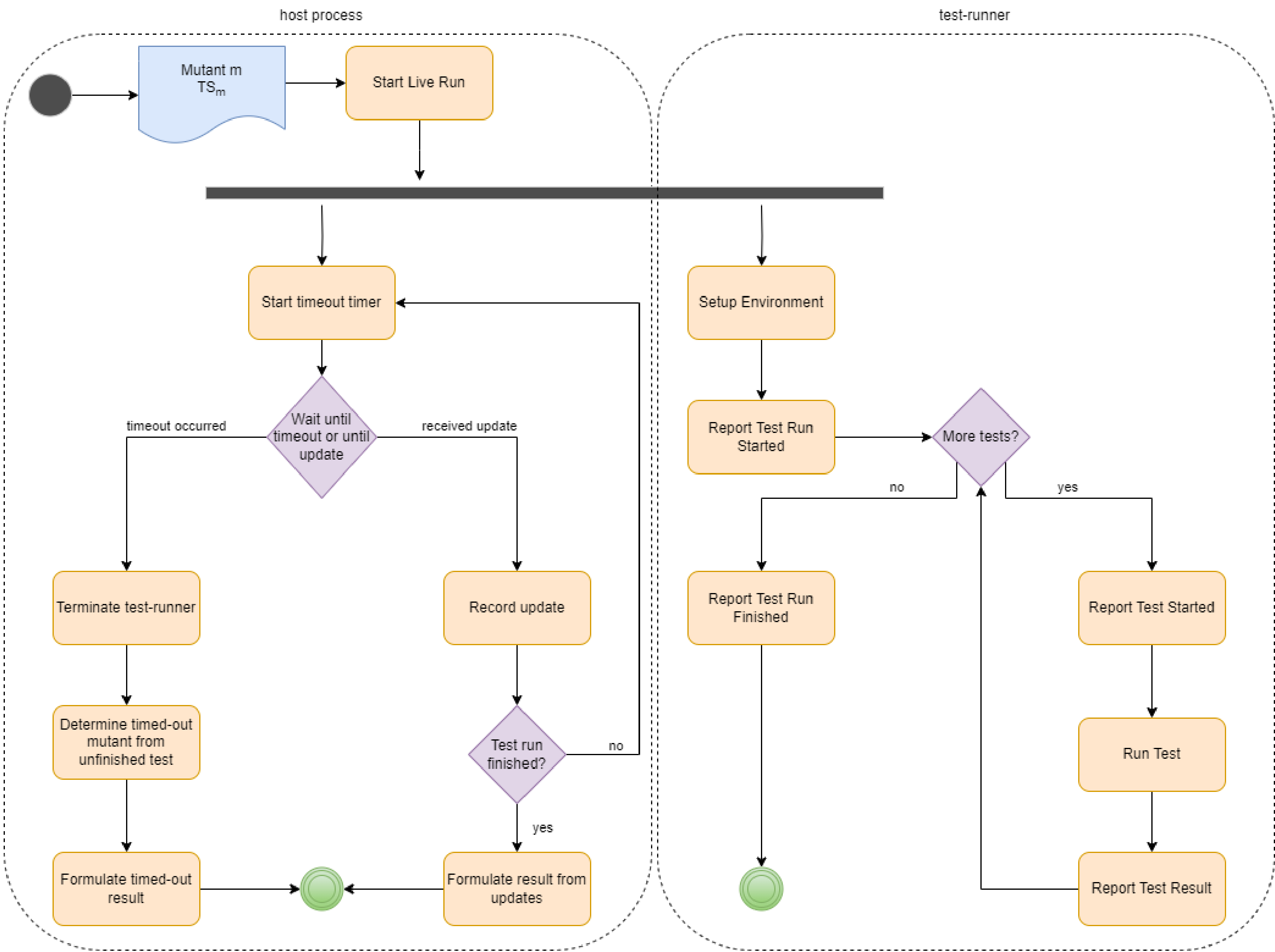


Figure 7.1: Process of live reporting during the execution of a mutant.

7.4 Smart Bail

We implemented smart bail for the mocha test-runner, however it is not as effective as the original bail. In Mocha, programmers can describe suites[2] (groups of tests) with the 'describe' function. Tests within suites are created with the 'it' function. The idea is that programmers can group the individual tests logically in a test suite. It also provides some guarantees to the execution order of the tests. For instance, the tests within the suites will always execute in the order they are defined while the test suites themselves do not provide any guarantee to the execution order other than being deterministic between multiple executions.

The current implementation of smart bail loops over all tests in a suite the moment it has started executing and checks for each test whether they are associated with a mutant that has already been killed. If so, then the test is skipped, otherwise do nothing (allowing it to run). This implementation of smart bail thus works between different suites but does not work between tests within the same suite. Take for example the following JavaScript test snippet:

```
1  describe('suite 1', () => {
2      it('test 1', () => {
3        // test that cannot kill mutant m1
4      });
5      it('test 2', () => {
6        // test that kills mutant m1
7      });
8      it('test 3', () => {
9        // test that eventually kills mutant m1
10     });
11 }
12
13 describe('suite 2', () => {
14     it('test 1', () => {
15       // test that times out on mutant m1
16     });
17 }
```

Let us assume that the tests are executed in the exact same order as they are described. With the current implementation of smart bail, mutant m_1 will execute all three tests in suite 1 because it cannot bail on tests in the same suite. At least, in this particular example, it will not execute suite 2 at all, which is great because it would lead to a time-out. Ideally, smart bail would also be able to bail right after test 2 has completed execution such that the relatively long running test 3 is not executed at all.

[2] Mocha suites should not be confused with the general term of a 'test suite' used throughout this report. All Mocha suites described are part of the test suite TS during mutation testing.

7.5 Configuration Options

Numerous new configuration options have been added to StrykerJS to customise the behavior to some extent. Most of these options will be used during the experiment in chapter 9. The following configuration options were added:

- `enableSimultaneousTesting`: Enables mutation testing with simultaneous mutants. When true, runs all mutants in disjoint groups in terms of test coverage where possible. This will increase performance but may decrease accuracy of the results. Defaults to false.
- `exportMutantsOnly`: Mutates the project and runs the configured checkers, then exports the remaining mutants to a file (only exports mutants that are not killed by the checker). After the mutants have been exported, the program terminates. Defaults to false.
- `exportMutantsFile`: Target file where the generated valid mutants should be exported to. Defaults to 'reports/generated-mutants.json'.
- `measureMetrics`: Indicates whether Stryker should measure metrics and export them to a file, defaults to false.
- `measureMetricsOutputFile`: Target file where the metrics should be exported to, default to 'reports/metrics.json'.
- `maximumGroupSize`: Sets the maximum size of the simultaneous mutant groups formed when simultaneous testing is enabled. Does nothing when importing mutant groups from a file. Defaults to 0.
- `importMutantGroups`: Indicates whether the program should import mutant groups from a file. These groups will be used for simultaneous testing. Defaults to false.
- `importMutantGroupsFile`: Source file from which mutant groups are imported. Defaults to 'reports/grouped-mutants.json'.
- `fakeTestSessionCreationDuration`: Artificially introduces work for creating a test session to increase the time spent creating test sessions. Defaults to 0.

Chapter 8

Validation

To evaluate the performance, quality and correctness of simultaneous mutation testing, it is necessary to test the implementation on numerous real-world applications. The following sections describe how test projects are gathered, how performance and quality are evaluated and which metrics are required for said evaluation.

8.1 Gathering test subjects

A Google form was created to gather real-world applications that make use of StrykerJS. The content of the form can be found in Appendix A. The form was published in Stryker's Slack channel and later tweeted on X. Unfortunately, no responses were recorded. Due to the lack of responses, we instead inquired the maintainer from the Stryker dashboard (see section 3.2.2) to retrieve a list of popular Stryker projects. The list contained 135 projects but after removing duplicates, projects that do not use StrykerJS (PHP projects were included in the list) and projects that do not make use of explicitly the mocha test-runner, only four projects remained. These four projects contained a total of six modules that can be used as test subjects for validation purposes. Table 8.1 shows an overview of the test subjects' characteristics, sorted alphabetically, including the total lines of code (LOC), number of non-static mutants ($|M|$) and the mutation score. A description of the projects' capabilities can be found below:

1. Cucumber expressions: provides an alternative to regular expressions. The idea is that the syntax used for cucumber expressions is more intuitive than the syntax used by regular expressions. For example, to parse an integer one can simply insert `"{int}"` into the cucumber expression as opposed to inserting `"0|-?[1-9][0-9]*"` for a regular expression.
2. Mutation testing elements: streamlines the display of mutation testing results by producing html from mutation reports. It includes the calculation

Project	**Module**	**LOC**	$\|M\|$	**Mutation Score**
cucumber-expressions[1]	cucumber-expressions	1573	984	83.2%
mutation-testing-elements[2]	metrics	641	288	70.5%
stryker-js[3]	instrumenter	2205	1411	83.9%
stryker-js	mocha-runner	521	250	71.3%
stryker-js	typescript-checker	617	364	83.9%
typed-inject[4]	typed-inject	331	120	96.7%

Table 8.1: Characteristics of test subjects.

of metrics, such as the mutation score, for which the module metrics is responsible.

3. StrykerJS: mutation testing tool for JavaScript and TypeScript. The first test subject of this project is instrumenter, which is responsible for generating all the mutants. The second test subject is mocha-runner, which is a test-runner implementation specifically created for the test framework Mocha. The final test subject in this project is typescript-checker, which is responsible for checking whether generated mutants adhere to the type constraints in the context of the code they are generated in.

4. Typed inject: type-safe dependency injection framework for TypeScript with which one can inject classes, interfaces and values.

8.2 Evaluating performance

Certain operations need to be timed for evaluating the performance during regular and simultaneous mutation testing. Since the goal of simultaneous testing is to improve the performance of mutation testing, only that stage of the entire process will be measured, in isolation. The time it takes for executing the test suite against each mutant individually is measured during regular mutation testing (T_m). Similarly, the time it takes for executing the test suite against each mutant group is measured during simultaneous mutation testing (T_g). To explain differences in performance, we are also required to measure, or derive, the time spent creating test sessions (T_{CTS}) and the overhead ($\mathcal{O}$). From the results, a difference in performance will be derived in terms of percentages, where 0% would mean no difference, less than 0% would mean a gain in performance and greater than 0% would mean a loss in performance.

[1] Git version hash: 501114b58e31e3b172b6aa65786a2d29ef04fe94
[2] Git version hash: c2a76e49614ec0165175b6f71081b3cf53e367ec
[3] Git version hash: 905889797e913291319645f49a1a99a762671781
[4] Git version hash: b865c8433c9ec101c1f51e16921519dd14cc7446

Expected status	Acceptable status
Killed	Killed, timeout
Survived	Survived
Timeout	Killed, timeout
No coverage	No coverage
Ignored	Ignored
Runtime error	Runtime error, killed, timeout
Compile error	Compile error

Table 8.2: Table mapping indicating acceptable status differences. See Table 3.1 for an overview on mutant states.

8.3 Evaluating quality

The output of (simultaneous) mutation testing will be recorded and then compared. To quantify the difference in quality, we need to compare the result of each mutant individually. More specifically, we need to compare the status of each mutant. Mutants that are killed by regular mutation testing should also be killed by simultaneous mutation testing. Likewise, mutants that survive regular mutation testing should also survive during simultaneous mutation testing.

Some of the results for the mutants depend on the order of execution of the test suite. The killed and timeout status are an example of that. If the test that will lead to a timeout is executed before a test that can kill the mutant, the result will be a timeout. Even more so, if *smart bail* has not been implemented, more simultaneous mutants will lead to a timeout even though they may have already been killed by another test before executing the test that causes a timeout. See Table 8.2 for an overview of these special cases. The expected status, or original status from regular mutation testing, may correspond to multiple acceptable statuses during simultaneous testing. As long as simultaneous testing has resulted in one of the acceptable states, then we consider the result of the mutant to be equal.

8.4 Evaluating Grouping Algorithms

Two grouping algorithms were introduced in chapter 6, namely: the pragmatic and the constraint algorithm. Both algorithms should be evaluated and compared. It is expected that the pragmatic algorithm is faster than the constraint algorithm, in terms of the total time it takes to form groups. It is also expected that the pragmatic algorithm yields a result with more groups than the constraint algorithm. To do this comparison, the total runtime for both algorithms will be recorded as well as the groups that are formed.

Chapter 9

Experimental Setup

To carry out the validation as described in chapter 8, a proper setup is required. Running StrykerJS on a test subject will be done in four stages, namely:

1. Baseline: The baseline stage entails running StrykerJS without simultaneous testing. The results of each mutant will be captured by the baseline run and will be assumed to be the correct result. This allows us to verify whether the other runs have the same output as the baseline.
2. Verification: The verification run will have simultaneous testing enabled but every group formed by the pragmatic algorithm will be forced to be of size 1. The results of the mutants should then be equivalent to the results of the baseline since the mutants are run in isolation. If the verification run has different results for the mutants, then it would mean that there is a fundamental issue with the implementation of simultaneous testing in StrykerJS.
3. Pragmatic: The pragmatic run has simultaneous mutation testing enabled and will make use of the pragmatic grouping algorithm to form simultaneous mutant groups.
4. Solver: The (constraint) solver run has simultaneous mutation testing enabled and will make use of the constraint solver grouping algorithm to form simultaneous mutant groups. As explained in section 6.6.7, the solver might run out of memory if the input is large. For that reason, we split the input into multiple parts of 200 mutants. If that still causes the solver to run out of memory, then we retry it once more, but with a split size of 150 mutants instead.

Due to the semantic differences between regular and simultaneous mutation testing, as explained in section 7.3, it is unfair to compare their performances directly. In particular, it would be unfair to compare the pragmatic and solver runs' performance with the performance of the baseline directly. Instead, the

verification run's performance is compared to the baseline' performance, allowing us to determine the introduced overhead. The pragmatic and solver runs' performance are compared to the verification run's performance.

Stryker uses configuration files that allows us to have some control over what it does exactly. This includes options that may have an impact on the performance of Stryker. As such, all test subjects will make use of the same configuration for options that may impact performance or quality. The initial configuration for all stages is as follows:

- `enableSimultaneousTesting: false`. Initial run will not use simultaneous testing. The results of this first run will be used as a baseline.
- `disableBail: false`. Bail will be enabled for all runs (may they be simultaneous or not).
- `ignoreStatic: true`. Static mutants will be ignored since coverage data is inaccurate for these mutants and because they require a full reset of the environment (meaning that simultaneous testing cannot improve the performance for static mutants anyway).
- `measureMetrics: true`. For performance validation it is necessary to measure the duration of operations. Should have little to no impact on the performance. Additionally, the amount of performance measurements done is similar between regular and simultaneous mutation testing.
- `concurrency: 1`. This setting sets the maximum amount of concurrent test-runners that StrykerJS may use. For consistent measurements it is required that not too many things are working at the same time.

  In (technical) detail: JavaScript is single-threaded by nature and the main process of StrykerJS measures almost everything. In the case that the main process has a lot of other work to do, like processing the updates sent by the child-processes during live reporting (see section 7.3), measurements for functions that return a promise will not be accurate. To reduce these inaccuracies, the workload of the main process should be kept to a minimum.
- `checkers: [ ]`. No checkers. StrykerJS supports checkers as plugins, such as the TypeScript checker. It is not required and not including checkers would lead to more mutants that can be tested. See section 5.3 for more details on the checker phase.
- `logLevel: info`. The log levels debug and trace would have too much of an impact on performance. Additionally, non-simultaneous and simultaneous testing have vastly different amount of logging which would mean that a comparison between the two is not fair.
- `fileLogLevel: off`. Same reasons as with logLevel. Initially it will be set to off. If the console shows errors/warning it will be enabled and rerun for further investigation (manually).

- `coverageAnalysis: perTest`. This coverage analysis strategy is the most accurate, which is necessary for grouping disjoint mutants.
- `testRunner: mocha`. Simultaneous testing is only implemented for Mocha, hence required.
- `reporters: [json, progress]`. The json reporter exports the result of the mutation run to a file, which is necessary for validating the quality. The progress reporter simply provides updates on the progress.
- `plugins: [/stryker-js/packages/mocha-runner/dist/src/index.js]`. The mocha test-runner to use. Contains the implementation for simultaneous testing.

Options not listed here will either take the default values of Stryker or be set by the (original) configuration of the test subject. To run Stryker on a test subject, the command 'npm run stryker' will be used. Variations on the default configuration will be effected by modifying the Stryker run command on the command line. Command line arguments may override values from the initial configuration file just provided. See section 7.5 for a detailed description of the commands used. For each stage, the following additional command line arguments will be provided:

1. Baseline: no changes need to be made, the default configuration is specifically built for the baseline.
2. Verification: command is extended with `--enableSimultaneousTesting --maximumGroupSize 1`.
3. Pragmatic: command is extended with `--enableSimultaneousTesting`.
4. Solver: command is extended with `--enableSimultaneousTesting --importMutantGroups`.

To automate the process of running all the aforementioned stages on a test subject, three bash scripts were created, namely: automate-stryker.sh, automate-solver.sh and automate-all.sh. The scripts used can be found in Appendix B. The first script, automate-stryker (section B.1), handles running the aforementioned Stryker commands on the command line 5 times for each stage for every test subject. Meaning that there will be a total of 20 runs for each test subject. The second script, automate-solver (section B.2), first runs the Stryker command extended with `--exportMutantsOnly` to export the mutants to a file (skips the entire mutation testing phase), which is then fed into the Java solver as input to form simultaneous mutant groups. The last script, automate-both (section B.3), makes use of both automate-stryker and automate-solver to fully automate the process. The test subjects are hard-coded in this script. Additionally, the test subject typescript-checker will be run by this script with a concurrency of 4, as opposed to a concurrency of 1 like the other test subjects,

OS Version	Windows 11 Enterprise, 22H2
Processor	12th Gen Intel(R) Core(TM) i7-12800H 2.40 GHz
Installed RAM	32,0 GB (31,7 GB usable)
System type	64-bit operating system, x64-based processor

Table 9.1: Host machine specification.

OS Version	Ubuntu 22.04.3
available RAM	15GB (+4.1GB swap)
Kernel release	5.15.146.1-microsoft-standard-WSL2

Table 9.2: WSL specification.

because this particular test subject would run for over 10 minutes for a single Stryker run with a concurrency of 1.

The experiment will be performed on a Windows 11 laptop in the Windows Subsystem for Linux (WSL). See tables 9.1 and 9.2 for the specifications. Additionally, the laptop's security will be disabled entirely and airplane mode will be enabled during the experiment.

Chapter 10

Results

A lot of data has been captured and the result will be presented in this chapter. This will be done in the same order as the research questions defined in section 4.3. The last section (see section 10.4) summarises the findings and the conclusions.

10.1 Performance

An important question to ask is whether simultaneous testing has better performance compared to regular mutation testing. For this reason, the durations of certain operations have been captured. Table 10.1 shows some performance statistics from the baseline runs, namely: the total number of test sessions (#test-sessions), the total time spent creating test sessions (T_{CTS}), the total time spent running tests (T_{tests}) and the total time spent in all test sessions ($T_{session}$, which includes T_{CTS}, T_{tests} and the time spent formulating a result). The change in percentages (Δ) of these statistics of the verification run compared to the baseline run are shown in Table 10.2. In particular, the deltas provide an indication of the introduced overhead by simultaneous mutation testing. The last column $\Delta T_{session}$ shows a clear overhead of at least 50%. Most of the overhead comes from the increase in time spent running tests, which is the result of the necessary feature live reporting (see section 7.3).

Table 10.3 shows the derivations of change in percentages for all test subjects for the groups formed by the pragmatic (signified by the **P** sub-columns) and solver (signified by the **S** sub-columns) algorithm compared to the verification run. The last row shows the average of the values in the rows above. From the last column, we may conclude that simultaneous testing for StrykerJS is not really useful. It only shows a small increase in performance of 2.9% on average for the pragmatic runs and it even shows a slowdown of 0.2% on average for the solver runs.

Module	#test-sessions	T_{CTS} (ms)	T_{tests} (ms)	$T_{session}$ (ms)
cucumber-expressions	984	718	7192	8093
metrics	288	97	284	429
instrumenter	1411	1230	9247	10752
mocha-runner	250	147	13864	14055
typescript-checker	364	2736	681214	683999
typed-inject	120	48	72	140

Table 10.1: Performance statistics for each project from the baseline run (all values are averages over 5 runs).

Module	Δ#test-sessions	ΔT_{CTS}	ΔT_{tests}	$\Delta T_{session}$
cucumber-expressions	0.0%	-0.7%	64.9%	55.6%
metrics	0.0%	18.8%	117.8%	72.1%
instrumenter	0.0%	-0.02%	61.5%	50.5%
mocha-runner	0.0%	167.7%	157.1%	176.0%
typescript-checker	0.0%	14.0%	52.2%	52.9%
typed-inject	0.0%	-2.5%	156.1%	67.2%

Table 10.2: Change in performance statistics of the verification runs compared to the baseline (all values are derived from averages over 5 runs).

	Δ#test-sessions		ΔT_{CTS}		ΔT_{tests}		$\Delta T_{session}$	
Module	**P**	**S**	**P**	**S**	**P**	**S**	**P**	**S**
cucumber-expressions	-33.2%	-28.4%	-9.0%	37.7%	-0.2%	6.0%	-0.7%	7.8%
metrics	-42.0%	-41.7%	-30.6%	-20.1%	-2.9%	-4.9%	-7.3%	-7.3%
instrumenter	-80.9%	-55.6%	-42.4%	-8.9%	-0.7%	2.9%	-3.9%	1.9%
mocha-runner	-76.0	-60.4%	-11.6%	-6.3%	0.2%	0.5%	-0.03%	0.5%
typescript-checker	-26.6%	-31.9%	5.3%	8.4%	2.4%	5.8%	2.5%	5.8%
typed-inject	-35.0	-44.2%	-13.2%	-9.0%	-5.7%	-6.3%	-7.8%	-7.5%
averaged	-49.0%	-43.7%	-16.9%	0.3%	-1.2%	0.7%	-2.9%	0.2%

Table 10.3: Change in statistics of the pragmatic and solver runs compared to the verification run (all values are derived from averages over 5 runs).

Module	Proportion p	avg($\|g\|$)		Maximum gain	
		P	**S**	**P**	**S**
cucumber-expressions	0.06	1.50	1.40	1.9%	1.6%
metrics	0.16	1.72	1.71	7.0%	7.0%
instrumenter	0.08	5.23	2.25	6.5%	4.4%
mocha-runner	0.01	4.17	2.60	0.8%	0.6%
typescript-checker	0.003	1.48	1.48	0.1%	0.1%
typed-inject	0.20	1.54	1.79	7.5%	9.7%

Table 10.4: The theoretical maximum performance gain per project per grouping algorithm.

Overall the small difference in performance, as opposed to the huge decrease in the number of test sessions, which is about 49% and 44% for the pragmatic and solver algorithm respectively, can be explained by the fact that the time spent creating test sessions is relatively small. Table 10.4 shows the maximum performance gain for all test subjects for both grouping algorithms. Following Amdahl's law, the maximum gain is computed from the proportion and the average group size as explained in section 5.2.3. Note that the proportions are derived from the test subjects' verification runs. This table indicates that the maximum gain from simultaneous testing is only about 4% on average, which is not that significant. Simultaneous mutation testing improves on having to create fewer test sessions as shown in the theory. But for StrykerJS specifically, creating test sessions simply does not take that much time. Hence, the benefit of simultaneous testing is small or even non-existent.

An interesting observation that can be made when comparing the actual change in performance (see Table 10.3) with the maximum performance gain (see Table 10.4 is that for some test subjects the actual gain exceeds the theoretically calculated maximum gain. The test subjects in question are metrics and typed-inject. However, this is not due to the reduction of test sessions but it has to do with the reduction in time spent running tests. Why the time spent running tests changes for any of the test subjects is not exactly clear, but it could be caused by changes to the order in which the tests are executed (see section 8.3). It could also be caused by JavaScript's JIT compiler, which optimises pieces of code that are executed often, which is more likely to happen when sessions run for a longer period due to having to test more mutants within one session.

A small follow-up experiment was set up to prove the effectiveness of simultaneous mutation testing and the impact on performance by the duration of creating test sessions. In this experiment, only the cucumber-expressions test subject was used. In addition to the command line arguments as described in chapter 9, `--fakeTestSessionCreationDuration 100` was added. This argument adds a delay of 100 milliseconds to creating a test session. The results of this run are shown, in a similar fashion as before, in Table 10.5, Table 10.6 and Table 10.7.

Type	**#test-sessions**	T_{CTS} **(ms)**	T_{tests} **(ms)**	$T_{session}$ **(ms)**
baseline	984	101335	9013	110743

Table 10.5: Performance statistics for the cucumber-expressions test subject's baseline run with an artificial delay of 100ms to creating a test session (average over 5 runs).

Type	**Δ#test-sessions**	ΔT_{CTS}	ΔT_{tests}	$\Delta T_{session}$
verification	0.0%	-0.1%	55.6%	4.1%

Table 10.6: Change in performance statistics for the cucumber-expressions test subject's verification run compared to the baseline run with an artificial delay of 100ms to creating a test session (all values are derived from averages over 5 runs).

The results in Table 10.7 show a performance increase for the groups formed by both the pragmatic and solver algorithms. Using the pragmatic algorithm improves performance by 30% when compared to the verification run (27% compared to baseline). Using the solver algorithm improves performance by 24% when compared to the verification run (21% compared to baseline). Even with the overhead introduced by live reporting, simultaneous testing still performs better. Do note that in this particular scenario, the time spent creating test sessions is about 90% of the sum of the durations spent in all test sessions.

Change in performance compared to baseline run:				
Type	**Δ#test-sessions**	ΔT_{CTS}	ΔT_{tests}	$\Delta T_{session}$
pragmatic	-33.2%	-33.0%	47.5%	-26.7%
solver	-28.4%	-27.3%	47.7%	-21.4%

Change in performance compared to verification run:				
Type	**Δ#test-sessions**	ΔT_{CTS}	ΔT_{tests}	$\Delta T_{session}$
pragmatic	-33.2%	-32.9%	-5.2%	-29.6%
solver	-28.4%	-27.2%	-5.1%	-24.5%

Table 10.7: Change in performance statistics for the cucumber-expressions test subject's pragmatic and solver run compared to the baseline and verification run with an artificial delay of 100ms to creating a test session (all values are derived from averages over 5 runs).

10.2 Quality

Another important question to ask is whether simultaneous mutation testing produces the same results as regular mutation testing, as discussed by the second research question (see section 4.3). Recall from section 8.3 that we do not require the results of each mutant to be equivalent to the baseline but that we also accept slightly different outputs for certain mutants, due to test execution order, as indicated by the status mapping in Table 8.2. Table 10.8 shows an overview of the amount of (non-acceptable) mismatched results produced on average during simultaneous testing for the verification, pragmatic and solver runs compared to the baseline. As can be seen in this table, there are some non-acceptable mismatched results. These mismatched results are caused by the test subjects instrumenter, mocha-runner and typescript-checker. The mismatched results of these test subjects are shown in tables 10.9, 10.10 and 10.11 respectively. The other test subjects (cucumber-expressions, metrics and typed-inject) did not have any mismatched results.

The first thing to note is that the verification run produces non-acceptable mismatched results for the mocha-runner and typescript-checker test subjects. That could mean that either the implementation of simultaneous testing is incorrect or that the original implementation of regular mutation testing is incorrect (or has become incorrect during the support of simultaneous testing). It could even mean that (some of) the test suites used by the test subjects are flawed in the sense that consecutive test runs in a test-runner yields different results. More specifically, most of the non-acceptable mismatched results were false positives, meaning that regular mutation testing killed the mutants whereas the verification run deemed the same mutants to be a survivor. Even more so, for verification purposes, one of the mismatched mutants was run in isolation during regular mutation testing, which then also indicated that mutant to be a survivor.

A second thing to note is that the verification run of the typescript-checker test subject shows a decimal value, which indicates that the verification run produces different results for consecutive runs with the exact same input. This strengthens the suspicion that consecutive mutant runs in a single test-runner may influence the results of each other, which is a problem that exists within

Type	**#misses**	**%misses**	**#non-acceptable misses**	**%non-acceptable misses**
verification	3.5	< 0.01	2.5	< 0.01
pragmatic	4.5	< 0.01	3.3	< 0.01
solver	2.2	< 0.01	1.2	< 0.01

Table 10.8: Overview of the average amount of mismatched results and average percentage of (non-acceptable) mismatched results over the total number of mutants for each run.

Project	**Type**	**#misses**	**%misses**	**#non-acceptable misses**	**%non-acceptable misses**
instrumenter	verification	0.0	0.0	0.0	0.0
instrumenter	pragmatic	6.0	< 0.01	6.0	< 0.01
instrumenter	solver	1.0	$<< 0.01$	1.0	$<< 0.01$

Table 10.9: Mismatched results for the instrumenter test subject compared to the baseline results.

Project	**Type**	**#misses**	**%misses**	**#non-acceptable misses**	**%non-acceptable misses**
mocha-runner	verification	5.0	0.01	1.0	< 0.01
mocha-runner	pragmatic	5.0	0.01	1.0	< 0.01
mocha-runner	solver	4.0	0.01	0.0	0.0

Table 10.10: Mismatched results for the mocha-runner test subject compared to the baseline results.

StrykerJS and is not a problem caused by simultaneous mutation testing. No further investigations have been done into this problem.

The results of both the mocha-runner and typescript-checker test subjects will be excluded from analysis for the remainder of this section due to the issues with the verification run. Overall the results indicate almost no difference in terms of results. In particular, the test subjects typed-inject, metrics and cucumber-expressions have the same results compared to regular mutation testing. For the instrumenter project, the pragmatic run had a non-acceptable mismatch of 6 whereas the solver run had a non-acceptable mismatch of 1. That is a mismatch of less than 0.01% over all mutants. Of the mismatches in the pragmatic run, two of them were false negatives (mutants that were survivors in the baseline but killed by simultaneous testing) and four of them were false positives (mutants that were killed in the baseline but survived during simultaneous mutation testing).

As shortly discussed in research question 2 (see section 4.3), it was expected that simultaneous testing would cause for both false positives and false negatives. This is due to the fact that a mutant is able to change the control flow of a program completely, meaning that it is possible that two simultaneous mutants that were initially disjoint in terms of test coverage actually have overlapping tests when enabled simultaneously. Due to the high mutation scores of the test subjects (see Table 8.1), it is likely that we find more false positives than false negatives because there are a relatively small number of non-equivalent mutants that survive, meaning that false negatives are less likely to happen. Although the results also seem to lean towards that conclusion, it is hard to make a definitive conclusion because of the limited number of mismatched results. Further investigation is necessary using test subjects with lower mutation scores.

Project	**Type**	**#misses**	**%misses**	**#non-acceptable misses**	**%non-acceptable misses**
typescript-checker	verification	16.2	0.04	14.2	0.04
typescript-checker	pragmatic	16.0	0.04	13.0	0.03
typescript-checker	solver	8.0	0.02	6.0	0.01

Table 10.11: Mismatched results for the typescript-checker test subject compared to the baseline results.

	$\lvert M\rvert$	**#groups**		**#simultaneous mutants**		**avg($\lvert g\rvert$)**		**min($\lvert g\rvert$)**	**max($\lvert g\rvert$)**		**Duration (ms)**	
Project		**P**	**S**	**P**	**S**	**P**	**S**		**P**	**S**	**P**	**S**
cucumber-expressions	984	125	196	452	475	3.616	2.423	2	39	10	24.4	371991
metrics	288	39	90	160	210	4.103	2.334	2	12	5	7.8	97977
instrumenter	1411	259	371	1400	1156	5.405	3.116	2	84	12	54.8	336407
mocha-runner	250	55	69	245	223	4.455	3.232	2	23	9	2.6	83926
typescript-checker	364	65	87	183	205	2.815	2.356	2	19	8	10.4	107940
typed-inject	120	29	46	71	99	2.448	2.152	2	7	3	2.0	93010

Table 10.12: Statistics of *non-singleton* simultaneous mutation groups formed by the pragmatic and solver algorithm.

10.3 Strategy for Grouping Mutants

The third research question we wish to answer is: What is the best strategy for grouping mutants together in terms of performance? In order to answer this question properly, some statistics were gathered on the groups that were formed by the pragmatic and solver algorithm. The statistics for each test subject are shown in Table 10.12 and includes the following: the total number of generated mutants ($\lvert M\rvert$), the number of (non-singleton) groups formed (#groups), the number of (non-singleton) simultaneous mutants (#simultaneous mutants), the average (avg($\lvert g\rvert$)), minimum (min($\lvert g\rvert$)) and maximum (max($\lvert g\rvert$)) size of the (non-singleton) simultaneous groups formed and the time it takes for the algorithm to form the groups (duration). The sub-columns P and S indicate whether the value relates to the pragmatic or solver algorithm respectively. If there are no sub-columns, then the results were equivalent for both algorithms. As stated in section 6.6.7, please note that the solver algorithm is constrained by time and memory, meaning that it may not perform well on large inputs.

The pragmatic algorithm is quite efficient in forming mutation groups compared to the solver. The maximum time spent forming groups by the pragmatic algorithm is 55 milliseconds for the instrumenter test subject, while the *minimum* time spent by the solver algorithm was almost 84 seconds for the mocha-runner test subject. In fact, just creating the constraints in the solver takes longer than the time it takes for the pragmatic algorithm to form groups.

The results also show that the pragmatic algorithm tends to create fewer (and

Project	Pragmatic reruns	Solver reruns
cucumber-expressions	0	0
metrics	0	0
instrumenter	0	0
mocha-runner	0	3
typescript-checker	21	2
typed-inject	0	0

Table 10.13: Number of reruns for each test subject per grouping algorithm.

therefore also larger) groups, this is true for all test subjects (see #groups and avg($|g|$). Since the main objective of the algorithms was to create as few groups as possible (see section 6.2), we may conclude that the groups formed by the pragmatic algorithm are superior to the groups formed by the solver. This is supported by the performance tables 10.3 and 10.4.

Another aspect to forming groups to take into account is the consequence of mutants that will cause a timeout. If the group leads to a timeout before all simultaneous mutants within that group have a properly defined result, then the mutation testing tool needs to rerun all mutants that do not have a properly defined result yet individually. The odds that a group will cause a timeout grows with the size of the group as the odds of it containing at least 1 mutant that causes a timeout increases. This would mean that since the pragmatic algorithm produces larger groups, it is expected that it also requires more reruns. Even more so, the sizes of the groups formed by the pragmatic algorithm varies drastically as the column on maximum size of non-singleton mutation groups indicate (max($|g|$)). For example, the pragmatic algorithm formed a group of 84 mutants in the instrumenter test subject, whereas the solver algorithm's largest group is only 12. Table 10.13 shows the number of reruns for the pragmatic and solver algorithm for each test subject. The impact of this is especially noticeable in the typescript-checker test subject. Here, the pragmatic run required 21 reruns, whereas the solver run only needed 2 reruns. However, due to the limited number of cases where reruns occur, no definitive conclusion can be made on whether it is desirable to have relatively large or small groups.

10.4 Summary of Findings

To summarise, simultaneous mutation testing for StrykerJS introduces an overhead of at least 50% due to the necessary feature live reporting. If we were to ignore the impact of the overhead, then simultaneous mutation testing increases the performance by 2.9% on average (when using the pragmatic grouping algorithm). The time spent creating test session in StrykerJS is not that high, meaning that there is little benefit to performing simultaneous mutation testing. The follow-up experiment shows that simultaneous testing can improve the performance by almost 27% under certain conditions, even with the over-

head introduced by simultaneous testing. Additionally, simultaneous testing produces the same results as regular mutation testing. Only 0.01% of the mutants that have been tested produce a different result. Concerning the grouping algorithms, the solver algorithm is in all ways inferior to the pragmatic algorithm. The pragmatic algorithm is more efficient in terms of time spent forming the groups and it also produces a smaller set of simultaneous mutant groups.

Chapter 11

Discussion

In the upcoming sections, we will examine potential threats to the research's validity, explore potential directions for future research, and ultimately wrap up the report with a recommendation, followed by a conclusion.

11.1 Threats to Validity

In this section we will discuss potential threats to the validity of the research. Threats to internal validity may have an impact on the conclusion drawn from the observed results due to the variables that have been changed. It is mainly concerned with whether the observed results are truly caused by the presumably independently changed variables. It is also concerned with whether the results are consistent as to enhance the reproducibility of the research. Threats to external validity are mainly concerned with the applicability of the research's results in the real world.

11.1.1 Threats to Internal Validity

- **Memory limit of group solver**: As stated in section 6.6.7, the solver algorithm runs out of memory when attempting to solve for 'large' inputs. This meant that we had to split the input into multiple parts. The splits themselves have a huge impact on the groups that are being formed. This has the consequence that comparing the pragmatic and solver algorithm's results is unfair. Although attempts were made to decrease the impact of the lack of memory available (by increasing the amount of memory the solver may use), it was not sufficient for the solver to really compete against the pragmatic algorithm.
- **Inconsistent performance metrics measurements**: Because computers do many things in parallel, it is possible that two executions of the

same program measure vastly different durations for the same operations. To minimise this threat, the average was taken of five iterations of the same program. Additionally, the device on which the experiment was performed had its security disabled and airplane mode enabled. Furthermore, the concurrency configuration option in Stryker was set to 1 to prevent overloading the main process, as explained in the description of configuration options for Stryker in chapter 9.

11.1.2 Threats to External Validity

- **High quality test subjects**: The test subjects used during the experiment all have adequate test suites as the mutation scores indicate. It could be the case that the results of the experiment are only representative for other projects in the real world with high quality test suites. To be more precise, it is possible that the quality of simultaneous testing is impacted more than the 0.01% found in the experiments. The last paragraph in section 10.2 touches on this issue briefly.
- **Restricted to StrykerJS**: We implemented simultaneous testing in the mutation testing tool StrykerJS. This means that the observed results, most notably the observed change in performance, may not be the same for other mutation testing tools. The potential gain of simultaneous mutation testing heavily depends on the time it takes to create a test session, which is different for every mutation testing tool.
- **Restricted to JavaScript and friends**: StrykerJS is a source-code mutator for JavaScript, TypeScript and alike. As such, the projects used during the experiments are written in JavaScript/TypeScript. The types of mutations generated in these projects may differ drastically in frequency between other programming languages. This indirectly influences the types of mutations that are grouped to be run simultaneously, which may influence performance and quality.

11.2 Future Work

In this section we discuss potential starting points for further research on the topic of simultaneous testing.

11.2.1 Timeouts and Simultaneous Testing

As explained in section 7.3, when a group of mutants times out, all the mutants within that group that do not have a properly defined result yet need to be rerun individually. This partially nullifies the benefit of simultaneous mutation testing, which is undesired. On the one hand, larger groups will impact performance more on timeout than smaller groups. On the other hand, smaller groups provide less of a performance gain in the first place. A more fine-grained theory

on the cost of simultaneous mutation testing (see section 5.2) that includes the effects of timeouts more precisely should help with determining how to deal with timeouts within mutation groups.

Additionally, the odds of a group timing out increases as the size of the group increases when assuming that every mutant is just as likely to timeout as any other. For example, consider the groups $G_1 = \{\{m_1, t, m_2, m_3\}\}$ and $G_2 = \{\{m_1, t\}, \{m_2, m_3\}\}$, where m_1, m_2 and m_3 are mutants that will *not* time out and t is a mutant that *will* timeout. In this example, group G_1 will perform worse than G_2 even though $|G_1| < |G_2|$. This is because G_1 will require 3 test sessions when accounting for the additional test sessions created to rerun m_1 and m_2, whereas G_2 only requires 2 because it does not need to rerun any mutants. To combat this issue, an investigation into the odds of a mutant timing out is necessary. More specifically, it could be the case that certain types of mutants produced by certain mutation operators have the tendency to time out more often than others. From the results, one could decide to not group any mutants with any other mutants that are likely to time out.

11.2.2 Simultaneous Mutation Testing for Other StrykerJS Test-runners

Simultaneous mutation testing has been implemented for the mocha test-runner specifically. It is possible that other test-runners spend more time creating test sessions, which could mean that these test-runners may see a performance increase if simultaneous mutation testing was implemented for them. Before implementing simultaneous mutation testing for any other test-runner, it is advised that developers measure the average time it takes to create test sessions first to decide whether it is beneficial.

11.2.3 Smart Mutation Switching

The current implementation used for simultaneous testing in the mocha test-runner will enable all mutants from a group simultaneously at all times. Interestingly enough, Mocha's application programming interface would allow us to switch (enable/disable) mutants in between tests. This should improve the quality of simultaneous mutation testing since the simultaneous mutants would be guaranteed to have no impact on each other. This should not have an impact on the performance.

11.2.4 Comparison with Stryker.NET

A comparison in performance and quality with Stryker.NET could be interesting. It may provide more insights into why simultaneous mutation testing works better or worse in other programming languages and/or other mutation testing tools.

11.3 Recommendation

For StrykerJS in particular, we recommend *not* implementing simultaneous mutation testing, at least not for the Mocha test-runner. The overhead introduced by simultaneous mutation testing simply outweighs the benefit of reducing the number of test sessions required. However, it is possible that simultaneous testing would perform better on other test-runners that need more time to create test sessions as discussed in section 11.2.2.

In general, simultaneous mutation testing can be a good optimisation if the time it takes to create test sessions is relatively large as was shown during the follow-up experiment. In order to implement simultaneous testing, it is required that the optimisations coverage analysis and mutant schemata have been implemented. Developers of mutation testing tools should first investigate the time it takes to create test sessions before considering to implement simultaneous mutation testing as to figure out whether simultaneous testing would prove beneficial. Amdahl's law can be used to determine the maximum possible performance gain once the proportion of time spent creating test sessions has been measured[1].

11.4 Conclusion

In this thesis we have investigated a novel approach to decrease the cost of mutation testing, called simultaneous mutation testing. The theory presented is used as a means to explain why mutation testing is costly in general, including the impact of certain optimisations, while also showing where the performance gain from simultaneous mutation testing should come from. Additionally, two algorithms are provided that are capable of forming acceptable mutant groups to be used during simultaneous testing.

The theory has been validated through a controlled experiment and shows that simultaneous mutation is effective when the average time spent creating test sessions is higher than the overhead introduced by simultaneous mutation testing. Although simultaneous testing could only improve the performance of StrykerJS by 3%, the follow-up experiment has shown that simultaneous testing can reduce the total time it takes to perform mutation testing up to 27%. Compared to many other optimisations, such as higher-order mutation, simultaneous mutation testing improves performance while also retaining the quality. In fact, less than 0.01% of the mutants had mismatched results compared to regular mutation testing. It is advised that developers measure the time it takes to create test sessions first before considering to implement it as the maximum possible performance gain relies entirely on that metric.

[1]For computing the maximum possible performance gain, one can let $|g| \to \infty$.

Bibliography

[1] M. A. Cachia, M. Micallef, and C. Colombo. Towards incremental mutation testing. *Electronic Notes in Theoretical Computer Science*, 294:2–11, Mar. 2013.

[2] M. Delamaro and J. Maldonado. Interface mutation: assessing testing quality at interprocedural level. In *Proceedings. SCCC'99 XIX International Conference of the Chilean Computer Science Society*. IEEE Comput. Soc, 1999.

[3] M. Delamaro, J. Maldonado, and A. Mathur. Integration testing using interface mutation. In *Proceedings of ISSRE '96: 7th International Symposium on Software Reliability Engineering*. IEEE Comput. Soc. Press, 1996.

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

[5] R. Demillo and E. Spafford. The mothra software testing environment, 01 1987.

[6] A. S. Ghiduk, M. R. Girgis, and M. H. Shehata. Higher order mutation testing: A systematic literature review. *Computer Science Review*, 25:29–48, Aug. 2017.

[7] A. Gillies. *Software quality: Theory and management (3rd edition)*. Lulu.com, Barking, England, Jan. 2011.

[8] Google. Or-tools | google for developers. `https://developers.google.com/optimization`. [Online; accessed January 29th 2024].

[9] W. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, SE-8(4):371–379, July 1982.

[10] D. Jackson and M. R. Woodward. *Parallel Firm Mutation of Java Programs*, page 55–61. Springer US, 2001.

[11] P. Jalote. *An Integrated Approach to Software Engineering*. Springer New York, 1997.

[12] Y. Jia and M. Harman. Constructing subtle faults using higher order mutation testing. In *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 249–258, 2008.

[13] Y. Jia and M. Harman. Higher order mutation testing. *Information and Software Technology*, 51(10):1379–1393, Oct. 2009.

[14] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 654–665, New York, NY, USA, 2014. Association for Computing Machinery.

[15] G. Kaminski, G. Williams, and P. Ammann. Reconciling perspectives of software logic testing. *Software Testing, Verification and Reliability*, 18(3):149–188, Sept. 2008.

[16] M. Kintis, M. Papadakis, and N. Malevris. Evaluating mutation testing alternatives: A collateral experiment. In *2010 Asia Pacific Software Engineering Conference*. IEEE, Nov. 2010.

[17] M. Kintis, M. Papadakis, and N. Malevris. Isolating first order equivalent mutants via second order mutation. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, Apr. 2012.

[18] 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*. IEEE, Mar. 2014.

[19] B. Kurtz, P. Ammann, and J. Offutt. Static analysis of mutant subsumption. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, Apr. 2015.

[20] W. B. Langdon, M. Harman, and Y. Jia. Efficient multi-objective higher order mutation testing with genetic programming. *Journal of Systems and Software*, 83(12):2416–2430, Dec. 2010.

[21] C. Y. Laporte and A. April. *Software Quality Assurance*. John Wiley & Sons, Nashville, TN, Dec. 2017.

[22] J. Lee, S. Kang, and P. Jung. Test coverage criteria for software product line testing: Systematic literature review. *Information and Software Technology*, 122:106272, June 2020.

[23] Y.-S. Ma, J. Offutt, and Y.-R. Kwon. MuJava. In *Proceedings of the 28th international conference on Software engineering*. ACM, May 2006.

[24] P. R. Mateo and M. P. Usaola. Parallel mutation testing. *Software Testing, Verification and Reliability*, 23(4):315–350, Mar. 2012.

[25] Q. V. Nguyen and L. Madeyski. Addressing mutation testing problems by applying multi-objective optimization algorithms and higher order mutation. *Journal of Intelligent & Fuzzy Systems*, 32(2):1173–1182, Jan. 2017.

[26] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.*, 5(2):99–118, apr 1996.

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

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

[29] A. J. Offutt and R. H. Untch. *Mutation 2000: Uniting the Orthogonal*, pages 34–44. Springer US, Boston, MA, 2001.

[30] M. Papadakis, Y. Jia, M. Harman, and Y. L. Traon. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. IEEE, May 2015.

[31] M. Papadakis and N. Malevris. An empirical evaluation of the first and second order mutation testing strategies. In *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*. IEEE, Apr. 2010.

[32] A. Parsai and S. Demeyer. Comparing mutation coverage against branch coverage in an industrial setting. *International Journal on Software Tools for Technology Transfer*, 22(4):365–388, May 2020.

[33] Pitest. Java mutation testing systems. `https://pitest.org/java_mutation_testing_systems/`. [Online; accessed October 3rd 2023].

[34] R. Pitts. Mutant selection strategies in mutation testing. In *2023 International Conference on Code Quality (ICCQ)*. IEEE, Apr. 2023.

[35] A. V. Pizzoleto, F. C. 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*, 157:110388, 2019.

[36] M. Polo, M. Piattini, and I. García-Rodríguez. Decreasing the cost of mutation testing with second-order mutants. *Software Testing, Verification and Reliability*, 19(2):111–131, June 2009.

[37] M. Reddy. Chapter 7 - performance. In M. Reddy, editor, *API Design for C++*, pages 209–240. Morgan Kaufmann, Boston, 2011.

[38] D. Schuler and A. Zeller. (un-)covering equivalent mutants. In *2010 Third International Conference on Software Testing, Verification and Validation*. IEEE, 2010.

[39] J. Smits. Callisto - selecting effective mutation operators for mutation testing, 2022.

[40] Stryker. Stryker mutator. `https://stryker-mutator.io/`. [Online; accessed October 5th 2023].

[41] Stryker. Supported mutators: Stryker mutator. `https://stryker-mutator.io/docs/mutation-testing-elements/supported-mutators/`. [Online; accessed October 5th 2023].

[42] Stryker. Strykerjs github. `https://github.com/stryker-mutator/stryker-js`, 2016. [Online; accessed October 5th 2023].

[43] Stryker. Stryker github. `https://github.com/stryker-mutator`, 2018. [Online; accessed October 5th 2023].

[44] Stryker. Stryker4s github. `https://github.com/stryker-mutator/stryker4s`, 2018. [Online; accessed October 5th 2023].

[45] Stryker. Stryker.net github. `https://github.com/stryker-mutator/stryker-net`, 2018. [Online; accessed October 5th 2023].

[46] R. H. Untch, A. J. Offutt, and M. J. Harrold. Mutation analysis using mutant schemata. In *Proceedings of the 1993 ACM SIGSOFT international symposium on Software testing and analysis*. ACM, July 1993.

[47] C. Wimmer, C. Stancu, P. Hofer, V. Jovanovic, P. Wögerer, P. B. Kessler, O. Pliss, and T. Würthinger. Initialize once, start fast: application initialization at build time. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29, Oct. 2019.

[48] L. Zhang, D. Marinov, and S. Khurshid. Faster mutation testing inspired by test prioritization and reduction. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM, July 2013.

[49] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid. Regression mutation testing. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ACM, July 2012.

Appendix A

Google Form

Faster Mutation Testing with StrykerJS

In order to validate the optimisation being made, it is required that it is tested on real-world applications that make use of Stryker. This form tries to capture the most important information of projects.

Sign in to Google to save your progress. Learn more

* Mandatory question

Link to project on Github *

The link to your project on Github (or other version control websites that work with git). May refer to a different branch or specific commit but it should be stable.

Number of mutants *

The total number of mutants that are generated and tested.

Lines of source code

Total number of lines of source code in the project. Excluding comments, blank lines and code that is not mutated (like html, the body of script elements within the html should be included).

What is your (current) mutation score (in %)? *

Which test-runners does your project use? *

It could be the case that projects use different test-runners for different modules/packages. You can select all test-runners used here.

- [ ] Cucumber
- [ ] Jasmine
- [ ] Jest
- [ ] Karma
- [ ] Mocha
- [ ] Tap
- [ ] Vitest
- [ ] Command runner

Which test-runner does your project use **mainly**? *

The main test-runner would be considered the one that tests the largest proportion of the code base.

Choose

Remarks

Here you can put some remarks about your project (setup). Such as: which test-runner is used for which module/package (when using more than 1 test-runner), and whether you have disabled specific mutation operators through 'excludedMutations' in the configuration.

Send | Page 1 of 1 | Clear form

Never send passwords through Google Forms.

This content is not created or approved by Google. Report Abuse - Terms of Service - Privacy Policy

Google Forms

Appendix B

Automated Scripts

B.1 automate-stryker.sh

```
#!/bin/bash

ARGC=$#
DIR=$(pwd)

if [ $ARGC -lt 1 ]
then
    echo "missing source/project directory";
    exit 10;
fi

if [ $ARGC -lt 2 ]
then
    echo "Missing output directory where the results should be stored";
    exit 10;
fi

SRC=$(realpath "$1")
OUTPUT=$(realpath -m "$2")
echo "source: $SRC"
echo "target: $OUTPUT"

execute_solver=false
solver_input="undefined"

if [ $ARGC -lt 3 ]
then
```

```
28      echo "Missing third argument for --importMutantGroupsFile, if intentional you can ignore
            this message. However, this does mean that we will skip the solver. Continuing after 5s
            ..."
29      sleep 5
30  else
31      if [ ! -f "$3" ]; then
32          echo "File does not exist: '$3'"
33          exit 10;
34      fi
35
36      execute_solver=true
37      solver_input="$(realpath $3)"
38  fi
39
40  CONCURRENCY=1
41
42  if [ $ARGC -ge 4 ]
43  then
44      CONCURRENCY=$4;
45  fi
46
47  echo "concurrency: $CONCURRENCY"
48
49
50  TIMEOUT="2s"
51  if [ $ARGC -ge 5 ]
52  then
53      TIMEOUT=$5;
54      echo "Timeout set to $TIMEOUT";
55  fi
56
57  if ! cd "$SRC"; then
58      exit 10;
59  fi;
60
61  if [ ! -d "$OUTPUT" ]; then
62      echo "Output directory '$OUTPUT' does not exist yet, it will be created"
63      mkdir -p "$OUTPUT"
64  fi;
65
66  sleep 3
67
68  function iterate {
69      for ((i = 1 ; i <= 5 ; i++ ));
70      do
71          local RUN_OUTPUT_DIRECTORY="${OUTPUT}/$2/run$i"
```

```
72          mkdir -p "$RUN_OUTPUT_DIRECTORY"
73          local RUN_LOG_FILE="${RUN_OUTPUT_DIRECTORY}/run.log"
74          local RUN_METRICS="${RUN_OUTPUT_DIRECTORY}/metrics.json"
75          local RUN_MUTATION="${RUN_OUTPUT_DIRECTORY}/mutation.json"
76          local FULL_COMMAND="$1 --measureMetricsOutputFile ${RUN_METRICS} --jsonReporterOptions
                .fileName ${RUN_MUTATION} --concurrency $CONCURRENCY"
77          set -o pipefail
78          #eval "$FULL_COMMAND | tee $RUN_LOG_FILE";
79          if ! eval "$FULL_COMMAND | tee $RUN_LOG_FILE"; then
80          #if ! eval "$FULL_COMMAND"; then
81              set +o pipefail
82              echo "An issue occurred executing '$FULL_COMMAND' on iteration $i. Terminating
                    script..."
83              exit 10;
84          else
85              set +o pipefail
86              echo "Iteration $i completed with command '$FULL_COMMAND', waiting '$TIMEOUT'
                    before continuing..."
87              sleep "$TIMEOUT"
88          fi;
89      done
90  }
91
92  function baseline {
93      iterate "npm run stryker --" "baseline"
94  }
95
96  function verification {
97      iterate "npm run stryker -- --enableSimultaneousTesting --maximumGroupSize 1" "
            verification"
98  }
99
100 function pragmatic {
101     iterate "npm run stryker -- --enableSimultaneousTesting" "pragmatic"
102 }
103
104 function solver {
105     iterate "npm run stryker -- --enableSimultaneousTesting --importMutantGroups --
            importMutantGroupsFile ${solver_input}" "solver"
106 }
107
108 baseline
109 echo "Finished baseline runs"
110 sleep 1
111
112 verification
```

```
113 echo "Finished verification runs"
114 sleep 1
115
116 pragmatic
117 echo "Finished pragmatic runs"
118 sleep 1
119
120 if [ "$execute_solver" = true ]; then
121     solver
122     echo "Finished solver (imported) runs"
123 else
124     echo "No solver runs were executed"
125 fi
126
127 cd $DIR
```

B.2 automate-solver.sh

```
1  #!/bin/bash
2
3  ARGC=$#
4  DIR=$(pwd)
5
6  if [ $ARGC -lt 1 ]
7  then
8      echo "missing source/project directory";
9      exit 10;
10 fi
11
12 if [ $ARGC -lt 2 ]
13 then
14     echo "Missing output directory where the results should be stored";
15     exit 10;
16 fi
17
18 if [ $ARGC -lt 3 ]
19 then
20     echo "Missing name of the to be exported file";
21     exit 10;
22 fi
23
24 if [ $ARGC -lt 4 ]
25 then
26     echo "Missing path to solver jar";
27     exit 10;
```

```
28 fi
29
30 SPLIT_SIZE=200
31 if [ $ARGC -ge 5 ]
32 then
33     SPLIT_SIZE="$5";
34 fi
35
36
37 SRC=$(realpath "$1")
38 OUTPUT_DIR=$(realpath -m "$2")
39 OUTPUT_FILE="$3"
40 SOLVER=$(realpath "$4")
41
42 echo "source: $SRC"
43 echo "output directory: $OUTPUT_DIR"
44 echo "output filename: $OUTPUT_FILE"
45 echo "solver: $SOLVER"
46 echo "splitSize: $SPLIT_SIZE"
47
48 if ! cd "$SRC"; then
49     exit 10;
50 fi;
51
52 if [ ! -d "$OUTPUT_DIR" ]; then
53     echo "Output directory '$OUTPUT_DIR' does not exist yet, it will be created"
54     mkdir -p "$OUTPUT_DIR"
55 fi;
56
57 sleep 3
58
59 EXPORTED_FILE="${OUTPUT_DIR}/${OUTPUT_FILE}"
60 if ! npm run stryker -- --exportMutantsOnly --exportMutantsFile "$EXPORTED_FILE" --
       measureMetricsOutputFile /dev/null; then
61     echo "Exporting mutants failed, terminating script...";
62     exit 10;
63 fi
64
65 echo "Mutants have been exported to ${EXPORTED_FILE}"
66
67 if ! java -Xmx8g -jar "$SOLVER" "$EXPORTED_FILE" "" $SPLIT_SIZE; then
68     echo "An issue occurred in the solver, attempting to retry with split size 150";
69     if ! java -Xmx8g -jar "$SOLVER" "$EXPORTED_FILE" "" 150; then
70         echo "An issue occurred in the solver with split size 150, terminating script...";
71         exit 10;
72     fi
```

```
73 fi
```

B.3 automate-both.sh

```
1  #!/bin/bash
2
3  TEST_SUBJECTS=(
4  "test-projects/cucumber-expressions/javascript"
5  "test-projects/mutation-testing-elements/packages/metrics"
6  "test-projects/typed-inject"
7  "test-projects/stryker-js/packages/instrumenter"
8  "test-projects/stryker-js/packages/mocha-runner"
9  "test-projects/stryker-js/packages/typescript-checker"
10 )
11
12 if [ $# -lt 1 ]
13 then
14     echo "Missing target directory where the results of the runs should be stored";
15     exit 10;
16 fi
17
18 # default to doing both
19 SHOULD_SOLVE=true;
20 SHOULD_STRYKER=true;
21
22 if [ $# -eq 2 ]
23 then
24     if [ "$2" = "solve" ]; then
25         SHOULD_STRYKER=false;
26         echo "Found 'solve' as secondary argument. This means that we will only run the solver
           . Continuing after 5s..."
27         sleep 5;
28     elif [ "$2" = "stryker" ]; then
29         SHOULD_SOLVE=false;
30         echo "Found 'stryker' as secondary argument. This means that we will only run stryker.
            Assumes solver has been used before on the same target directory (first argument)
           to properly set up the use of the stryker command. Continuing after 5s..."
31         sleep 5;
32     else
33         echo "Unknown second argument '$2', expected 'solver' or 'stryker' (second argument is
            optional)";
34         exit 10;
35     fi
36 else
```

```
37     echo "No second argument provided, will run solver and stryker consecutively on each test 
           subject. Continuing after 5s...";
38     sleep 5;
39 fi
40 
41 if [ $# -gt 2 ]
42 then
43     echo "Too many arguments were provided, expected at most 2 but got $#";
44     exit 10;
45 fi
46 
47 echo "Test subjects:"
48 for subject in "${TEST_SUBJECTS[@]}"
49 do
50   echo $subject
51   if [ ! -d "$subject" ]; then
52     echo "Test subject '$subject' is not a valid directory"
53     exit 10;
54   fi;
55 done
56 
57 EXPORTED_MUTANTS=exported-mutants.json
58 IMPORTED_MUTANTS=grouped-mutants.json
59 RESULT_DIR="$1"
60 
61 # https://stackoverflow.com/questions/8352851/shell-how-to-call-one-shell-script-from-another-
     shell-script
62 solve() {
63   current_dir=$(pwd)
64   source automate-solver.sh "$1" "$2" "$EXPORTED_MUTANTS" solver.jar
65   cd "$current_dir"
66 }
67 
68 stryker() {
69   current_dir=$(pwd)
70   # this test subject is really slow without a concurrency of 4 (~10 minutes)
71   if [ "$4" = "test-projects/stryker-js/packages/typescript-checker" ]; then
72     source automate-stryker.sh "$1" "$2" "$3" 4;
73   else
74     source automate-stryker.sh "$1" "$2" "$3" 1;
75   fi
76   cd "$current_dir"
77 }
78 
79 for subject in "${TEST_SUBJECTS[@]}"
80 do
```

```
81    current_dir=$(pwd)
82    real_subject=$(realpath "$subject")
83    OUTPUT_DIR="${RESULT_DIR}/$subject"
84    if [ "$SHOULD_SOLVE" = true ]; then
85      solve "$real_subject" "$OUTPUT_DIR"
86    fi
87
88    if [ "$SHOULD_STRYKER" = true ]; then
89      stryker "$real_subject" "$OUTPUT_DIR" "${OUTPUT_DIR}/${IMPORTED_MUTANTS}" "$subject"
90    fi
91
92    cd "$current_dir"
93  done
```