FACULTY
OF SCIENCES

Optimising Continuous Integration using Test Case Prioritisation

Pieter De Clercq
Student number: 01503338

Supervisors: Prof. dr. Bruno Volckaert, Prof. dr. ir. Filip De Turck
Counsellors: Jasper Vaneessen, Dwight Kerkhove

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in de informatica

Academic year 2019-2020

GHENT
UNIVERSITY

Admission

Acknowledgements

Completing this thesis would not have been possible without the help and support of many people, some of which I want to thank personally.

First of all I want to thank prof. dr. Bruno Volckaert and prof. dr. ir. Filip De Turck for allowing me to propose my own subject and for their prompt and clear responses to every question I have asked. I especially want to thank you for giving me the permission to insert a two-week hiatus during the Easter break, so I could help out on the UGent Dodona project.

Secondly I want to express my gratitude towards my counsellors Jasper Vaneessen and Dwight Kerkhove, for steering me into researching this topic, as well as their guidance, availability, and willingness to review every intermediary version of this thesis.

Furthermore I want to thank my parents, my brother Stijn and my family for convincing me and giving me the possibility to study at the university, to support me throughout my entire academic career and to provide me the opportunity to pursue my childhood dreams.

Last, but definitely not least, I want to thank my amazing friends, a few of them in particular. My best friend Robbe, for always being there when I need him, even when I least expect it, for both supporting as well as protecting me against my sometimes unrealistic ideas and ambition to excel. Helena for never leaving my side, for always making me laugh even when I don't want to, and most importantly to remind me that I should relax more often. Jana for my daily dose of laughter and fun, and for her inexhaustible positivity. Tobiah for the endless design discussions and for outperforming me in almost every school project, encouraging me to continuously raise the bar and never give up. Finally I want to thank Doortje and Freija for answering my mathematical questions, regularly asking about my thesis progression and thereby motivating me to persevere.

Thank you.

Pieter – Ghent, 2020

Summary

ChapterSummary Summary in English will come here.

Samenvatting

Nederlandse samenvatting komt hier.

Optimising Continuous Integration using Test Case Prioritisation

Pieter De Clercq

Supervisor(s): Prof. dr. B. Volckaert, Prof. dr. ir. F. De Turck, J. Vaneessen, D Kerkhove

Abstract—**This abstract is very abstract.**

Keywords—**words, will, appear, here, soon**

I. Introductie

Things will appear here. [1]

References

[1] Michael Cusumano, Akindutire Michael, and Stanley Smith, "Beyond the waterfall : software development at microsoft," 02 1995.

Optimaliseren van Continue Integratie door middel van Test Prioritering

Pieter De Clercq

Supervisor(s): Prof. dr. B. Volckaert, Prof. dr. ir. F. De Turck, J. Vaneessen, D Kerkhove

Abstract—**Dit abstract is super abstract.**

Trefwoorden—**woorden, komen, hier**

I. Introductie

Dingen komen hier. [1]

Referenties

[1] Michael Cusumano, Akindutire Michael, and Stanley Smith, "Beyond the waterfall : software development at microsoft," 02 1995.

Vulgarising summary

Vulgarising summary will come here.

Contents

Glossary

CI Continuous Integration. ix, 2, 11–16, 28, 32, 56

TCP Test Case Prioritisation. ix, x, 2, 16, 18, 19, 21, 23, 34, 38–40, 44, 50, 56

TCS Test Case Selection. ix, 2, 16, 18, 19, 27, 50, 56

TSM Test Suite Minimisation. ix, 2, 16–18, 20, 21, 23, 31, 37, 50, 52, 56

VCS Version Control System. 2

Chapter 1

Introduction

Given the complexity and rapid pace at which software is being built today, it is inevitable that at some point bugs will emerge. These bugs can either be introduced by a malfunctioning new feature, or by breaking existing functionality (*a regression*). In order to detect bugs in an application before its users do, an adequate *testing infrastructure* is required.

This testing infrastructure consists of multiple *test cases*, collectively referred to as the *test suite* of the application. The quality of a test suite can be assessed in multiple ways. The first and most commonly used method is to measure which fraction of the source code is tested by at least one test case, a ratio which is indicated as the *coverage* of the application. Another possibility is to apply transformations to the source code and validate whether or not this results in a failed test case, a process indicated as *mutation testing*.

Ideally, this testing process should be automated and performed after every change to the source code. This is generally very time-consuming, and as such has led to the creation of various automation frameworks and tools, collectively called Continuous Integration (CI). Common examples of CI practices are automatically running the test suite and estimating the code coverage after every pushed change to the Version Control System (VCS).

However, applying these practices and maintaining a qualitative test comes at a cost. After every addition or modification to the source code, at least one new test case must be introduced to validate its correctness. As a result of the speed at which the source code tends to grow, the test suite suffers from severe scalability issues. While it is desirable and ideally required to execute every single test case in the test suite, there are examples known to literature where this is not possible since this incurs an increasing delay in the development process, which in turn results in economic loss.

Three approaches can be taken towards resolving this issue by reducing the time waiting for the test results: Test Suite Minimisation (TSM), Test Case Selection (TCS) and Test Case Prioritisation (TCP). The main subject of this thesis will be to implement a framework for TCP. To accomplish this, the next chapter will introduce important concepts which are used in modern software engineering. Chapter 3 will elaborate on the aforementioned approaches and present accompanying algorithms. The implementation details of the new framework will be discussed in chapter 4. Afterwards, chapter 5 will evaluate the performance of this framework and provide insights into the characteristics of a typical test suite. More specifically, this chapter will investigate the probability of (repeated) test failure and the average duration of a test run. Finally, chapter 6 will present additional ideas and improvements to the framework.

Chapter 2

Software Engineering

The Institute of Electrical and Electronics Engineers `[IEEE]` defines the practice of Software Engineering as: ”Application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software; that is, the application of engineering to software” [19, p. 421]. The word “systematic” in this definition, emphasises the need for a structured process, depicting guidelines and models that describe how software should be developed the most efficient way possible. Such a process does exist and it is often referred to as the Software Development Life Cycle (SDLC) [19, p. 420]. In the absence of a model, i.e. when the developer does what they deem correct without following any rules, the term *Cowboy coding* is used [21, p. 34].

2.1 Software Development Life Cycle

An implementation of the SDLC consists of two major components. First, the process is broken down into several smaller phases. Depending on the nature of the software, it is possible to omit steps or add more steps. I have compiled a simple yet generic approach from multiple sources [13, 18], to which most software projects adhere. This approach consists of five phases.

1. **Requirements phase:** This is the initial phase of the development process. During this phase, the developer gets acquainted with the project and compiles a list of the desired functionalities [18]. Using this information, the developer eventually decides on the required hardware specifications and possible external software which will need to be acquired.

2. **Design phase:** After the developer has gained sufficient knowledge about the project requirements, they can use this information to draw an architectural design of the application. This design consists of multiple documents, including user stories and UML-diagrams.

3. **Implementation phase:** During this phase, the developer will write code according to the specifications defined in the architectural designs.

4. **Testing phase:** This is the most important phase. During this phase, the implementation is tested to identify potential bugs before the application is used by other users.

5. **Operational phase:** In the final phase, the project is fully completed and it is integrated in the existing business environment.

Subsequently, a model is chosen to define how to transition from one phase into another phase. A manifold of models exist [13], each having advantages and disadvantages, but I will consider the basic yet most widely used model, which is the Waterfall model by Benington [4]. The initial Waterfall model required every phase to be executed sequentially and in order, cascading. However, this imposes several issues, the most prevalent being the inability to revise design decisions taken in the second phase, when performing the actual implementation in the third phase. To mitigate this, an improved version of the Waterfall model was proposed by Royce [31]. This version allows a phase to transition back to a previous phase (Figure 2.1).

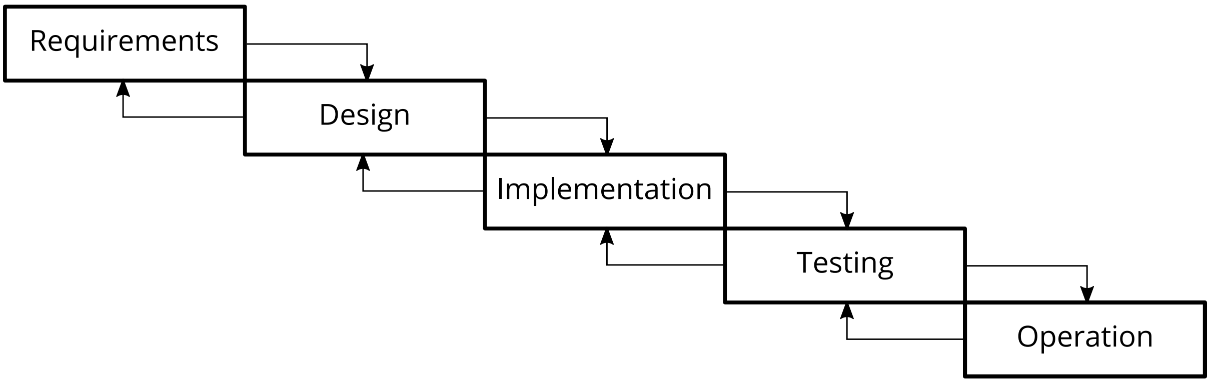


Figure 2.1: Improved Waterfall model by Royce

In this thesis I will solely focus on the implementation and testing phase, as these are the most time-consuming phases of the entire process. The modification to the Waterfall model by Royce is particularly useful when applied to these two phases, in the context of *software regressions*. A regression [28] is a feature that was previously working correctly, but is now malfunctioning. This behaviour can have external causes, such as a change in the system clock because of daylight saving time, but can also be the result of a change to another, seemingly unrelated part of the application code [17].

Software regressions and other functional bugs can ultimately incur disastrous effects, such as severe financial loss or damage to the reputation of the software company. The most famous example in history is without any doubt the explosion of the Ariane 5-rocket, which was caused by an integer overflow [22]. In order to reduce the risk of bugs, malfunctioning components should be detected as soon as possible to proactively defend against potential failures. Because of this reason, the testing phase is to be considered as the most important phase of the entire development process and an application should therefore include sufficient tests. The collection of all tests included in an application, or a smaller chosen subset of certain tests, is referred to as the *test suite*. Tests can be classified in multiple categories, this thesis will consider three distinguishable categories:

1. **Unit test:** This is the most basic kind of test. The purpose of a unit test is to verify the behaviour of an individual component [35]. The scope of a unit test should be limited to a small and isolated piece of code, such as one function. Unit tests are typically implemented as *white-box tests* [17, p. 12]. A white-box test is constructed by manually inspecting the function under test, to identify important *edge values*. The unit test should then feed these values as arguments to the function under test, to observe its behaviour. Common edge cases include zero, negative numbers, empty arrays or array boundaries that might result in an overflow.

2. **Integration test:** A more advanced test, an integration test verifies the interaction between multiple individually tested components [35]. Examples of integration tests include the communication between the front-end and the back-end side of an application. As opposed to unit tests, an integration test is an example of a *black-box* test [17, p. 6], meaning that implementation-specific details should be irrelevant or unknown when writing an integration test.

3. **Regression test:** After a regression has been detected, a regression test [19, p. 372] is added to the test suite. This regression test should replicate the exact conditions and sequence of actions that have caused the regression, to warden the implementation against subsequent failures if the same conditions would reapply in the future.

2.1.1 Test Suite Assessment

Coverage

The most frequently used metric to measure the quantity and thoroughness of a test suite is the *code coverage* or *test coverage* [19, p. 467]. The test coverage is expressed as a percentage and indicates which fraction of the application code is affected by code in the test suite. Internally, this works by augmenting every statement in the application code using binary instrumentation. A hook is inserted before and after every statement to keep track of which statements are executed during tests. Many different criteria exist to interpret these instrumentation results and thus to express the fraction of covered code [27], the most commonly used ones are *statement coverage* and *branch coverage*.

Statement coverage expresses the fraction of code statements that are executed in any test of the test suite [17], out of all executable statements in the application code. Analogously, the fraction of lines covered by a test may be used to calculate the *line coverage* percentage. Since one statement can span multiple lines and one line may also contain more than one statement, both of these criteria implicitly represent the same value. Statement coverage is heavily criticised in literature [27, p. 37], since it is possible to achieve a statement coverage percentage of 100% on a code fragment which can be proven to be incorrect. Consider the code fragment in Listing 2.1. If a test would call the `example`-function with arguments $\{a = 1, b = 2\}$, the test will pass and every statement will be covered, resulting in a statement coverage of 100%. However, it is clear to see that if the function would be called with arguments $\{a = 0, b = 0\}$, a *division-by-zero* error would be raised, resulting in a crash. This very short example already indicates that statement coverage is not trustworthy, yet it may still be useful for other purposes, such as detecting unreachable code which may safely be removed.

```c
int example(int a, int b) {
        if (a == 0 || b != 0) {
                return a / b;
        }
}
```

Listing 2.1: Example of irrelevant statement coverage in C.

Branch coverage on the other hand, requires that every branch of a conditional statement is traversed at least once [27, p. 37]. For an `if`-statement, this results in two tests being required, one for every possible outcome of the condition (`true` or `false`). For a `loop`-statement, this requires a test case in which the loop body is never executed and another test case in which the loop body is always executed. Remark that while this criterion is stronger than statement coverage, it is still not sufficiently strong to detect the bug in Listing 2.1. In order to mitigate this, *multiple-condition coverage* [27, p. 40] is used. This criterion requires that for every conditional statement, every possible combination of subexpressions is evaluated at least once. Applied to Listing 2.1, the `if`-statement is only covered if the following four cases are tested, which is sufficient to detect the bug.

- $a = 0, b = 0$
- $a = 0, b \neq 0$
- $a \neq 0, b = 0$
- $a \neq 0, b \neq 0$

It should be self-evident that achieving and maintaining a coverage percentage of 100% at all times is critical. However, this does not necessarily imply that all lines, statements or branches need to be covered explicitly [7]. Some parts of the code might simply be irrelevant or untestable. Examples include wrapper or delegation methods that simply call a library function. All major programming languages have frameworks and libraries available to collect coverage information during test execution, and each of these frameworks allows the developer to exclude parts of the code from the final coverage calculation. As of today, the most popular options are JaCoCo[1] for Java, coverage.py[2] for Python and simplecov[3] for Ruby. These frameworks are able to generate in-depth statistics on which parts of the code are covered and which parts require more tests, as illustrated in Figure 2.3.

Mutation testing

Whereas code coverage can be used to identify whether or not a part of the code is currently affected by the test suite, *mutation testing* can be used to measure its quality and ability to detect future failures. This technique creates several syntactically different instances of the source code, referred to as *mutants*. A mutant can be created by applying one or more *mutation operators* to the original source code. These mutation operators are aimed at simulating typical mistakes that developers tend to make, such as the introduction of off-by-one errors, removal of statements and replacement of logical connectors [30]. The *mutation order* refers to the amount of mutation operators that have been applied consecutively to an instance of the code. This order is traditionally rather low, as a result of the *Competent Programmer Hypothesis*, which states that programmers develop programs which are near-correct [20].

Creating and evaluating the mutant versions of the code is a computationally expensive process and requires human intervention, which is why very few software developers have managed to employ this technique in practice. Figure 2.2 shows how mutation testing is performed. First of all, the mutation system takes the original program P and a set of test cases T. Then, several mutation operators are applied to construct a large set of mutants P'. The next step is to evaluate every test case t on the original program P to verify its correctness, this is a task that needs to be performed manually. If at least one of these test cases proves incorrect, a bug has been found in the original program, which needs to be resolved before the mutation analysis can continue. When P successfully passes every test case, every test case are evaluated for each of the mutants. A mutant p' is said to be "killed" if its output is different from

[1] `https://www.jacoco.org/jacoco/`
[2] `https://github.com/nedbat/coveragepy`
[3] `https://github.com/colszowka/simplecov`

P for at least one test case, otherwise it is considered "surviving". After executing all test cases, the set of surviving mutants should be analysed in order to introduce subsequent test cases that can be used to kill them. However, it is also possible that the surviving mutants are functionally equivalent to P. This needs to be verified manually, since the detection of program equivalence is impossible [20, 30].

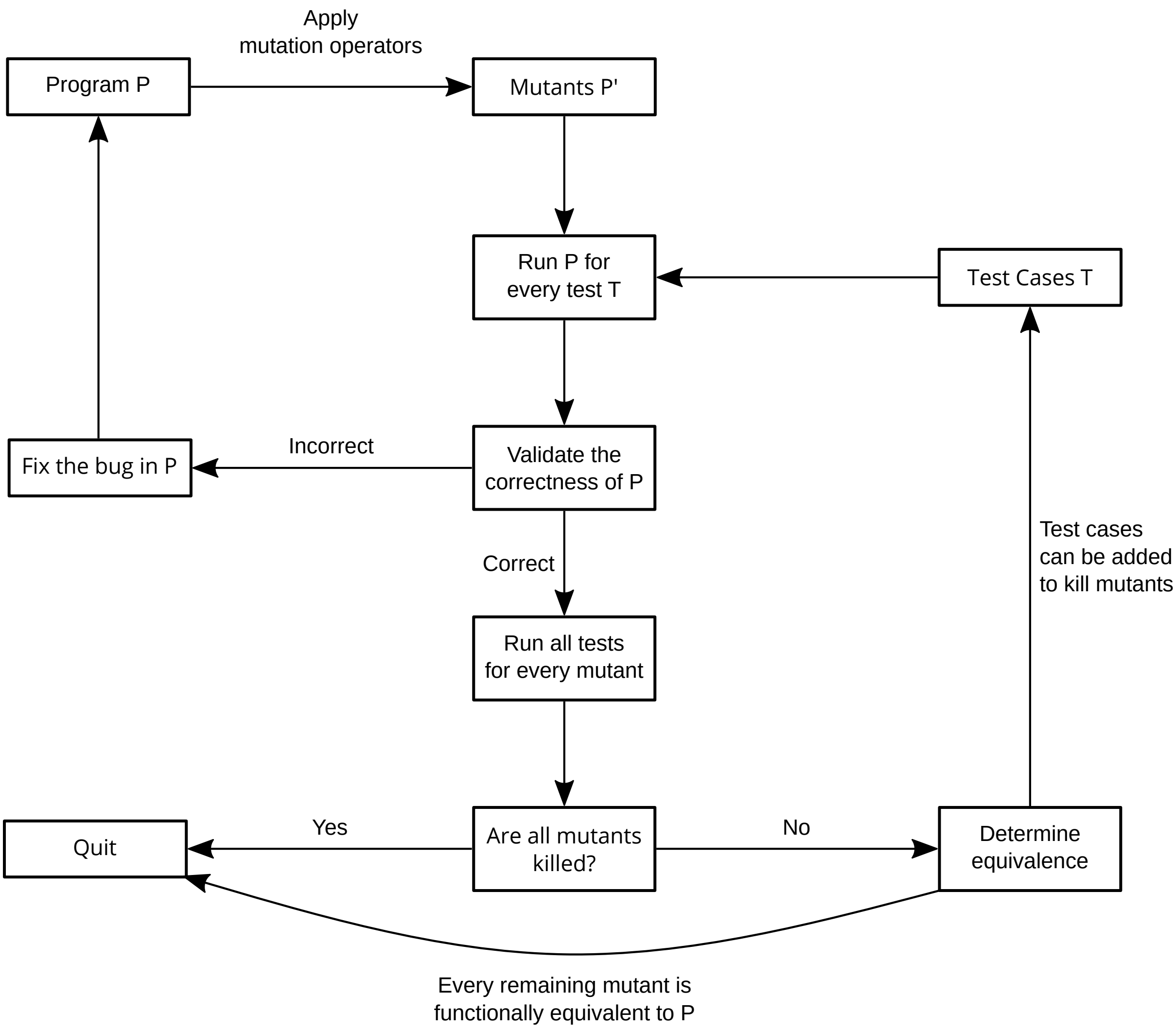


Figure 2.2: Process of Mutation Testing (based on [30])

After every mutant has either been killed or marked equivalent to the original problem, the test suite is assigned a *mutation score* which is calculated using Equation 2.1. In an ideal test suite, this score should be equal to 1, indicating that the test suite was able to detect every mutant.

$$\text{Mutant Score} = \frac{\text{killed mutants}}{\text{non-equivalent mutants}} \tag{2.1}$$

io.github.thepieterdc.http.impl

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
HttpClientImpl		59%		14%	7	14	18	40	2	9	0	1
HttpResponseImpl		55%		n/a	9	15	10	22	9	15	0	1
Total	88 of 211	58%	6 of 7	14%	16	29	28	62	11	24	0	2

(a) JaCoCo coverage report of `https://github.com/thepieterdc/dodona-api-java`

Coverage report: 75%

Module ↓	*statements*	*missing*	*excluded*	*coverage*
awesome/__init__.py	4	1	0	75%

```python
def smile():
    return ":)"

def frown():
    return ":("
```

Total	**4**	**1**	**0**	**75%**

(b) coverage.py report of `https://github.com/codecov/example-python`

Helpers (88.41% covered at 22.84 hits/line)

12 files in total. **716** relevant lines. **633** lines covered and **83** lines missed

File	% covered	Lines	Relevant Lines	Lines covered	Lines missed	Avg. Hits / Line
app/helpers/standard_form_builder.rb	**100.0 %**	5	3	3	0	11.0
app/helpers/renderers/feedback_code_renderer.rb	**100.0 %**	25	16	16	0	5.4
app/helpers/institutions_helper.rb	**100.0 %**	2	1	1	0	1.0
app/helpers/api_tokens_controller_helper.rb	**100.0 %**	2	1	1	0	1.0
app/helpers/renderers/pythia_renderer.rb	**93.94 %**	290	165	155	10	3.6
app/helpers/renderers/feedback_table_renderer.rb	**90.59 %**	349	202	183	19	16.8
app/helpers/exercise_helper.rb	**90.16 %**	125	61	55	6	3.5
app/helpers/courses_helper.rb	**86.67 %**	36	15	13	2	28.4
app/helpers/repository_helper.rb	**85.71 %**	11	7	6	1	2.6
app/helpers/application_helper.rb	**85.59 %**	220	111	95	16	62.6
app/helpers/users_helper.rb	**84.62 %**	20	13	11	2	1.4
app/helpers/renderers/lcs_html_differ.rb	**77.69 %**	236	121	94	27	38.2

Showing 1 to 12 of 12 entries

(c) simplecov report of `https://github.com/dodona-edu/dodona`

Figure 2.3: Statistics from Code coverage tools

2.2 Agile Software Development

2.2.1 The need for Agile

In the wake of the world economic crisis, software companies were forced to devote efforts into researching how their overall expenses could be reduced. This research has concluded that in order to reduce financial risks, the *time-to-market* of an application should be as short as possible. In order to accomplish this, further research was conducted, resulting in an increase of attention for agile methodologies in scientific literature [16]. As was previously described in **??**, agile methodologies strive to deliver a minimal version as soon as possible, allowing additional functionality to be added in an incremental fashion. This effectively results in a shorter *time-to-market* and lower costs, since the company can decide to cancel the project much earlier in the process.

In addition to a reduced time-to-market, maintaining an agile workflow has also proven beneficial to the success rate of development. A study performed by The Standish Group revealed that the success rate of agile projects is more than three times higher compared to when traditional methodologies are practised, as illustrated in Figure 2.4.

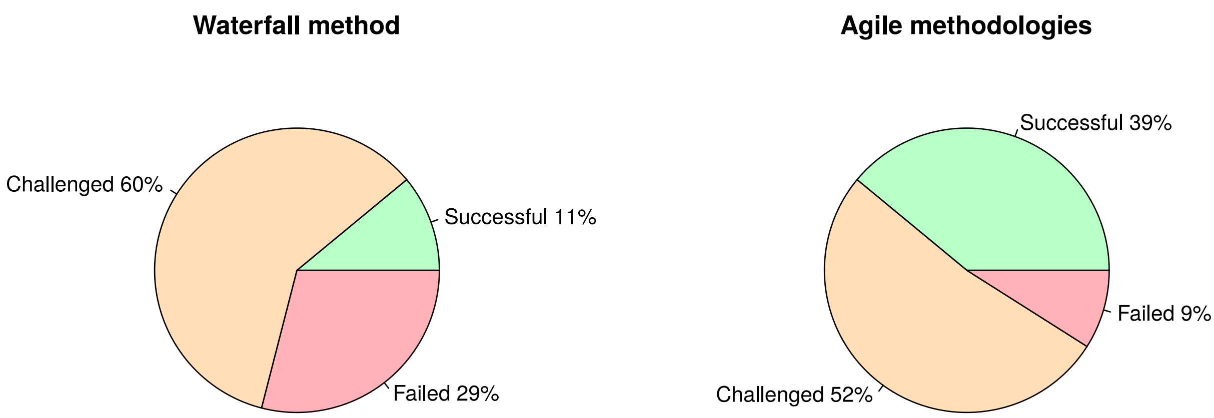


Figure 2.4: Success rate of Agile methodologies [14].

2.2.2 Continuous Integration

In traditional software development, the design phase typically leads to a representation of the required functionality in multiple, stand-alone modules. Subsequently, every module is implemented separately by individual developers. Afterwards, an attempt is made to integrate all the modules into the final application, an event to which Meyer refers to as the “Big Bang” [26, p.103]. The name *Big Bang* reflects the complex nature of this operation. This can prove to be a challenging operation, because every developer can take unexpected assumptions at the start of the project, which may ultimately result in mutually incompatible components. Furthermore, since the code was written over a span of several weeks to months, the developers often need to rewrite code that they have not touched in a long time. Eventually this will lead to unanticipated delays and costs [32].

Contrarily, agile development methodologies advocate the idea of frequent, yet small deliveries (**??**). Consequently, this implies that the code is built often and that the modules are integrated multiple times, on a *continuous* basis, rather than just once at the end, thus allowing for early identification of problems [12]. This practice of frequent builds is referred to as *Continuous Integration* [25, 26]. It should be noted that this idea has existed and has been applied before the agile manifesto was written. The first notorious software company that has adopted this practice is Microsoft, already in 1989 [6, p.11]. Cusumano reports that Microsoft typically builds the entire application at least once per day [6, p.12], therefore requiring developers to integrate and test their changes multiple times per day.

The introduction of Continuous Integration `[CI]` in software development has important consequences on the life cycle. Where the waterfall model used a cascading life cycle, Continuous Integration employs a circular, repetitive structure consisting of three phases, as visualised in Figure 2.5.

1. **Implementation:** In the first phase, every developer individually writes code for the module they were assigned to. At a regular interval, the code is committed to the remote repository.

2. **Integration:** When the code is committed, the developer simultaneously fetches the changes to other modules. Afterwards, the developer must integrate the changes with his own module, to ensure compatibility. In case a conflict occurs, the developer is responsible for its resolution [25].

3. **Test:** After the module has successfully been integrated, the test suite is run to ensure no bugs have been introduced.

Adopting Continuous Integration can prove to be a lengthy and repetitive task. Luckily, a variety of tools and frameworks exist to automate this process. Essentially, these tools are typically attached to a version control system (e.g. Git, Mercurial, ...), using a `post-receive` hook. Every time a commit is pushed by one of the developers, the CI system is notified, after which the code is automatically built and tests are executed. Optionally, the system can be configured to automatically publish successful

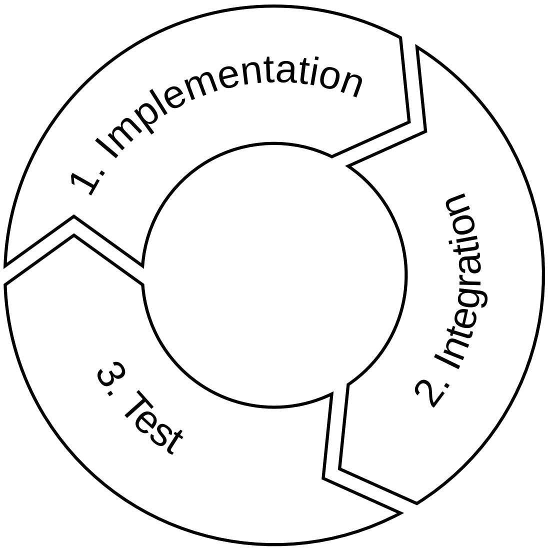


Figure 2.5: Development Life Cycle with Continuous Integration

runs to the end users, a process referred to as *Continuous Delivery*. I will now proceed by discussing four prominent Continuous Integration systems.

Jenkins

Jenkins CI[4] was started as a hobby project in 2004 by Kohsuke Kawaguchi, a former employee of Sun Microsystems. Jenkins is programmed in Java and profiles itself as "The leading open source automation server". It was initially named Hudson, but after Sun was acquired by Oracle, issues related to the trademark Hudson arose. In response, the developer community decided to migrate the Hudson code to a new repository and rename the project to Jenkins [32]. As of today, Jenkins is still widely used for many reasons. Since it is open source and its source code is located on GitHub, it is free to use and can be self-hosted by the developers in a private environment. Furthermore, Jenkins provides an open ecosystem to support developers into writing new plugins and extending its functionality. A market research conducted by ZeroTurnaround in 2016 revealed that Jenkins is the preferred Continuous Integration tool by 60% of the developers [23].

Figure 2.6: Logo of Jenkins CI (`https://jenkins.io/`)

[4] `https://jenkins.io/`

GitHub Actions

Following the successful beta of GitHub Actions which had started in August 2019, GitHub launched its own Continuous Integration system later that year in November[5]. GitHub Actions executes builds in the cloud on servers owned by GitHub and can therefore only be used in conjunction with a GitHub repository, support for GitHub Enterprise repositories is not yet available. The developers can define builds using *workflows* that can be configured to run both on Linux, Windows as well as OSX hosts. Private repositories are allowed a fixed amount of free build minutes per month, while builds of public repositories are always free of charges [9]. Similar to Jenkins, GitHub Actions can be extended with custom plugins. These plugins can be created using either a Docker container, or in native JavaScript, which allows faster execution [1]. It should be noted however that due to the recent nature of this system, not many plugins have been created yet.

Figure 2.7: Logo of GitHub Actions (`https://github.com/features/actions`)

GitLab CI

GitLab, the main competitor of GitHub, announced its own Continuous Integration service in late 2012 named GitLab CI[6]. GitLab CI builds are configured using *pipelines* and are executed by *GitLab Runners*. These runners are operated by developers on their own infrastructure. Additionally, GitLab also offers the possibility to use *shared runners*, which are hosted by themselves [2]. Equivalent to the aforementioned GitHub Actions, shared runners can be used for free by public repositories and are bounded by quota for private repositories [11]. A downside of using GitLab CI is the lack of a community-driven plugin system, however this is a planned feature [7].

Figure 2.8: Logo of GitLab CI (`https://gitlab.com/`)

[5] `https://github.blog/2019-08-08-github-actions-now-supports-ci-cd/`
[6] `https://about.gitlab.com/blog/2012/11/13/continuous-integration-server-from-gitlab/`
[7] `https://gitlab.com/gitlab-org/gitlab/issues/15067`

Travis CI

The final Continuous Integration platform which I will discuss is Travis CI. This Continuous Integration system was launched in 2011 and can only be used in addition to an existing GitHub repository. Travis CI build tasks can be configured in a similar fashion as GitLab CI, but the builds can exclusively be executed on their servers. Besides running builds after a commit has been pushed to the repository, it is also possible to schedule daily, weekly or monthly builds using cron jobs. Similar to GitHub Actions, open-source projects can be built at zero cost and a paid plan exists for private repositories [8]. It is not possible to create custom plugins, however Travis CI already features built-in support for a variety of programming languages. In 2020, almost 1 million projects are being built using Travis CI [33].

Figure 2.9: Logo of Travis CI (`https://travis-ci.com/`)

Chapter 3

Related work

In the previous chapter, the paramount importance of frequently integrating one's changes into the upstream repository was emphasised. Additionally, Continuous Integration was introduced as both a practice and a tool to facilitate this often complex and time-consuming process. Continuous Integration is, however, not the golden bullet for software engineering, as there is a flip side to applying this practice. After every integration, all of the unit and regression tests in the test suite must be executed to ensure that the integration was successful and that no new bugs have been introduced. As the project evolves, the size of the codebase increases and consequently the amount of tests will increase as well in order to maintain a sufficiently high coverage level. An increase in the size of the test suite will inevitably lead to an increase in test duration [29], which imposes an issue of scaling. Walcott, Soffa and Kapfhammer illustrate the magnitude of this problem by providing an example of a codebase consisting of 20.000 lines, for which the tests require up to seven weeks to complete [34].

Fortunately, multiple developers and researchers have found some techniques that can be used to address the scalability issues of growing test suites. The techniques currently known to literature can be classified in three categories. Developers can either apply *Test Suite Minimisation*, *Test Case Selection* or *Test Case Prioritisation* [29]. All three techniques are applicable to any test suite, however there is a trade-off to be made. Depending on which technique is chosen, it will either have a major impact on the duration of the test suite execution in exchange for a reduced test coverage level, or it will result in a higher test adequacy.

In the following sections, the details of these three approaches will be discussed and accompanying algorithms will be provided. Since the approaches share common ideas, the algorithms can (albeit with minor modifications) be applied to all approaches. The final section will investigate the adoption and integration of these techniques and algorithms in existing prominent software testing frameworks.

3.1 Classification of approaches

3.1.1 Test Suite Minimisation

Test Suite Minimisation, also referred to as *Test Suite Reduction*, aims to reduce the size of the test suite by permanently removing redundant tests. This problem is formally defined by Rothermel in 1 [36] and illustrated in Figure 3.1.

Definition 1 (Test Suite Minimisation)**.**
Given:

- $T = \{t_1, \ldots, t_n\}$ *a test suite consisting of tests* t_j*.*
- $R = \{r_1, \ldots, r_n\}$ *a set of requirements that must be satisfied in order to provide the desired "adequate" testing of the program.*
- $\{T_1, \ldots, T_n\} \subseteq T$ *subsets of test cases, one associated with each of the requirements* r_i*, such that any one of the test cases* $t_j \in T_i$ *can be used to satisfy requirement* r_i*.*

Test Suite Minimisation is then defined as the task of finding a set T' *of test cases* $t_j \in T$ *that satisfies all requirements* r_i*.*

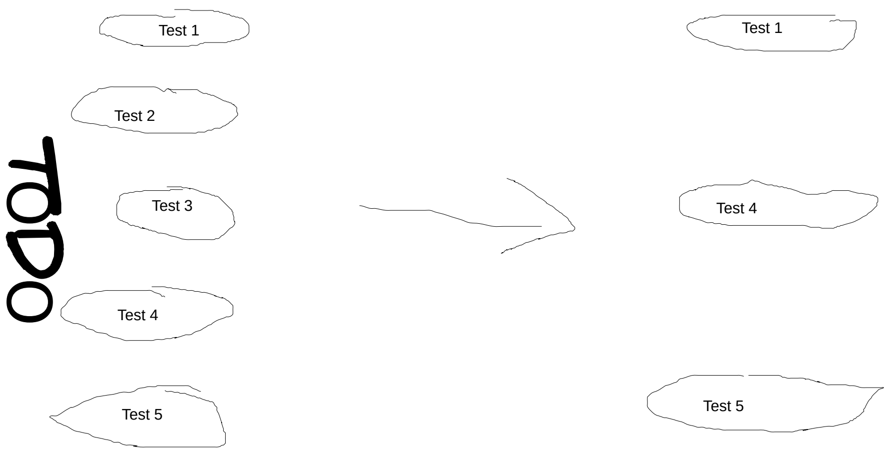


Figure 3.1: Test Suite Minimisation

If we apply this definition to the concepts introduced in **??**, the requirements R can be interpreted as lines in the codebase that must be covered. With respect to the definition, a requirement can be satisfied by any test t_j that belongs to subset T_i of T. Observe that the problem of finding T' is closely related to the *hitting set problem* (2) [36].

Definition 2 (Hitting Set Problem)**.**
Given:

- $S = \{s_1, \ldots, s_n\}$ *a finite set of elements.*
- $C = \{c_1, \ldots, c_n\}$ *a collection of sets, with* $\forall c_i \in C : c_i \subseteq S$*.*
- K *a positive integer,* $K \leq |S|$*.*

The hitting set is a subset $S' \subseteq S$ *such that* S' *contains at least one element from each subset in* C*.*

In the context of Test Suite Minimisation, T' is precisely the hitting set of T_is. In order to effectively minimise the amount of tests in the test suite, T' should be the minimal hitting set [36], which is an NP-complete problem as it can be reduced to the *Vertex Cover*-problem [10].

3.1.2 Test Case Selection

The second algorithm closely resembles the previous one. Instead of determining the minimal hitting set of the test suite in order to permanently remove tests, this algorithm has a notion of context. Prior to the execution of the tests, the algorithm performs a *white-box static analysis* of the codebase to identify which parts have been changed. Subsequently, only the tests regarding modified parts are executed, making the selection temporary (Figure 3.2) and modification-aware [36]. Rothermel and Harrold define this formally in 3.

Definition 3 (Test Case Selection)**.**
Given:

- P *the previous version of the codebase*
- P' *the current (modified) version of the codebase*
- T *the test suite*

Test Case Selection aims to find a subset $T' \subseteq T$ *that is used to test* P'*.*

3.1.3 Test Case Prioritisation

Where the previous algorithms both attempted to execute as few tests as possible, it might sometimes be desired or even required that all tests pass. In this case, the previous ideas can be used as well. In Test Case Prioritisation, we want to find a permutation of the sequence of all tests instead of eliminating certain tests. The order of the permutation is chosen specifically to achieve a given goal as soon as possible, allowing for early termination of the test suite upon failure [36]. Some examples of

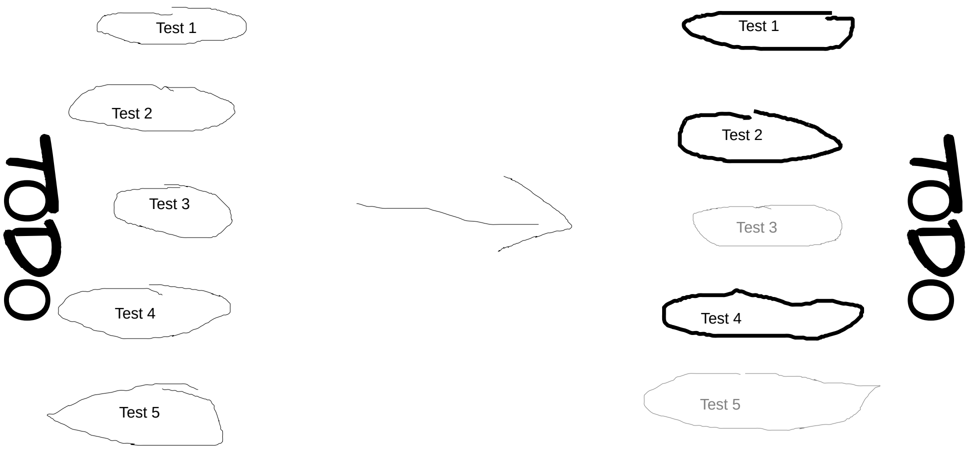


Figure 3.2: Test Case Selection

goals include covering as many lines of code as fast as possible, or early execution of tests with a high probability of failure. A formal definition of this algorithm is provided in 4.

Definition 4 (Test Case Prioritisation)**.**
Given:

- T *the test suite*
- PT *the set of permutations of* T
- $f : PT \mapsto \mathbb{R}$ *a function from a subset to a real number, this function is used to compare sequences of tests to find the optimal permutation.*

Test Case Prioritisation finds a permutation $T' \in PT$ *such that* $\forall T'' \in PT : f(T') \geq f(T'') \Rightarrow (T'' \neq T')$

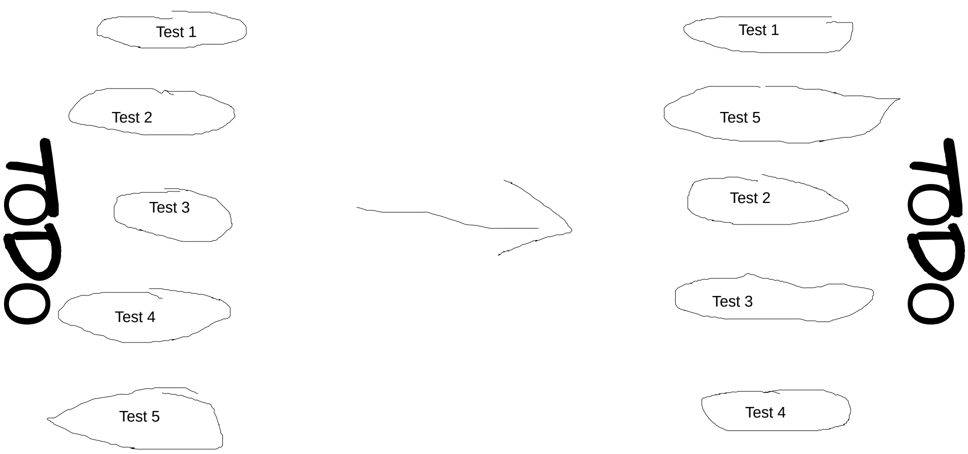


Figure 3.3: Test Case Prioritisation

3.2 Algorithms

In subsection 3.1.1 the relation was explained between applying Test Suite Minimisation and finding the minimal hitting set of the test suite and the set of requirements, which is an NP-complete problem. Therefore, the use of *heuristics* is required. A heuristic is an experience-based method that can be used to solve a hard to compute problem by finding a fast approximation [17]. However, the found solution will mostly be suboptimal or might sometimes even fail to find any solution at all. Considering its relation to the minimal hitting set problem, heuristics that are known to literature for solving this problem can also be used to implement Test Suite Minimisation. A selection of these heuristics will be discussed below. It should be noted however that the used terminology and naming of the variables might have been changed to ensure mutual consistency. Every algorithm has been adapted to adhere to the conventions provided in 5 and 6.

Definition 5 (Naming convention)**.**

- C*: the set of all lines in the application source code that are covered by at least one test case* $t \in TS$*.*
  - CT_l *denotes the test group* l*, which corresponds to the set of all tests* $t \in TS$ *that cover source code line* $l \in C$*.*
- RS*: the representative set of test cases, these are the test cases that have been selected by the algorithm.*
- TS*: the set of all test cases in the test suite.*
  - TL_t *denotes the set of all source code lines that are covered by test* $t \in TS$*.*

Definition 6 (Cardinality)**.** *For a finite set* S*, the cardinality* $|S|$ *is defined as the number of elements in* S*. In case of potential confusion, the cardinality of* S *can also be denoted as* $Card(S)$*.*

3.2.1 Greedy algorithm

The first algorithm is a *greedy* heuristic, which was originally designed by Chvatal to find an approximation for the set-covering problem [29]. A greedy algorithm always makes a locally optimal choice, assuming that this will eventually lead to a globally optimal solution [5]. Algorithm 1 presents the Greedy algorithm for Test Suite Minimisation. The goal of the algorithm is to construct a set of test cases that cover every line in the code, by requiring as few tests as possible.

Initially, the algorithm starts with an empty result set RS, the set TS of all test cases and the set C of all coverable source code lines. Furthermore, TL_t denotes the set of source code lines in C that are covered by test case $t \in TS$. Subsequently, the algorithm iteratively selects test cases from TS and adds them to RS. The locally optimal choice is to always select the test case that will contribute the most still uncovered lines, ergo the test t for which the cardinality of the intersection between C and TL_t is maximal. After every iteration, the now covered lines TL_t are removed from C and the selection process is repeated until C is empty. Upon running the tests, only the tests in RS must be executed. This algorithm can be converted to make it applicable to Test Case Prioritisation by converting the set RS to a list to maintain the order in which the test cases were selected, which is equivalent to the prioritised order of execution.

Algorithm 1 Greedy algorithm for Test Suite Minimisation

Input: Set TS of all test cases,
    Set C of all source code lines that are covered by any $t \in TS$,
    TL_t the set of all lines are covered by test case $t \in TS$.
Output: Subset $RS \subseteq TS$ of tests to execute.
$RS \leftarrow \emptyset$
while $C \neq \emptyset$ **do**
    $t\_max \leftarrow 0$
    $tl\_max \leftarrow \emptyset$
    for all $t \in TS$ **do**
        $tl\_current \leftarrow C \cap TL_t$
        if $|tl\_current| > |tl\_max|$ **then**
            $t\_max \leftarrow t$
            $tl\_max \leftarrow tl\_current$
    $RS \leftarrow RS \cup \{t\_max\}$
    $C \leftarrow C \setminus tl\_max$

3.2.2 HGS

The second algorithm was created by Harrold, Gupta and Soffa [15]. This algorithm constructs the minimal hitting set of the test suite in an iterative fashion. As opposed to the greedy algorithm (subsection 3.2.1), the HGS algorithm considers the test groups CT instead of the set TLt to obtain a list of test cases that cover all source code lines. More specifically, this algorithm considers the distinct test groups, denoted as CTD. Two test groups are considered indistinct if they differ in at least one test case. The pseudocode for this algorithm is provided in Algorithm 2.

Similar to the previous algorithm, an empty representative set RS is constructed in which the selected test cases will be stored. The algorithm begins by iterating over every source code line $l \in C$ and constructing the corresponding set of test groups CT_l. As mentioned before, for performance reasons this set is reduced to CTD, only retaining distinct test groups. Next, the algorithm selects every test group of which the cardinality is equal to 1 and adds these to RS. This corresponds to every test case that covers a line of code, which is exclusively covered by that single test case. Subsequently, the lines that are covered by any of the selected test cases are removed from C. This process is repeated for an incremented cardinality, until every line in C is covered. Since the remaining test groups will now contain more than one test case, the algorithm needs to make a choice on which test case to select. The authors have chosen that the test case that occurs in the most test groups is preferred. In the event of a tie, this choice is deferred until the next iteration.

The authors have provided an accompanying calculation of the computational time complexity of this algorithm [15]. With respect to the naming convention introduced in 5, additionally let n denote the number of distinct test groups CTD, nt the number of test cases $t \in TS$ and MAX_CARD the cardinality of the largest test group. The HGS algorithm consists of two steps which are performed repeatedly. The first step involves computing the number of occurrences of every test case t in each test group. Given that there are n distinct test groups and, in the worst case scenario, each test group can contain MAX_CARD test cases which all need to be examined once, the computational cost of this step is equal to $O(n * MAX\_CARD)$. In order to determine which test case should be included in the representative set RS, the algorithm needs to find all test cases for which the number of occurrences in all test groups is maximal, which requires at most $O(nt * MAX\_CARD)$. Since every repetition of these two steps adds a test case that belongs to at least one out of n test groups to the representative set, the overall runtime of the algorithm is $O(n * (n + nt) * MAX\_CARD)$.

Algorithm 2 HGS algorithm ([15])

Input: Distinct test groups $T_1, \dots T_n \in CDT$, containing test cases from TS.
Output: Subset $RS \subseteq TS$ of tests to execute.
$marked \leftarrow array[1 \dots n]$ ▷ initially $false$
$MAX\_CARD \leftarrow max\{Card(T_i)|T_i \in CDT\}$
$RS \leftarrow \bigcup\{T_i|Card(T_i) = 1\}$
for all $T_i \in CDT$ **do**
    if $T_i \cap RS \neq \emptyset$ **then** $marked[i] \leftarrow true$
$current \leftarrow 1$
while $current < MAX\_CARD$ **do**
    $current \leftarrow current + 1$
    while $\exists T_i : Card(T_i) = current, marked[i] = false$ **do**
        $list \leftarrow \{t|t \in T_i : Card(T_i) = current, marked[i] = false\}$
        $next \leftarrow SelectTest(current, list)$
        $reduce \leftarrow false$
        for all $T_i \in CDT$ **do**
            if $next \in T_i$ **then**
                $marked[i] = true$
                if $Card(T_1) = MAX\_CARD$ **then** $reduce \leftarrow true$
        if $reduce$ **then**
            $MAX\_CARD \leftarrow max\{Card(T_i)|marked[i] = false\}$
        $RS \leftarrow RS \cup \{next\}$
function SELECTTEST($size$, $list$)
    $count \leftarrow array[1 \dots nt]$
    for all $t \in list$ **do**
        $count[t] \leftarrow |\{T_j|t \in T_j, marked[T_j] = false, Card(T_j) = size\}|$
    $tests \leftarrow \{t|t \in list, count[t] = max(count)\}$
    if $|tests| = 1$ **then return** $tests[0]$
    else if $|tests| = MAX\_CARD$ **then return** $tests[0]$
    else return $SelectTest(size + 1, tests)$

3.2.3 ROCKET algorithm

In contrast to the previously discussed algorithms which focused on Test Suite Minimisation, the ROCKET algorithm is tailored for Test Case Prioritisation. This algorithm was presented by Marijan, Gotlieb and Sen [24] as part of a case study to improve the testing efficiency of industrial video conferencing software. Unlike the previous algorithms that only take code coverage into account, this algorithm also considers historical failure data and test execution time. The objective of this algorithm is twofold: select the test cases with the highest consecutive failure rate, whilst also maximising the number of executed test cases in a limited time frame. The below algorithm has been modified slightly, since the time frame is a domain-specific constraint for this particular industry case and irrelevant for this thesis. Since this is a prioritisation algorithm rather than a selection or minimisation algorithm, it yields a total ordering of all the test cases in the test suite, ordered using a weighted function.

The modified version of the algorithm (pseudocode is provided in Algorithm 3) takes three inputs:

- the set of test cases to prioritise $TS = \{T_1, \ldots, T_n\}$
- the execution time for each test case $E = \{E_1, \ldots, E_n\}$
- the failure status for each test case over the previous m successive executions $F = \{F_1, \ldots, F_n\}$, where $F_i = \{f_1, \ldots, f_m\}$

The algorithm starts by creating an array P of length n, which contains the priority of each test case. The priority of each test case is initialised at zero. Next, an $m \times n$ failure matrix MF is constructed and filled using the following formula.

$$MF[i,j] = \begin{cases} 1 & \text{if test case } T_j \text{ passed in execution } (current - i) \\ -1 & \text{if test case } T_j \text{ failed in execution } (current - i) \end{cases}$$

This matrix MF is visualised in Table 3.1. This table contains the hypothetical failure rates of the last three executions of six test cases.

run	T_1	T_2	T_3	T_4	T_5	T_6
$current - 1$	1	1	1	1	-1	-1
$current - 2$	-1	1	-1	-1	1	-1
$current - 3$	1	1	-1	1	1	-1

Table 3.1: Visualisation of the failure matrix MF.

Afterwards, P is filled with the cumulative priority of each test case, which is calculated by multiplying its failure rate with a domain-specific weight heuristic ω. This heuristic is used to derive the probability of repeated failures of the same test, given earlier failures. In their paper [24], the authors apply the following weights:

$$\omega_i = \begin{cases} 0.7 & \text{if } i = 1 \\ 0.2 & \text{if } i = 2 \\ 0.1 & \text{if } i >= 3 \end{cases}$$

$$P_j = \sum_{i=1 \ldots m} MF[i,j] * \omega_i$$

Finally, the algorithm groups test cases based on their calculated priority in P. Every test case that belongs to the same group is equally relevant for execution in the current test run. However, within every test group the tests will differ in execution time E. The final step is to reorder test cases that belong to the same group in such a way that test cases with a shorter duration are executed earlier in the group.

Algorithm 3 ROCKET algorithm

Input: Set $TS = \{T_1, \dots, T_n\}$ of all test cases,
    Execution time E of every test case,
    Failure status FS for each test case over the previous m successive iterations.
Output: Priority of test cases P.
$P \leftarrow array[1 \dots n]$ ▷ initially 0
$MF \leftarrow array[1 \dots m]$
for all $i \in 1 \dots m$ **do**
    $MF[i] \leftarrow array[1 \dots n]$
    for all $j \in 1 \dots n$ **do**
        if test case T_j failed in run $(current - i)$ **then** $MF[i][j] \leftarrow -1$
        else $MF[i][j] \leftarrow 1$
for all $j \in 1 \dots n$ **do**
    for all $i \in 1 \dots m$ **do**
        if $i = 1$ **then** $P[j] \leftarrow P[j] + (MF[i][j] * 0.7)$
        else if $i = 2$ **then** $P[j] \leftarrow P[j] + (MF[i][j] * 0.2)$
        else $P[j] + (MF[i][j] * 0.1)$
$Q \leftarrow \{P[j] | j \in 1 \dots n\}$ ▷ distinct priorities
$G \leftarrow array[1 \dots Card(Q)]$ ▷ initially empty sets
for all $j \in 1 \dots n$ **do**
    $p \leftarrow P[j]$
    $G[p] \leftarrow G[p] \cup \{j\}$
Sort every group in G based on ascending execution time in E.
Sort P according to which group it belongs and its position within that group.

3.3 Adoption in testing frameworks

Some of the approaches discussed above have been integrated in existing software testing frameworks. This paper will now proceed by conducting an analysis of these frameworks and tools to analyse which optimisation features are available and how they were implemented.

3.3.1 Gradle and JUnit

Gradle[1] is a dependency manager and application framework for Java, Groovy and Kotlin projects. Gradle supports multiple plugins to automate tedious tasks, such as configuration management, testing and deploying. One of the supported testing integrations is JUnit[2], which is the most widely used unit testing framework by Java developers. JUnit 5 is the newest version which is still actively being developed as of today. The framework is integrated as the testing framework of choice in several other Java libraries and frameworks, such as Android and Spring. JUnit offers mediocre support for features that optimise the execution of the test cases, especially when used in conjunction with Gradle. The following three key features are available:

1. **Parallel test execution:** JUnit comes bundled with multiple test processors that are responsible for processing test classes and to execute the test cases. One of these test processors is the `MaxNParallelTestClassProcessor`, which is capable of running a configurable amount of test cases in parallel. This results in a major speed-up of the overall test suite execution.

2. **Prioritise failed test cases:** Another test class processor which is provided by Gradle, is the `RunPreviousFailedFirstTestClassProcessor`. This processor will prioritise test cases that have failed in the previous run, similar to the idea of the ROCKET-algorithm (subsection 3.2.3), albeit without taking into account the duration of these test cases.

3. **Test order specification:** JUnit allows the user to specify the order in which test cases will be executed[3]. By default, a random yet deterministic order is used. The order can be manipulated by annotating the test class with the `@TestMethodOrder`-annotation, or by annotating individual test cases with the `@Order(int)`-annotation. This feature can only be used to alter the order of test cases within the same test class, it is not possible to perform inter-test class reordering. This feature could be used to sort test cases based on their execution time.

[1] `https://gradle.org`
[2] `https://junit.org`
[3] `https://junit.org/junit5/docs/current/user-guide/#writing-tests-test-execution-order`

Figure 3.4: Logo of Gradle

Figure 3.5: Logo of JUnit 5

3.3.2 OpenClover

OpenClover[4] is a code coverage framework for Java and Groovy projects. The framework was created by Atlassian and open-sourced in 2017. It profiles itself as "the most sophisticated code coverage tool", by extracting useful metrics from the coverage results and by providing features that can optimise the test suite. Among these features are powerful integrations with development software and prominent Continuous Integration services. Furthermore, OpenClover offers the automatic analysis of the coverage results to detect relations between the application source code and the test cases. This feature allows OpenClover to predict which test cases will have been affected, given a set of modifications to the source code. Subsequently, these predictions can be interpreted to implement Test Case Selection. This results in a reduced test suite execution time.

Figure 3.6: Logo of Atlassian Clover

[4] `https://openclover.org`

Chapter 4

Proposed framework: VeloCIty

The implementation part of this thesis consists of a framework and a set of tools, tailored at optimising the test suite as well as providing accompanying metrics and insights. The framework was named *VeloCIty* to reflect its purpose of enhancing the speed at which Continuous Integration is practised. This paper will now proceed by describing the design goals of the framework. Afterwards, a high-level schematic overview of the implemented architecture will be provided, followed by a more in-depth explanation of every pipeline step. In the final section of this chapter, the *Alpha* algorithm will be presented.

4.1 Design goals

VeloCIty has been implemented with four design goals in mind:

1. **Extensibility:** It should be possible and straightforward to support additional Continuous Integration systems, programming languages and test frameworks. Subsequently, a clear interface should be provided to integrate additional prioritisation algorithms.

2. **Minimally invasive:** Integrating VeloCIty into an existing test suite should not require drastic changes to any of the test cases.

3. **Language agnosticism:** This design goal is related to the framework being extensible. The implemented tools should not need to be aware of the programming language of the source code, nor the used test framework.

4. **Self-improvement:** The prioritisation framework supports all of the algorithms presented in section 3.2. It is possible that the performance of a given algorithm is strongly dependent on the nature of the project it is being applied to. In order to facilitate this behaviour, the framework should be able to measure the performance of every algorithm and “learn” which algorithm offers the best prediction, given a set of source code.

4.2 Architecture

The architecture of the VeloCIty framework consists of seven steps that are performed sequentially in a pipeline fashion, as illustrated in the sequence diagram (Figure 4.1). Every step is executed by one of three individual components, which will now be introduced briefly.

4.2.1 Agent

The first component that will be discussed is the agent. This is the only component that depends actively on both the programming language, as well as the used test framework, since it must interact directly with the source code and test suite. For every programming language or test framework that needs to be supported, a different implementation of an agent must be provided. This implementations are however strongly related, so much code can be reused or even shared. In this thesis, an agent was implemented in Java, more specifically as a plugin for the widely used Gradle and JUnit test framework. This combination was previously described in subsection 3.3.1. This plugin is responsible for running the test suite in a certain prioritised order, which is obtained by communicating with the controller (subsection 4.2.2). After the test cases have been executed, the plugin sends a feedback report to the controller, where it is analysed.

4.2.2 Controller

The second component is the core of the framework, acting as an intermediary between the agent on the left side and the predictor (subsection 4.2.3) on the right side. In order to satisfy the second design goal and allow language agnosticism, the agent communicates with the controller using the `HTTP` protocol by exposing a *REST*-interface. Representational State Transfer `[REST]` is a software architecture used by modern web applications that allows standardised communication using existing HTTP methods. On the right side, the controller does not communicate directly with the predictor, but rather stores prediction requests in a shared database which is periodically polled by the predictor. Besides routing prediction requests from the agent to the predictor, the controller will also update the meta predictor by evaluating the accuracy of earlier predictions of this project.

4.2.3 Predictor and Metrics

The final component is twofold. Its main responsibility is to apply the prioritisation algorithms and predict an order in which the test cases should be executed. This order

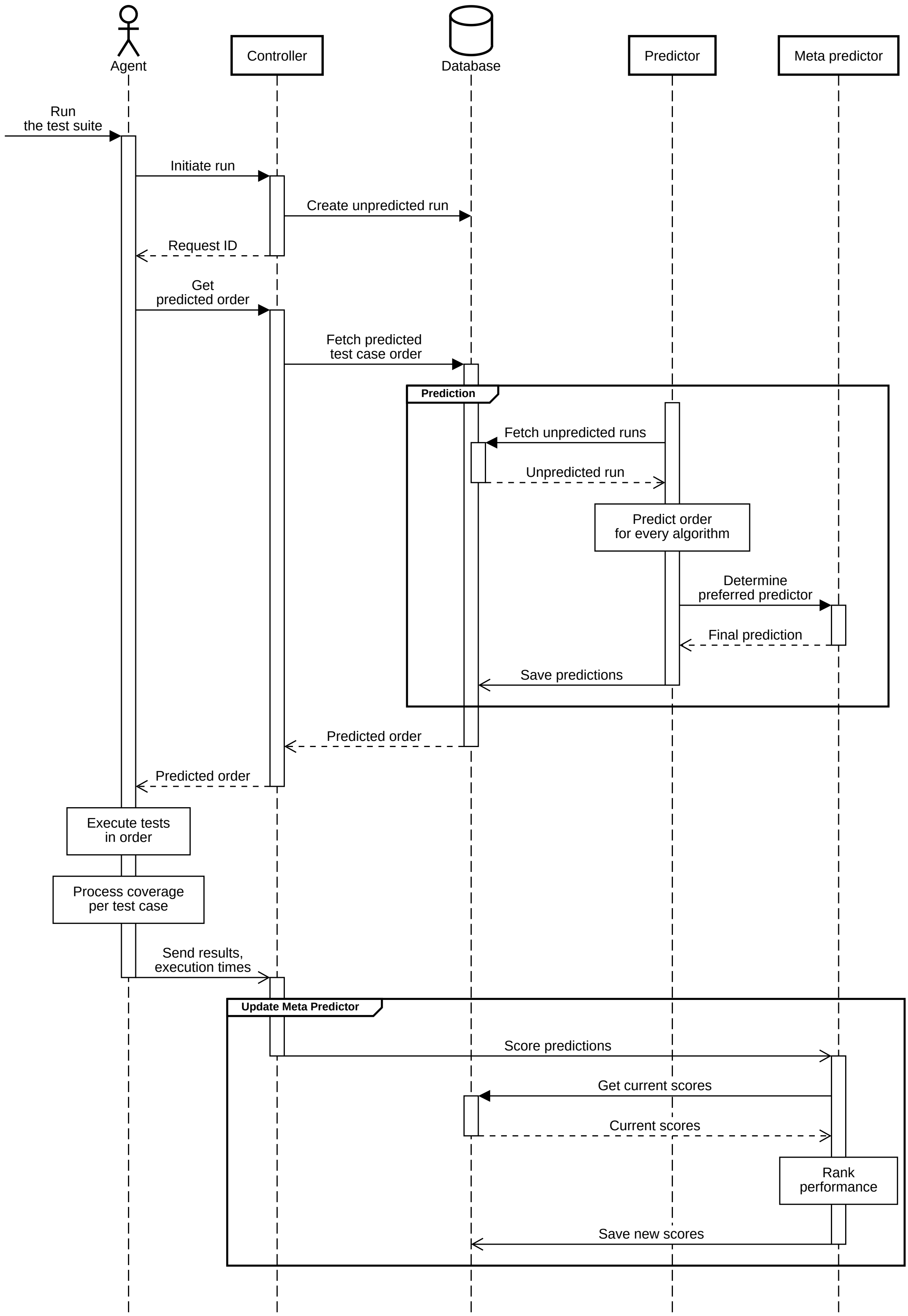


Figure 4.1: Sequence diagram of VeloCIty

is calculated by first executing ten algorithms and subsequently picking the algorithm that has been preferred by the meta predictor. Additionally, this component is able to provide metrics about the test suite, such as identifying superfluous test cases by applying Test Suite Minimisation. More specifically, this redundancy is obtained using the greedy algorithm (subsection 3.2.1). Both of these scripts have been implemented in Python, because of its simplicity and existing libraries for many common operations, such as numerical calculations (NumPy[1]) and machine learning (TensorFlow[2]).

4.3 Pipeline

This section will elaborate on the individual steps of the pipeline. The steps will be discussed by manually executing the pipeline that has hypothetically been implemented on a Java project. For the sake of simplicity, this explanation will assume a steady-state situation, ensuring the existence of at least one completed run of this project in the database at the controller side.

4.3.1 Initialisation

As was explained before, the provided Java implementation of the agent was designed to be used in conjunction with Gradle. In order to integrate VeloCIty into a Gradle project, the build script (`build.gradle`) should be modified in two places. The first change is to include and apply the plugin in the header of the file. Afterwards, the plugin requires three properties to be configured:

- `base` the path to the Java source files, relative to the location of the build script. This will typically resemble `src/main/java`.
- `repository` the url to the git repository at which the project is hosted. This is required in subsequent steps of the pipeline, to detect which code lines have been changed in the commit currently being analysed.
- `server` the url at which the controller can be reached.

Listing 4.1 contains a minimal integration of the agent in a Gradle build script, applied to a library for generating random numbers[3]. The controller is hosted at the same host as the agent and is accessible at port `8080`.

Listing 4.1: Minimal Gradle buildscript

```
buildscript {
```

[1] https://numpy.org/
[2] https://www.tensorflow.org/
[3] https://github.com/thepieterdc/random-java

```
    dependencies {
        classpath 'io.github.thepieterdc.velocity:velocity-
           junit:0.0.1-SNAPSHOT'
    }
}

plugins {
    id 'java'
}

apply plugin: 'velocity-junit'

velocity {
    base 'src/main/java/'
    repository 'https://github.com/thepieterdc/random-java'
    server 'http://localhost:8080'
}
```

After the project has been configured, the test suite must be executed. For the Gradle agent, this involves executing the built-in `test` task. This task requires an additional argument to be passed, which is the commit hash of the changeset to prioritise. In every discussed Continuous Integration system, this commit hash is available as an environment variable.

The first step is for the agent to initiate a new test run in the controller. This is accomplished by sending a `POST`-request to the `/runs` endpoint of the controller, which will reply with an identifier. On the controller side, this request will result in a new prioritisation request being enqueued in the database that will asynchronously be processed by the predictor daemon in the next step.

4.3.2 Prediction

The prediction of the test execution order is performed by the predictor daemon. This daemon continuously polls the database to fetch new test runs that need to be predicted. When a new test run is detected, the predictor executes every available prediction algorithm in order to obtain multiple prioritised test sequences. The following algorithms are available:

AllInOrder The first algorithm will simply prioritise every test case alphabetically and will be used for for benchmarking purposes in **??**.

AllRandom The second algorithm has also been implemented for benchmarking purposes. This algorithm will “prioritise” every test case arbitrarily.

AffectedRandom This algorithm will only consider the test cases that cover source code lines which have been modified in the current commit. These test cases will be ordered randomly, followed by the other test cases in the test suite in no particular order.

GreedyCoverAll The first of three implementations of the Greedy algorithm (subsection 3.2.1) will execute the algorithm to prioritise the entire test suite.

GreedyCoverAffected As opposed to the previous greedy algorithm, the second Greedy algorithm will only consider test cases covering changed source code lines to be prioritised. After these test cases, the remaining test cases in the test suite will be ordered randomly.

GreedyTimeAll Instead of greedily attempting to cover as many lines of the source code using as few tests as possible, this implementation will attempt to execute as many tests as possible, as soon as possible. In other words, this algorithm will prioritise test cases based on their average execution time.

HGSAll This algorithm is an implementation of the algorithm presented by Harrold, Gupta and Soffa (subsection 3.2.2). It is executed for every test case in the test suite.

HGSAffected Similar to the *GreedyAffected* algorithm, this algorithm is identical to the previous *HGSAll* algorithm besides that it will only prioritise test cases covering changed source code lines.

ROCKET The penultimate algorithm is a straightforward implementation of the pseudocode provided in subsection 3.2.3.

Alpha The final algorithm has been inspired by the other implemented algorithms. section 4.4 will further elaborate on the details.

Subsequently, the final prioritisation order is determined by applying the meta predictor. Essentially, the meta predictor can be seen as a table which assigns a score to every algorithm, indicating its performance on this codebase. subsection 4.3.4 will explain later how this score is updated. The predicted order by the algorithm with the highest score is eventually elected by the meta predictor as the final prioritisation order, and saved to the database.

4.3.3 Test case execution

Regarding the agent, the identifier obtained in subsection 4.3.1 is used to poll the controller by sending a `GET` request to `/runs/id`, which will reply with the test execution order if this has already been determined. One of the discussed features of Gradle in subsection 3.3.1 was the possibility to execute test cases in a chosen order by adding annotations. However, this feature cannot be used to implement the Java agent, since it only supports ordering test cases within the same test class. In order to facilitate complete control over the order of execution, a custom `TestProcessor` and `TestListener` have been implemented.

The `TestProcessor` is responsible for processing every test class in the classpath and forward it along with configurable options to a delegate processor. The final processor in this chain will eventually perform the actual execution of the test class. Since the delegate processors that are built into Gradle will by default execute every method in the test class, the custom processor needs to work differently. The implemented agent will first store every received test class into a list and load the class to obtain all test cases in the class using reflection. After all classes have been processed, the processor will iterate over the prioritised order. For every test case t in the order, the delegate processor is called with a tuple of the corresponding test class and an options array which excludes every test case except t. This will effectively forward the same test class multiple times to the delegate processor, but each time with an option that restricts test execution to the prioritised test case, resulting in the desired behaviour.

Subsequently, the `TestListener` is a method that is called before and after every invocation of a test case. This listener allows the agent to calculate the duration of every test case, as well as collect the intermediary coverage and save this on a per-test case basis.

4.3.4 Post-processing and analysis

The final step of the pipeline is to provide feedback to the controller, to evaluate the accuracy of the predictions and thereby implementing the fourth design goal of self-improvement. After executing all test cases, the agent sends the test case results, the execution time and the coverage per test case to the controller by issuing a `POST` request to `/runs/id/test-results` and `/runs/id/coverage`.

Upon receiving this data, the controller will update the meta predictor using the following procedure. The meta predictor is only updated if at least one of the test cases has failed, since the objective of Test Case Prioritisation is to detect failures as fast as possible, thus every prioritised order is equally good if there are no failures at all. If however a test case did fail, the predicted orders are inspected to calculate the duration until the first failed test case for every order. Subsequently, the average of all these durations is calculated. Finally, the score of every algorithm that predicted a below average duration until the first failure is increased, otherwise it is decreased. This will eventually lead to the most accurate algorithms being preferred in subsequent test runs.

4.4 Alpha algorithm

Besides the algorithms which have been presented in section 3.2, an additional algorithm has been implemented: the *Alpha* algorithm. This was constructed by examining the philosophy behind every discussed algorithm and subsequently combining the best ideas into a novel prioritisation algorithm. The specification below will assume the same naming convention as described in 5. The pseudocode is provided in Algorithm 4

The algorithm consumes the following inputs:

- the set of all n test cases: $TS = \{T_1, \ldots, T_n\}$
- the set of m *affected* test cases: $AS = \{T_1, \ldots, T_m\} \subseteq TS$. A test case t is considered "affected" if any source code line which is covered by t has been modified or removed in the commit that is being predicted.
- C: the set of all lines in the application source code, for which a test case $t \in TS$ exists that covers this line and that has not yet been prioritised. Initially, this set contains every covered source code line.
- the failure status of every test case, for every past execution out of k executions of that test case: $F = \{F_1, \ldots, F_n\}$, where $F_i = \{f_1, \ldots, f_k\}$. $F_{tj} = 1$ implies that test case t has failed in execution $current - j$.
- the execution time of test case $t \in TS$ for run $r \in [1 \ldots k]$, in milliseconds: D_{tr}.
- for every test case $t \in TS$, the set TL_t is composed of all source code lines that are covered by test case t.

The first step of the algorithm is to determine the execution time E_t of every test case t. This execution time is calculated as the average of the durations of every successful (i.e.) execution of t, since a test case will be prematurely aborted upon the first failed assertion, which introduces bias in the duration timings. In case t has never been executed successfully, the average is computed over every execution of t.

$$E_t = \begin{cases} \overline{\{D_{ti} | i \in [1 \ldots k], F_{ti} = 0\}} & \exists j \in [1 \ldots k], F_{tj} = 0 \\ \overline{\{D_{ti} | i \in [1 \ldots k]\}} & \text{otherwise} \end{cases}$$

Next, the algorithm executes every affected test case that has also failed at least once in its three previous executions. This reflects the behaviour of a developer attempting to resolve the bug that caused the test case to fail. Specifically executing *affected* failing test cases first is required in case multiple test cases are failing and the developer is resolving these one by one, an idea which was extracted from the ROCKET algorithm (subsection 3.2.3). In case there are multiple affected failing test cases, the test cases are prioritised by increasing execution time. After every selected test case, C is updated by subtracting the code lines that have been covered by at least one of these test cases.

Afterwards, the same operation is repeated for every failed but unaffected test case, likewise ordered by increasing execution time. Where the previous step helps developers to get fast feedback about whether or not the specific failing test case they were working on has been resolved, this step ensures that other failing test cases are not forgotten and are executed early in the run as well. Similar to the previous step, C is again updated after every prioritised test case.

Research (subsection 5.4.1) has indicated that on average, only a small fraction ($10\,\% - -20\,\%$) of all test runs will contain failed tests, resulting in the previous two steps not being executed at all. Therefore, the most time should be dedicated to executing test cases that cover affected code lines. More specifically, the next step of the algorithm executes every affected test case, sorted by decreasing cardinality of the intersection between C and the lines which are covered by the test case. Conforming to the prior two steps, C is also updated to reflect the selected test case. As a consequence of these updates, the cardinalities of these intersections change after every update, which will ultimately lead to affected tests not strictly requiring to be executed. This idea has been adopted from the Greedy algorithm subsection 3.2.1.

In the penultimate step, the previous operation is repeated in an identical fashion for the remaining test cases, similarly ordered by the cardinality of the intersection with the remaining uncovered lines in C.

Finally, the algorithm selects every test case which had not yet been prioritised. Notice that these test cases do not contribute to the test coverage, as every test case that would incur additional coverage would have been prioritised already in the previous step. Subsequently, these test cases are actually redundant and are therefore candidates for removal by Test Suite Minimisation. However, since this is a prioritisation algorithm, these tests will still be executed and prioritised by increasing execution time.

Algorithm 4 Alpha algorithm for Test Case Prioritisation

Input: Set $TS = \{T_1, \ldots, T_n\}$ of all test cases,
Set $AS = \{T_1, \ldots T_m\} \subseteq TS$ of affected test cases,
Set C of all source code lines that are covered by any $t \in TS$,
Execution times D_{tr} of every test case t, over all k runs r of that test case,
Failure status FS for each test case over the previous m successive iterations,
Sets $TL = \{TL_1, \ldots, TL_n\}$ of all source code lines that are covered by test case $t \in TS$.
Output: Ordered list P of n test cases and their priorities.
$P \leftarrow array[1 \ldots n]$ ▷ initially 0
$i \leftarrow n$
$FTS \leftarrow \{t | t \in TS \wedge (F[t][1] = 1 \vee F[t][2] = 1 \vee F[t][3] = 1)\}$
$AFTS \leftarrow AS \cap FTS$
for all $t \in AFTS$ **do** ▷ sorted by execution time in E (ascending)
    $C \leftarrow C \setminus TL[t]$
    $P[t] \leftarrow i$
    $i \leftarrow i - 1$
$FTS \leftarrow FTS \setminus AFTS$
for all $t \in FTS$ **do** ▷ sorted by execution time in E (ascending)
    $C \leftarrow C \setminus TL[t]$
    $P[t] \leftarrow i$
    $i \leftarrow i - 1$
$AS \leftarrow AS \setminus FTS$
while $AS \neq \emptyset$ **do**
    $t\_max \leftarrow AS[1]$ ▷ any element from AS
    $tl\_max \leftarrow \emptyset$
    for all $t \in AS$ **do**
        $tl\_current \leftarrow C \cap TL_t$
        if $|tl\_current| > |tl\_max|$ **then**
            $t\_max \leftarrow t$
            $tl\_max \leftarrow tl\_current$
    $C \leftarrow C \setminus tl\_max$
    $P[t] \leftarrow i$
    $i \leftarrow i - 1$
$TS \leftarrow TS \setminus (AS \cup FTS)$
while $TS \neq \emptyset$ **do**
    $t\_max \leftarrow TS[1]$ ▷ any element from TS
    $tl\_max \leftarrow \emptyset$
    for all $t \in TS$ **do**
        $tl\_current \leftarrow C \cap TL_t$
        if $|tl\_current| > |tl\_max|$ **then**
            $t\_max \leftarrow t$
            $tl\_max \leftarrow tl\_current$
    $C \leftarrow C \setminus tl\_max$
    $P[t] \leftarrow i$
    $i \leftarrow i - 1$
return P

Chapter 5

Evaluation

This chapter will evaluate the performance of the previously presented VeloCIty framework. In the first section, the two test subjects that will be used in the subsequent experiments will be introduced. The next section will restate the research questions formally, and extend these. Afterwards, the procedure of how the data was obtained will be elaborated. The final section will provide answers to the research questions as well as present the results of applying Test Case Prioritisation to the test subjects.

5.1 Test subjects

5.1.1 Dodona

Dodona[1] is an open source online learning environment created by Ghent University, allowing students from secondary schools and universities in Belgium and South-Korea to submit solutions to programming exercises and receive automated feedback. The application is built using the Ruby-on-Rails web framework. In order to automate the testing process of the application, Dodona makes use of Github Actions (Figure 2.2.2) to run tests using the default `MiniTest` testing framework. The coverage of the test suite is being recorded by `SimpleCov`[2].

5.1.2 Stratego

Leg uit wat het is.

5.2 Research questions

The following research questions will be answered in the subsequent sections:

RQ1: What is the probability that a test run will contain at least one failed test case? The first research question will provide useful insights into whether a typical test run has a tendency towards failing or succeeding. It is expected that a typical test run will succeed and therefore that executing every test case is not strictly necessary.

[1] https://dodona.ugent.be/
[2] https://github.com/colszowka/simplecov

RQ2: If a test run has failed, what is the probability that the next run will fail as well? The ROCKET algorithm (subsection 3.2.3) relies on the assumption that if a test case has failed in a given run, it will most likely fail in the subsequent run as well. This research question will investigate the correctness of this hypothesis.

RQ3: What is the average duration of a test run? Measuring how long it takes to execute a typical test run is required to estimate the benefit of applying any form of test suite optimisation. Only successful test runs should be considered to reduce bias introduced by prematurely aborting execution.

RQ4: What is the performance of applying Test Case Prioritisation to Dodona? This research question will investigate how quickly a failing test case in the Dodona project can be discovered using VeloCIty.

RQ5: What is the performance of applying Test Case Prioritisation to Stratego? This research question will investigate how quickly a failing test case in the Stratego project can be discovered using VeloCIty.

5.3 Data collection

5.3.1 Travis CI build data

In order to answer the first three research questions, build data for several projects hosted on Travis CI (Figure 2.2.2) was used. This data was obtained from two sources.

The first source is a database of 35 793 144 jobs, provided by Durieux et al [8]. Due to the magnitude of this dataset (61.11 GiB), a big data framework is required to parse the log files. In order to collect the required data to provide an answer to the first and third research question, two simple `MapReduce` pipelines have been executed using the Apache Spark[3] framework..

In addition to the first source, another 3 702 595 jobs have been analysed from the *TravisTorrent* project. This project [3] scrapes the API of Travis CI and combines this with data obtained from the GitHub API to augment the dataset with extra information. One of these additional values is the identifier of the previous executed build of every project. This identifier is required to answer the second research question. Furthermore, the amount of failed test cases in every run is included, which can be used to distinguish whether the test run has actually failed or the project could not be

[3] `https://spark.apache.org/`

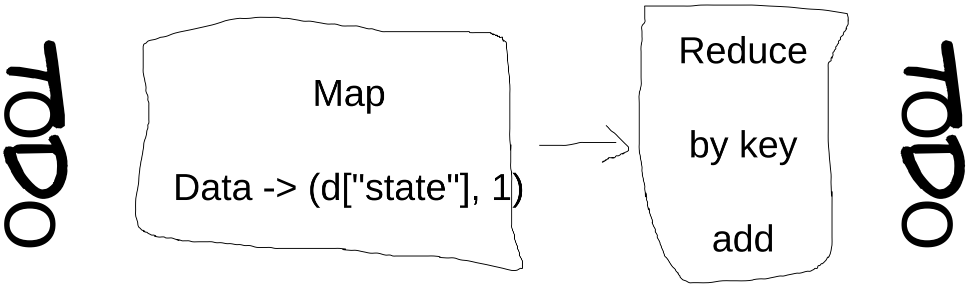


Figure 5.1: MapReduce pipeline to find the failed runs

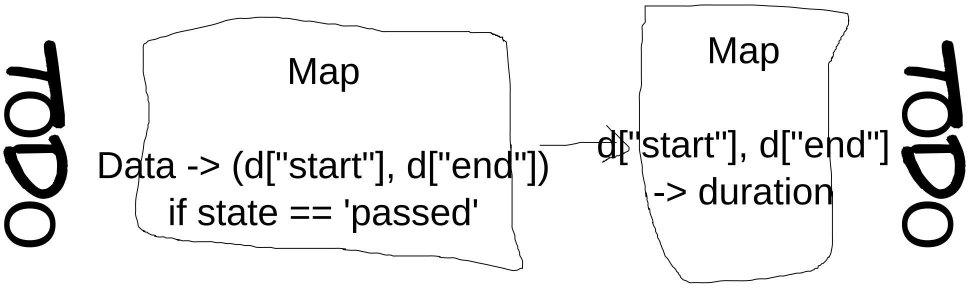


Figure 5.2: MapReduce pipeline to find the average duration of a successful test run

compiled and thus allowing a more fine-grained answer to the first research question as well. After analysis of this dataset, the execution time was not accurate compared to the actual timings on the Travis CI build page, so this dataset was not used in the third research question. For analysis, the creators of TravisTorrent have provided a Google BigQuery[4] interface to allow querying the dataset, which is required given its magnitude. The following queries have been executed to obtain the required data:

Listing 5.1: TravisTorrent query: Find the amount of failed runs

```
SELECT
  COUNTIF(tr_log_bool_tests_failed) as failed,
  COUNTIF(tr_log_bool_tests_ran) as ran,
  COUNT(1) as total
FROM `travistorrent`
```

Listing 5.2: TravisTorrent query: Find the probability of consecutive failures

```
(
  SELECT gh_build_started_at, true as failed
  FROM `travistorrent`
  WHERE
    tr_build_id IN (
      SELECT DISTINCT(tr_prev_build)
      FROM `travistorrent`
```

[4] https://bigquery.cloud.google.com/

```sql
      WHERE tr_log_bool_tests_ran=true AND tr_log_bool_tests_failed=true
    )
    AND gh_build_started_at IS NOT null AND tr_log_bool_tests_failed=true
) UNION ALL (
  SELECT gh_build_started_at, false as failed
  FROM `travistorrent`
  WHERE
    tr_build_id IN (
      SELECT DISTINCT(tr_prev_build)
      FROM `travistorrent`
      WHERE tr_log_bool_tests_ran=true AND tr_log_bool_tests_failed=false
    )
    AND gh_build_started_at IS NOT null AND tr_log_bool_tests_failed=true
)
```

5.3.2 Dodona build data

As mentioned before, Dodona makes use of the MiniTest testing framework and the SimpleCov coverage tracker. By default, MiniTest only emits which test cases have failed without any further information. Furthermore, SimpleCov is only capable of calculating the coverage for the entire test suite and does not allow to retrieve the coverage on a per-test basis. In order to answer the fourth research question which analyses the performance of applying the VeloCIty predictors to Dodona, a Python script has been created to repeat every failed test run in a modified way to allow timing the execution and tracking the individual test case coverage. Essentially, this script queries the API of GitHub Actions to find failed test runs, 62 failed runs were used in this thesis. For every failed commit, the script queries the API again to find the first successfully tested parent commit. These parent commits are used to obtain the coverage per test case on a codebase which resembles the failed commits as close as possible. After the appropriate parent commits have been identified, they are modified by applying the following two transformations and subsequently rescheduled in GitHub Actions:

- **Cobertura formatter:** The currently used coverage formatter is only capable of generating a HTML report of the coverage, which is not desirable for analysis. The Cobertura formatter on the other hand is able to generate XML reports instead, these are already supported by the Controller and Java agent.

- **Disable parallel execution:** By default, the test cases are executed concurrently and divided over four processes. However, since SimpleCov is not thread-safe, this would render the tracked coverage invalid.

5.4 Results

5.4.1 RQ1: Probability of failure

Figure 5.3 contains two pie charts that illustrate the amount of failed and successful test runs. The left chart contains the results of the dataset provided by Durieux et al [8]. This dataset contains 4 558 279 failed test runs versus 24 323 724 successful runs, which corresponds to a failure probability of 18.74 %. The other pie chart visualises data from the TravisTorrent project. According to this dataset, the run has failed prior to starting the test suite in 42.89 % of the executions. For the remaining part of the runs, 225 766 out of 2 114 920 executions contained at least one failed test case, corresponding to a failure percentage of 10.67 %.

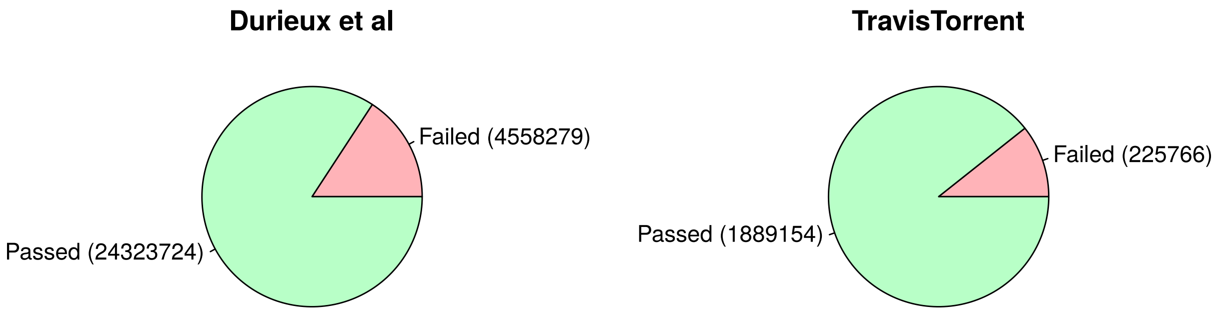


Figure 5.3: Probability of test run failure

5.4.2 RQ2: Probability of consecutive failure

In order to find consecutive failures, only the TravisTorrent project can be used as every entry in this dataset contains the identifier of the previous build which is required to link consecutive builds. The dataset contains 211 040 test runs of which the test suite of the preceding test run was both executed and contained at least one failed test case. As illustrated in Figure 5.4, 109 224 of these test runs failed as well, versus 101 816 test runs (51.76 %) that did succeed.

5.4.3 RQ3: Average test run duration

The Travis CI dataset provided by Durieux et al [8] has been filtered to exclude test runs with an execution time of less than 10 s, as this generally implies that the test

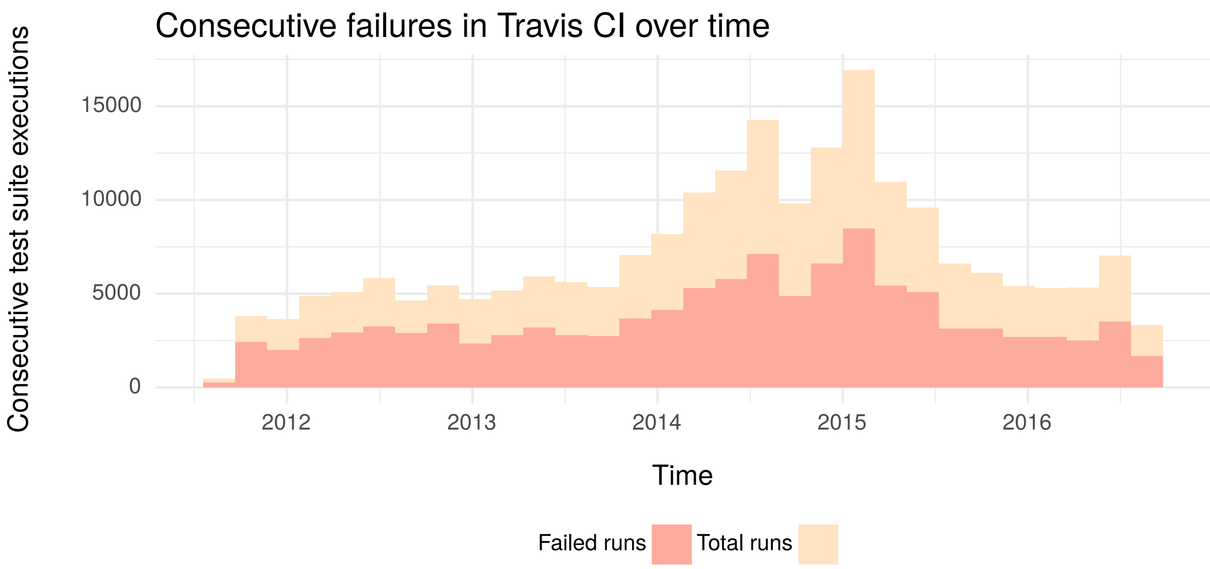


Figure 5.4: Consecutive test run failures on Travis CI

suite did not actually execute due to an initialisation failure. Table 5.1 contains the characteristics of the remaining analysed test runs. This table suggests that Travis CI is primarily used for small projects, yet the maximal value is a strong outlier. Figure 5.5 confirms the existence of 71 378 test runs with an execution time of more than one hour. Further investigation has pointed out that these are mostly projects which are using mutation testing (Listing 2.1.1), such as `plexus/yaks`[5].

# runs	Minimum	Mean	Median	Maximum
24 320 504	10 s	385 s	178 s	26 h11 min26 s

Table 5.1: Characteristics of the test run durations in [8].

5.4.4 RQ4: Applying Test Case Prioritisation to Dodona

Given the 62 collected test runs, another 9 runs have been omitted because these have been identified as a bug in the configuration of the test suite, preventing any test to be executed at all. This is something which cannot be detected by a prioritisation framework, since this requires more contextual information about the project.

Figure 5.3 compares the performance of respectively the Alpha algorithm, the Greedy algorithm, the HGS algorithm and the ROCKET algorithm to the original, non-prioritised execution. The Alpha and HGS algorithm provide the most accurate predictions, with the latter algorithm being the least consistent. The Greedy algorithm on the other hand succeeds in predicting some executions very accurately, while failing to predict

[5]A Ruby library for hypermedia (`https://github.com/plexus/yaks`).

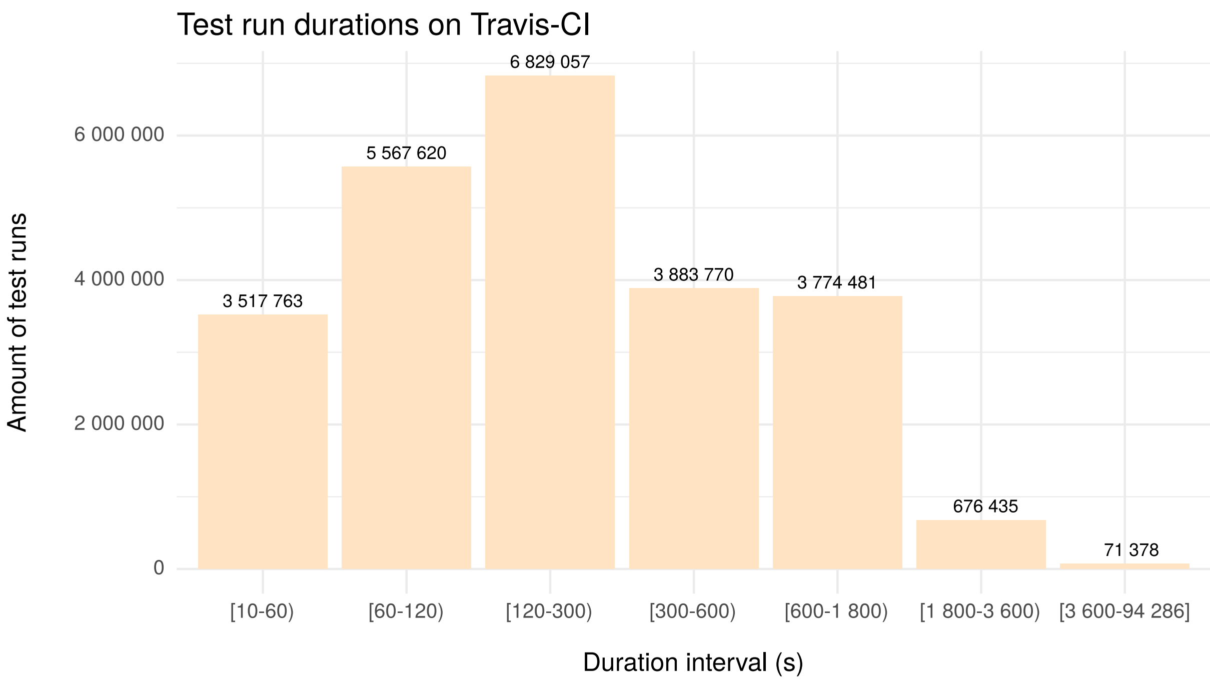
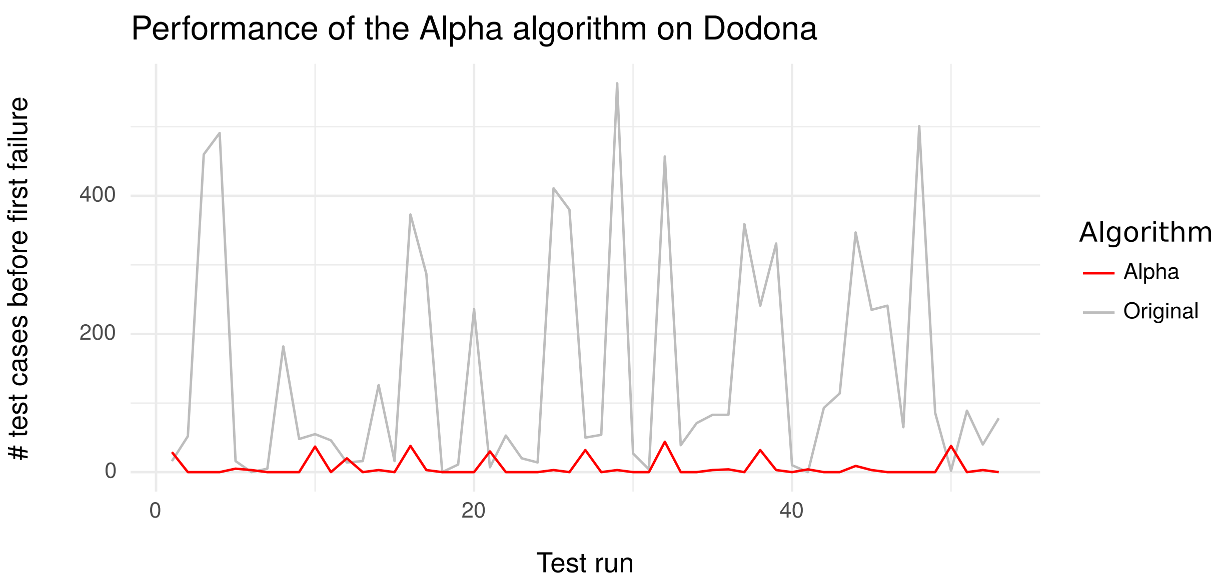


Figure 5.5: Test run durations on Travis CI

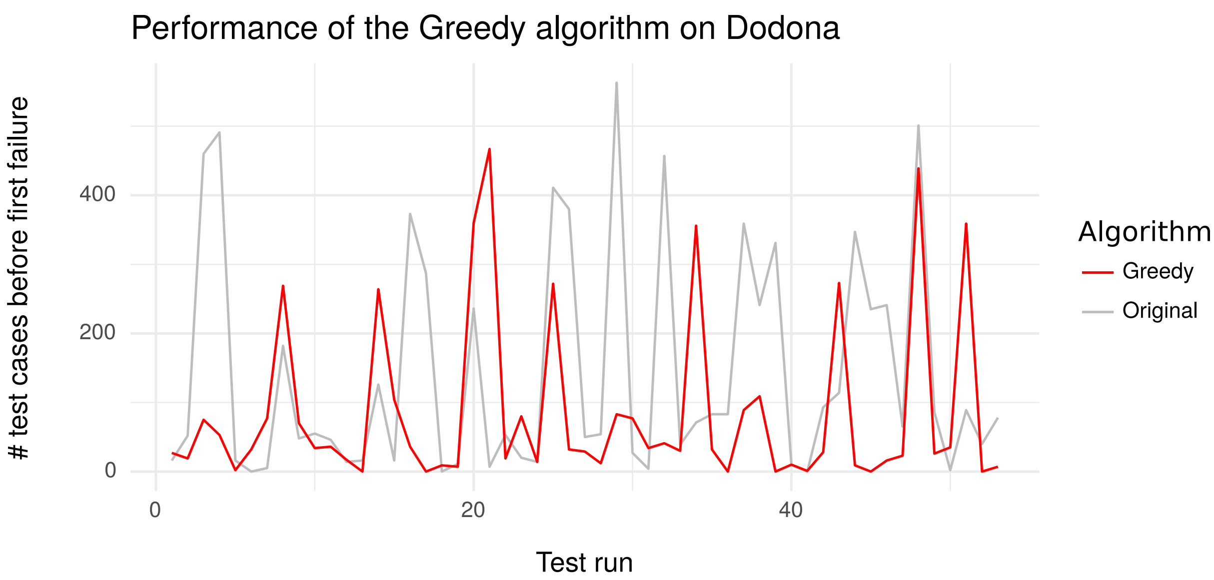
other runs anywhere near, which is the expected behaviour of a greedy heuristic. Finally, the ROCKET algorithm is not suitable for this project.



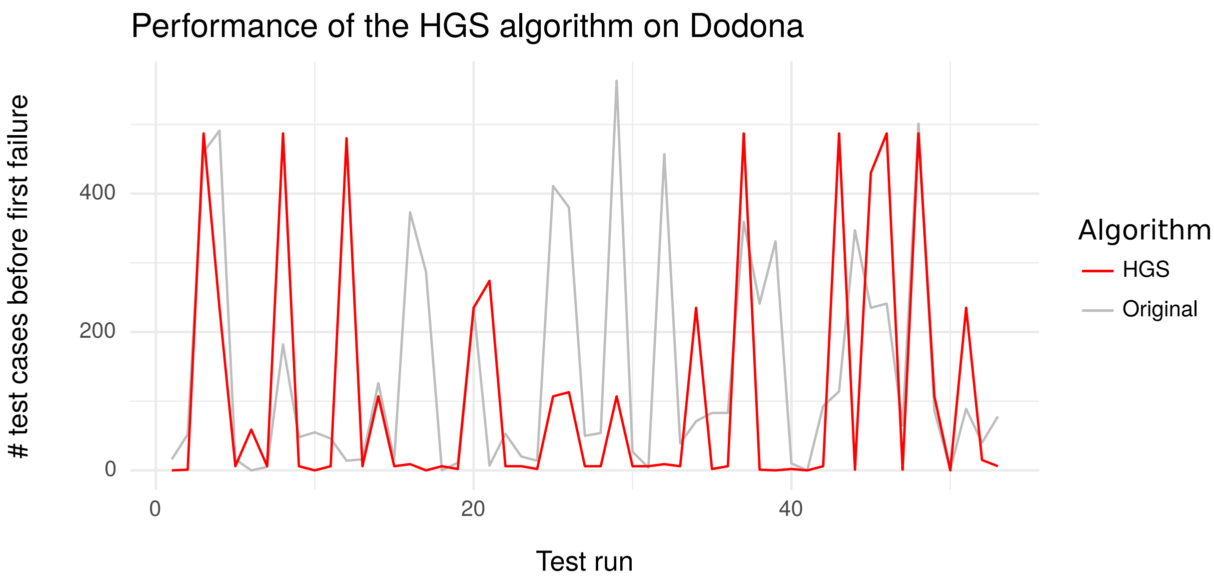
(a) Alpha algorithm

Table 5.2 contains the minimum, mean, median and maximum median amount of test cases until the first failure is observed. This table indicates that, except for the GreedyCoverAffected algorithm[6], every predictor is able to perform at least one successful prediction. Furthermore, the maximum amount of executed test cases is lower than

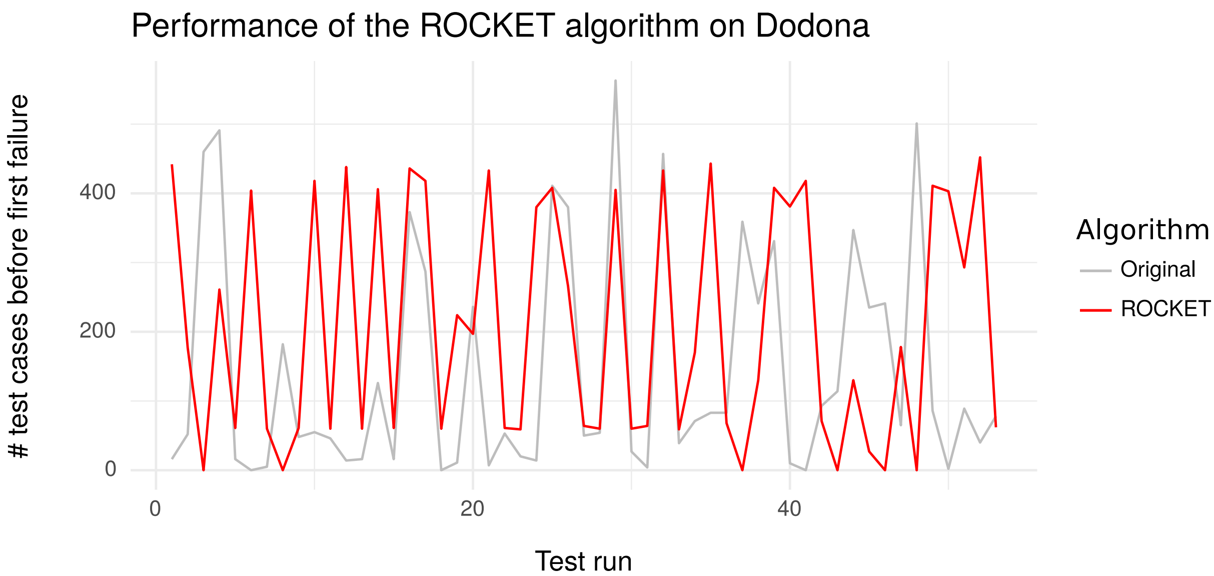
[6]The AllInOrder algorithm can be considered a deterministic random algorithm and therefore not an actual predictor.



(b) Greedy algorithm



(c) HGS algorithm



(d) ROCKET algorithm

Figure 5.3: Prediction performance on the Dodona project

the original, for every predictor. The previous paragraph has already observed that the Alpha and HGS algorithm provide the best prediction accuracy for Dodona, this hypothesis is confirmed by the low median and mean values for these algorithms. These values confirm as well that the ROCKET algorithm is not able of providing accurate predictions.

Algorithm	**Minimum**	**Mean**	**Median**	**Maximum**
Original	0	143	65	563
Alpha	0	7	6	44
AffectedRandom	0	82	18	428
AllInOrder	1	102	71	455
AllRandom	0	71	16	477
GreedyCoverAffected	31	307	296	446
GreedyCoverAll	0	85	32	467
GreedyTimeAll	0	209	172	452
HGSAffected	0	54	10	511
HGSAll	0	109	6	487
ROCKET	0	208	170	452

Table 5.2: Amount of executed test cases until the first failure.

Similarly, Table 5.3 contains the minimum, mean, maximum and median duration until the first failed test case is observed. This data further confirms the observations made in the previous paragraph and the effectiveness of the GreedyTimeAll predictor. Notice that the ROCKET algorithm performs better time-wise than quantity wise.

duration Original AllRandom Min. :118165 Min. : 0.00 Min. : 0.000 1st Qu.:195877 1st Qu.: 0.00 1st Qu.: 0.000 Median :297371 Median : 2.00 Median : 4.000 Mean

Algorithm	Minimum	Mean	Median	Maximum
Original	0 s	154 s	125 s	622 s
Alpha	0 s	5 s	0 s	84 s
AffectedRandom	0 s	64 s	10 s	355 s
AllInOrder	0 s	177 s	171 s	456 s
AllRandom	0 s	64 s	11 s	611 s
GreedyCoverAffected	6 s	123 s	109 s	264 s
GreedyCoverAll	0 s	60 s	25 s	306 s
GreedyTimeAll	0 s	44 s	15 s	178 s
HGSAffected	0 s	58 s	6 s	581 s
HGSAll	0 s	130 s	16 s	578 s
ROCKET	0 s	47 s	16 s	178 s

Table 5.3: Duration until the first failure.

:303527 Mean : 67.57 Mean : 9.257 3rd Qu.:414956 3rd Qu.:110.00 3rd Qu.:13.500 Max. :505996 Max. :278.00 Max. :64.000

AllInOrder GreedyCoverAll AffectedRandom Min. : 0.000 Min. : 0.000 Min. : 0.000 1st Qu.: 0.000 1st Qu.: 0.000 1st Qu.: 0.000 Median : 1.000 Median : 1.000 Median : 2.000 Mean : 9.229 Mean : 8.971 Mean : 8.886 3rd Qu.:12.000 3rd Qu.:13.000 3rd Qu.:11.500 Max. :37.000 Max. :44.000 Max. :45.000

HGSAffected GreedyCoverAffected HGSAll Min. : 0.000 Min. : 0.00 Min. : 0.0 1st Qu.: 0.000 1st Qu.: 0.00 1st Qu.: 0.0 Median : 3.000 Median : 2.00 Median : 7.0 Mean : 8.886 Mean :15.09 Mean : 9.6 3rd Qu.:11.500 3rd Qu.:31.00 3rd Qu.:12.5 Max. :64.000 Max. :43.00 Max. :43.0

Alpha Rocket GreedyTimeAll Min. : 0.000 Min. : 0.00 Min. : 0.00 1st Qu.: 0.000 1st Qu.: 3.50 1st Qu.: 3.50 Median : 1.000 Median : 27.00 Median : 27.00 Mean : 8.971 Mean : 42.23 Mean : 42.23 3rd Qu.:13.000 3rd Qu.: 49.00 3rd Qu.: 49.00 Max. :44.000 Max. :216.00 Max. :216.00

Original_ms AllRandom_ms AllInOrder_ms Min. : 0 Min. : 0 Min. : 0 1st Qu.: 0 1st Qu.: 0 1st Qu.: 0 Median : 9560 Median : 739 Median : 862 Mean : 73612 Mean : 13299 Mean : 32391 3rd Qu.:122586 3rd Qu.: 13592 3rd Qu.: 78063 Max. :283433 Max. :184805 Max. :139345

GreedyCoverAll_ms AffectedRandom_ms HGSAffected_ms Min. : 0 Min. : 0 Min. : 0 1st Qu.: 0 1st Qu.: 0 1st Qu.: 0 Median : 620 Median : 317 Median : 1318 Mean : 15251 Mean : 19732 Mean : 23738 3rd Qu.: 8620 3rd Qu.: 6460 3rd Qu.: 11786 Max. :197463 Max. :174491 Max. :211702

GreedyCoverAffected_ms HGSAll_ms Alpha_ms Min. : 0 Min. : 0 Min. : 0 1st Qu.: 0 1st Qu.: 0 1st Qu.: 0 Median : 4229 Median : 2818 Median : 647 Mean :21835 Mean : 12494 Mean : 15189 3rd Qu.:37885 3rd Qu.: 8475 3rd Qu.: 8386 Max. :75637 Max. :157003 Max. :193618

Rocket_ms GreedyTimeAll_ms idx Min. : 0 Min. : 0 Min. : 1.0 1st Qu.: 0 1st Qu.: 0 1st Qu.: 9.5 Median : 254 Median : 257 Median :18.0 Mean : 6844 Mean : 6852 Mean :18.0 3rd Qu.: 1760 3rd Qu.: 1754 3rd Qu.:26.5 Max. :100011 Max. :99765 Max. :35.0

Chapter 6

Conclusion

The main purpose of this thesis has been to investigate different approaches towards optimising the test suite of a common software project. The concepts of Test Suite Minimisation, Test Case Selection and Test Case Prioritisation have been introduced and accompanying algorithms have been presented. A novel client-server oriented framework for the latter approach has been proposed, as well as a new prioritisation algorithm. Finally, VeloCIty has been applied to the UGent Dodona project, proving its ability to predict test case failure and therefore reduce the execution time of the test suite.

A second purpose of this thesis was to gain useful insights into the behaviour of a typical test suite. These insights have been formulated as three additional research questions, to which answers have been provided in the previous chapter.

6.1 Future work

The proposed VeloCIty implementation in this thesis is currently able to prioritise a Gradle Java project using 10 available predictors and a meta predictor. While this is certainly functional, it is far from complete and multiple improvements can be added.

6.1.1 Java Agent

The existing Java Agent can be extended in multiple ways. The most prominent addition would be to allow test cases to be executed in parallel. At the moment of writing, this is not possible yet. In order to facilitate parallel testing, one must first decide how to schedule the prioritised test cases across multiple threads, since the execution time of a test case varies strongly. One possibility to perform this scheduling is to use the average execution time per test case, which is obtained from prior runs. Alternatively, this can be performed at runtime by using any existing inter-thread communication paradigm such as message passing. On the implementation side of parallelisation, the current `TestProcessor` should be adapted to inherit from the `MaxNParallelTestClassProcessor`. A thread pool should ideally be used to reduce the overhead of restarting a new thread for every test case.

6.1.2 Predictions

Further research and improvements to the predictors can be made on four different aspects.

The first enhancement is that currently the predictor does not discriminate between a unit test or an integration test. Recall that the scope of a unit test is limited to a small fraction of the application and that its execution time is ideally rather low. An integration test however usually takes longer to execute and tests multiple components of the application at once. The predictor could make use of this distinction and assume that a failure in a unit test has a high probability of resulting in a failed integration test as well, hence prioritising unit tests over integration tests.

Secondly, the prediction algorithms currently take into account which source code lines have either been modified or removed in order to prioritise affected test cases. Likewise, test cases of which the code has been modified should also be considered as candidates for prioritisation, as the changed test case might contain a bug as well.

A third and unexplored research opportunity is to investigate the joint performance of multiple prediction algorithms combined. This could be integrated with the existing meta predictor. Instead of assigning a score to the entire prediction, multiple predictions could be intermingled using predefined weights.

The final improvement is to take into account branch coverage in addition to the statement coverage which is currently used. This is a rather complex feature as not every coverage framework is capable of reporting accurately which branches have been covered and which ones have not. A suggested implementation would be to instrument the source code and rewrite every condition of every branch as separate `if`-statements.

6.1.3 Meta predictor

The proposed meta predictor increases the score of every predictor which predicted an above-average ranking and decreases the score of the other predictors. However, a possible problem with this approach is that the nature of the source code might evolve and change as time progresses. Using the current updating strategy it will take several test suite invocations for an alternative predictor to be preferred by the meta predictor. If a saturating counter would be used instead (Figure 6.1), this would be resolved much more quickly, allowing a more versatile meta predictor.

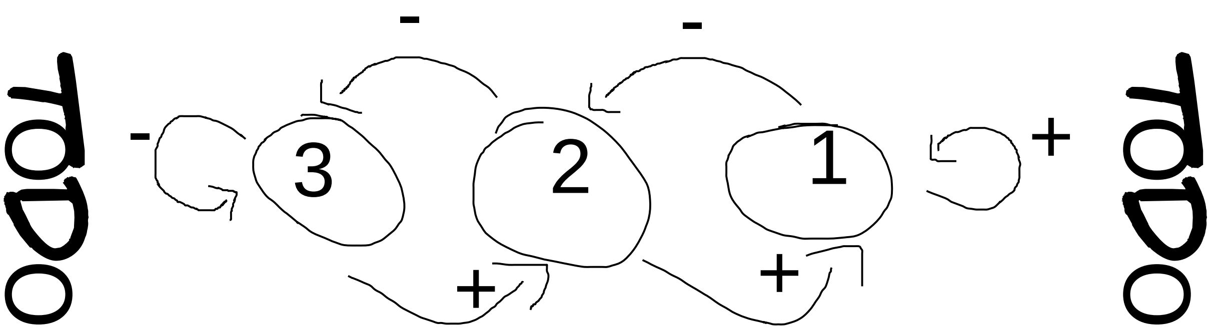


Figure 6.1: Saturating counter

In addition to implementing a different update strategy, it might be worth to investigate the use of machine learning or linear programming models as a meta predictor, or even as a prediction algorithm.

6.1.4 Final enhancements

Finally, since some of the implemented algorithms are inherently Test Suite Minimisation algorithms rather than prioritisation algorithms, the framework might opt to not execute some test cases at all, whereas now the entire test suite is always executed.

Support for other programming languages and frameworks is possible by implementing new agents. The basic implementation is straightforward to restart the test suite after every executed test case, should test case reordering not be supported natively by the test framework.

Bibliography

[1] *About GitHub Actions*. URL: https://help.github.com/en/actions/getting-started-with-github-actions/about-github-actions.

[2] Mohammed Arefeen and Michael Schiller. "Continuous Integration Using Gitlab". In: *Undergraduate Research in Natural and Clinical Science and Technology (URNCST) Journal* 3 (Sept. 2019), pp. 1–6. DOI: 10.26685/urncst.152.

[3] Moritz Beller, Georgios Gousios, and Andy Zaidman. "TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration". In: *Proceedings of the 14th working conference on mining software repositories*. 2017.

[4] H.D. Benington. *Production of large computer programs*. ONR symposium report. Office of Naval Research, Department of the Navy, 1956, pp. 15–27. URL: https://books.google.com/books?id=tLo6AQAAMAAJ.

[5] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844.

[6] Michael Cusumano, Akindutire Michael, and Stanley Smith. "Beyond the waterfall : software development at Microsoft". In: (Feb. 1995).

[7] Charles-Axel Dein. *dein.fr*. Sept. 2019. URL: https://www.dein.fr/2019-09-06-test-coverage-only-matters-if-at-100-percent.html.

[8] Thomas Durieux et al. "An Analysis of 35+ Million Jobs of Travis CI". In: (2019). DOI: 10.1109/icsme.2019.00044. eprint: arXiv:1904.09416.

[9] *Features • GitHub Actions*. URL: https://github.com/features/actions.

[10] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. USA: W. H. Freeman & Co., 1990. ISBN: 0716710455.

[11] *GitLab CI/CD*. URL: https://docs.gitlab.com/ee/ci/.

[12] *GitLab Continuous Integration & Delivery*. URL: https://about.gitlab.com/stages-devops-lifecycle/continuous-integration/.

[13] A. Govardhan. "A Comparison Between Five Models Of Software Engineering". In: *IJCSI International Journal of Computer Science Issues 1694-0814* 7 (Sept. 2010), pp. 94–101.

[14] Standish Group et al. "CHAOS report 2015". In: *The Standish Group International* (2015). URL: https://www.standishgroup.com/sample_research_files/CHAOSReport2015-Final.pdf.

[15] M. Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. "A Methodology for Controlling the Size of a Test Suite". In: *ACM Trans. Softw. Eng. Methodol.* 2.3 (July 1993), pp. 270–285. ISSN: 1049-331X. DOI: 10.1145/152388.152391. URL: https://doi.org/10.1145/152388.152391.

[16] Naftanaila Ionel. "AGILE SOFTWARE DEVELOPMENT METHODOLOGIES: AN OVERVIEW OF THE CURRENT STATE OF RESEARCH". In: *Annals of Faculty of Economics* 4 (May 2009), pp. 381–385.

[17] "ISO/IEC/IEEE International Standard - Software and systems engineering –Software testing –Part 1:Concepts and definitions". In: *ISO/IEC/IEEE 29119-1:2013(E)* (Sept. 2013), pp. 1–64. DOI: 10.1109/IEEESTD.2013.6588537.

[18] "ISO/IEC/IEEE International Standard - Systems and software engineering – System life cycle processes". In: *ISO/IEC/IEEE 15288 First edition 2015-05-15* (May 2015), pp. 1–118. DOI: 10.1109/IEEESTD.2015.7106435.

[19] "ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary". In: *ISO/IEC/IEEE 24765:2017(E)* (Aug. 2017), pp. 1–541. DOI: 10.1109/IEEESTD.2017.8016712.

[20] Y. Jia and M. Harman. "An Analysis and Survey of the Development of Mutation Testing". In: *IEEE Transactions on Software Engineering* 37.5 (2011), pp. 649–678.

[21] N. Landry. *Iterative and Agile Implementation Methodologies in Business Intelligence Software Development*. Lulu.com, 2011. ISBN: 9780557247585. URL: https://books.google.be/books?id=bUHJAQAAQBAJ.

[22] G. Le Lann. "An analysis of the Ariane 5 flight 501 failure-a system engineering perspective". In: *Proceedings International Conference and Workshop on Engineering of Computer-Based Systems*. Mar. 1997, pp. 339–346. DOI: 10.1109/ECBS.1997.581900.

[23] Simon Maple. *Development Tools in Java: 2016 Landscape*. July 2016. URL: https://www.jrebel.com/blog/java-tools-and-technologies-2016.

[24] D. Marijan, A. Gotlieb, and S. Sen. "Test Case Prioritization for Continuous Regression Testing: An Industrial Case Study". In: *2013 IEEE International Conference on Software Maintenance*. 2013, pp. 540–543.

[25] Robert C. Martin and Micah Martin. *Agile Principles, Patterns, and Practices in C# (Robert C. Martin)*. USA: Prentice Hall PTR, 2006. ISBN: 0131857258.

[26] Bertrand Meyer. "Overview". In: *Agile!: The Good, the Hype and the Ugly*. Cham: Springer International Publishing, 2014, pp. 1–15. ISBN: 978-3-319-05155-0. DOI: 10.1007/978-3-319-05155-0_1. URL: https://doi.org/10.1007/978-3-319-05155-0_1.

[27] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. 3rd. Wiley Publishing, 2011. ISBN: 1118031962.

[28] Dor Nir, Shmuel Tyszberowicz, and Amiram Yehudai. “Locating Regression Bugs”. In: *Hardware and Software: Verification and Testing*. Ed. by Karen Yorav. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 218–234. ISBN: 978-3-540-77966-7.

[29] Raphael Noemmer and Roman Haas. “An Evaluation of Test Suite Minimization Techniques”. In: Dec. 2019, pp. 51–66. ISBN: 978-3-030-35509-8. DOI: 10.1007/978-3-030-35510-4_4.

[30] A. Jefferson Offutt and Roland H. Untch. “Mutation 2000: Uniting the Orthogonal”. In: *Mutation Testing for the New Century*. Ed. by W. Eric Wong. Boston, MA: Springer US, 2001, pp. 34–44. ISBN: 978-1-4757-5939-6. DOI: 10.1007/978-1-4757-5939-6_7. URL: https://doi.org/10.1007/978-1-4757-5939-6_7.

[31] W. W. Royce. “Managing the Development of Large Software Systems: Concepts and Techniques”. In: *Proceedings of the 9th International Conference on Software Engineering*. ICSE ’87. Monterey, California, USA: IEEE Computer Society Press, 1987, pp. 328–338. ISBN: 0-89791-216-0. URL: http://dl.acm.org/citation.cfm?id=41765.41801.

[32] John Ferguson Smart. *Jenkins: The Definitive Guide*. Beijing: O’Reilly, 2011. ISBN: 978-1-4493-0535-2. URL: https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/.

[33] Travis. *Travis CI - Test and DeployYour Code with Confidence*. Feb. 2020. URL: https://travis-ci.org.

[34] Kristen R. Walcott et al. “TimeAware Test Suite Prioritization”. In: *Proceedings of the 2006 International Symposium on Software Testing and Analysis*. ISSTA ’06. Portland, Maine, USA: Association for Computing Machinery, 2006, pp. 1–12. ISBN: 1595932631. DOI: 10.1145/1146238.1146240. URL: https://doi.org/10.1145/1146238.1146240.

[35] James Whittaker. “What is software testing? And why is it so hard?” In: *Software, IEEE* 17 (Feb. 2000), pp. 70–79. DOI: 10.1109/52.819971.

[36] S. Yoo and M. Harman. “Regression Testing Minimization, Selection and Prioritization: A Survey”. In: *Softw. Test. Verif. Reliab.* 22.2 (Mar. 2012), pp. 67–120. ISSN: 0960-0833. DOI: 10.1002/stv.430. URL: https://doi.org/10.1002/stv.430.

List of Figures

List of Listings

List of Tables