

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



The author gives the permission to use this thesis for consultation and to copy parts of it for personal use. Every other use is subject to the copyright laws, more specifically the source must be extensively specified when using from this thesis.

De auteur geeft de toelating deze masterproef voor consultatie beschikbaar te stellen en delen van de masterproef te kopiëren voor persoonlijk gebruik. Elk ander gebruik valt onder de bepalingen van het auteursrecht, in het bijzonder met betrekking tot de verplichting de bron uitdrukkelijk te vermelden bij het aanhalen van resultaten uit deze masterproef.

Pieter De Clercq – June 10, 2020.

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 this subject and for their prompt and clear responses to every question I have asked. I especially want to thank you for permitting me 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 with the opportunity to pursue my childhood dreams.

Last, but surely 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 my wildest dreams while protecting me against my often unrealistic ideas and ambition to excel. Helena for never leaving my side, for always making me laugh when I don't want to, and most importantly to remind me that I should relax from time to time. Jana for my daily dose of laughter, fun and for her inexhaustible positivity. Tobiah for the endless design discussions and for outperforming me in almost every school project, to encourage me to continuously raise the bar and to never give up. Finally, I want to thank Femke, Doortje and Freija for answering my mathematical questions, regularly asking about my thesis progression and motivating me to persevere throughout my entire education.

Thank you.

Pieter – Ghent, 2020

Summary

In traditional software engineering, developers typically build the entire application as one monolith by designing, implementing and testing everything sequentially. Each of these steps consumes a considerable amount of time, making software engineering a costly and time-intensive activity. In the wake of the world economic crisis, however, software developers have been forced to diminish their expenses drastically. The most viable way to reduce financial risks is to release a small version of the application as soon as possible and incrementally extend its functionality, a philosophy known as Agile Software Development.

While this approach entails several short-term benefits, a problem will still emerge in the long run. Frequent and shorter release cycles imply the need for automated testing solutions where human interaction is no longer strictly necessary. This process exists under the name of Continuous Integration, although it is not a silver bullet. To fully automate the testing process, the application needs to contain many test cases that guarantee everything is working as intended.

However, since the developers add new features to the application at a rapid pace, the number of test cases will increase at least as fast, since every code change requires at least one test case. As a result, the time it takes to (automatically) execute all the test cases will increase superlinearly as well, nullifying the benefits of maintaining a short release cycle.

The solution to this scalability problem is the introduction of optimisation techniques. This thesis presents three techniques: Test Suite Minimisation, Test Case Selection and Test Case Prioritisation. The objective of the first two techniques is to exclude test cases which will probably not fail from being executed. The latter technique does execute every test case but determines an execution sequence that prioritises test cases based on their probability of failing in the current run.

This technique has been implemented in a framework which features three existing algorithms and one new prioritisation algorithm. Next, the effect of applying this technique has been evaluated on two existing applications. The results are promising, resulting in ten times less executed test cases and up to thirty times faster detection of failing test cases.

Samenvatting

In een traditioneel softwareontwikkelingsproces bouwen programmeurs gewoonlijk de volledige applicatie in één keer. Deze applicatie is het eindresultaat van een driedelige procedure die begint met een functionele analyse. Vervolgens wordt de applicatie geïmplementeerd en ten slotte grondig getest. Elk van deze stappen neemt een aanzienlijke hoeveelheid tijd in beslag, waardoor softwareontwikkeling een zeer dure en langdradige aangelegenheid is. Sinds de wereldwijde economische crisis zijn ontwikkelaars echter genoodzaakt om drastisch te besparen op hun uitgaven. De gemakkelijkste manier om dit te bereiken is zo snel mogelijk een minimale versie van de applicatie uit te brengen met enkel de essentiële functionaliteit en deze vervolgens geleidelijk aan uit te breiden. Dit idee staat bekend als Agile Softwareontwikkeling, waarbij “agile” duidt op flexibiliteit.

Hoewel deze aanpak op korte termijn de financiële risico's vermindert, doet er zich op de langere termijn een ander probleem voor. Om een frequente ontwikkelingscyclus te kunnen hanteren, is er nood aan een mogelijkheid om tests automatisch uit te voeren, zonder menselijke tussenkomst. Dit is mogelijk met behulp van Continue Integratie, maar dit is geen wondermiddel. Om het testproces volledig te kunnen automatiseren, dienen er voldoende en vooral adequate tests aanwezig te zijn.

Softwareontwikkeling vandaag de dag gebeurt razendsnel. Het tempo waaraan ontwikkelaars applicaties uitbreiden, laat ook het aantal tests superlineair toenemen. Elke verandering aan de code van de applicatie leidt immers tot de toevoeging van minstens één extra test. Dit heeft een negatief effect op de uitvoeringstijd van de tests, waardoor de voordelen van een korte ontwikkelingscyclus tenietgedaan worden.

Dit probleem van schaalbaarheid kan opgelost worden door gebruik te maken van optimalisatietechnieken. Deze masterproef stelt drie technieken voor: Testpakket Minimalisering, Test-Selectie en Test-Prioritering. De eerste twee technieken proberen te voorspellen welke tests zullen slagen en voeren deze redundante tests bijgevolg niet uit. De derde techniek voert wel elke test uit, maar bepaalt een ideale uitvoeringsvolgorde zodanig dat tests met een hoge kans op falen eerder worden uitgevoerd.

Deze masterproef presenteert een implementatie van deze techniek met drie bestaande en een nieuw prioriteringsalgoritme. Het effect van deze techniek is geëvalueerd op twee bestaande applicaties. De resultaten zijn veelbelovend. De eerste falende test wordt gemiddeld dertig keer sneller gedetecteerd dan zonder deze techniek.

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—Ever since the introduction of traditional software development models in the previous century, the complexity and magnitude of today's software have vastly increased. This evolution has led to the adoption of Agile software development approaches, which pose the need for frequent integration and automated testing. Eventually, this increase in size will also negatively affect the size of the testing suites, resulting in severe scalability issues. This thesis proposes a framework and a novel algorithm for test suite optimisation by prioritising test cases which are likely to fail. The performance has been evaluated on two existing applications, and the results are auspicious.

Keywords—Continuous Integration, test suite, performance, optimisation, prioritisation

I. INTRODUCTION

The most characteristic trait of modern times is the astonishing speed at which everything in this world is evolving. This statement holds in particular for the field of computer science, where new technologies emerge almost every single day. While these inventions often seem primarily hardware-related, such as smartwatches, self-driving cars, or even biological technology, they cannot function without an even more sophisticated software component that controls them. As such, both the complexity and size of software have grown exponentially over the last decades.

Software engineers have experienced that the traditional software methodologies are no longer sufficient to handle this fast-paced development and, as a result, have shifted towards other strategies. Instead of implementing the entire application at once, developers now prefer the Agile approaches [?]. The Agile methodologies depict that an initial version of the application should be released as soon as possible and extend its functionality incrementally, to reduce the financial risks taken. A report of The Standish Group confirms that the adoption of these new approaches has led to a decreasing failure rate for new projects [?].

However, this evolution is not necessarily a good unfolding. Managing to build a project is one thing, guaranteeing it is built reliably is a different matter. An increase in the complexity of software inevitably makes the software more error-prone. As an attempted solution, the Agile approaches propose Continuous Integration (CI) [?]. This practice requires that the test

suite of the application is executed after every code change (Figure 1). Multiple CI-services have been created to assist in this process by means of automation. Optionally, every passed version can automatically be released to the end-users (Continuous Deployment).

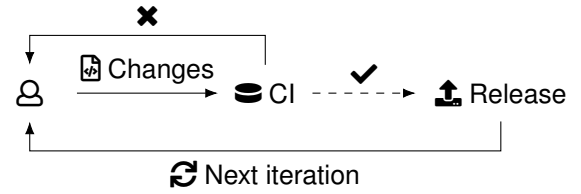


Figure 1: Continuous Integration.

Yet, in the long run, a new problem will emerge. As mentioned before, the size of applications increases exponentially. Every change in the code must be followed by the introduction of one or more test cases to guarantee the correctness. Therefore, the size of the test suite will increase even faster than the size of the project itself, which leads to severe scalability issues.

II. TECHNIQUES

This thesis presents three techniques to tackle this problem.

A. Test Suite Minimisation (TSM)

The first technique is called Test Suite Minimisation (TSM) [?]. TSM attempts to reduce the size of the test suite by permanently removing redundant test cases according to the following definition:

Given:

- $T = \{t_1, \dots, t_n\}$ a test suite consisting of test cases t_j .
- $R = \{r_1, \dots, r_m\}$ a set of requirements that must be satisfied in order to provide the desired “adequate” testing of the program.
- $\{T_1, \dots, T_m\}$ subsets of test cases in T , such that any one of the test cases $t_j \in T_i$ can be used to satisfy requirement r_i .

TSM is then defined as the task of finding a subset T' of test cases $t_j \in T$ that satisfies every requirement r_i .

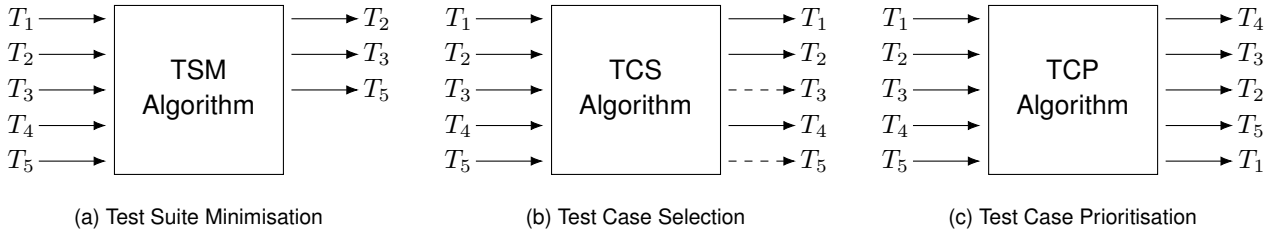


Figure 2: Illustration of the techniques.

B. Test Case Selection (TCS)

Instead of permanently removing redundant test cases, it is also possible to analyse the code changes and deduce which test cases should be executed and which might ones be omitted altogether. This technique is referred to as Test Case Selection (TCS) [?].

Given:

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

TCS aims to find a subset $T' \subseteq T$ that is used to test P' .

C. Test Case Prioritisation (TCP)

While TSM and TCS execute only the relevant test cases, sometimes it might be required that every test case is successfully executed. Consider, for example, critical software for medical purposes. In this case, it is still possible to optimise the test suite by executing all test cases in a specific order. Test Case Prioritisation (TCP) [?] constructs an ordered sequence that aims to fulfil a predetermined objective as fast as possible. This thesis primarily uses the detection of the first failed test case as the objective.

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 test cases to find the optimal permutation.

TCP finds a permutation $T' \in PT$ such that $\forall T'' \in PT : f(T') \geq f(T'') \Rightarrow (T'' \neq T')$.

III. ALGORITHMS

This thesis prefers to use TCP since this technique does not incur the risk of false-negative failing test cases. In order to determine the optimal order of execution, this thesis presents three existing algorithms.

The input data for these algorithms is threefold:

- **Affected test cases:** By combining previous coverage results and the list of changes that the developer has made to the code, the framework can estimate which test cases are likely affected by those changes and assign a higher priority.
- **Historical data:** Next, historical failure data can be used. Suppose that a test case has failed in its previous execution, then there exists a chance that it will subsequently fail in the current run. Either way, this test case should be executed early to verify that the underlying issue has been resolved.
- **Execution timings:** Finally, if two test cases are considered equally likely to fail, the average duration of the test case can be used as a tie-breaker. Since the objective of TCP is to optimise the test suite, the test case with the lowest duration should be preferred.

A. Greedy algorithm

The first algorithm is a greedy heuristic that was initially designed as an algorithm for the set-covering problem [?]. This heuristic starts with an empty set of test cases and the set of all code lines C . Next, the algorithm iteratively selects the test case that contributes the most code lines that are not yet covered, updating C after every selected test case. The algorithm halts when either all test cases are selected or C is empty. In order to modify this algorithm to make it applicable to TCP, the selection order must be preserved and used as the prioritised sequence.

B. HGS algorithm

The second algorithm was created by Harrold, Gupta and Soffa [?]. As opposed to the greedy heuristic, this algorithm uses a different perspective. First, the algorithm sorts the code lines increasingly based on the number of test cases that cover them. The motivation for this sorting operation is that some test cases must inevitably be executed, as they are the only test cases that cover a given set of lines. However, these test cases can also cover other lines and therefore make other test cases redundant. The algorithm iterates over the code lines in this order and selects one corresponding test case in each iteration. Afterwards, the order is updated to remove source code lines which are now covered by the selected test case. This process is repeated until there are no lines left to cover.

C. ROCKET algorithm

Finally, this thesis considers the ROCKET algorithm [?]. This algorithm prioritises test cases by assigning a score to every test case, which is calculated using historical failure data. Afterwards, the algorithm computes the cumulative score CS_t of every test case t and defines the following objective function g , in which E_t represents the execution time of the test case:

$$g = (\text{maximise}(CS_t), \text{minimise}(E_t))$$

Finally, the algorithm optimises this function to determine the ideal order of execution S , as follows:

$$(\forall i \in 1 \dots n)(g(S_i) \geq g(S_{i+1}))$$

IV. FRAMEWORK

This thesis proposes VeloCity as a language-agnostic framework that enables Test Case Prioritisation on existing software projects. The architecture consists of three main components, a meta predictor, and a novel prioritisation algorithm.

A. Agent

The first component is the agent. This component hooks into the testing framework of the application to execute the test cases in the required order. The agent obtains the optimal execution sequence by communicating with the next component, which is the controller.

B. Controller

The controller is a daemon which performs two tasks. First, the controller listens for requests from agents and acts as a relay to the predictor daemon. Additionally, the controller receives feedback from the agent after every executed test run. This information is used to update the meta predictor, which will be described later.

C. Predictor

The final main component of the architecture is the predictor daemon. This component is responsible for interpreting the code changes and determining the optimal order of execution using ten built-in prediction algorithms. These algorithms are variations of the three discussed algorithms, as well as the Alpha algorithm.

D. Meta predictor

Since the predictor daemon contains multiple prediction algorithms, a small extra component is required to decide which sequence should be preferred as the final execution order. The meta predictor is a table which assigns a score to every prediction algorithm.

The final execution order is the one that has been predicted by the prediction algorithm with the highest score. The controller evaluates the performance of every algorithm in the feedback phase, as mentioned before, and updates the scores accordingly.

E. Alpha algorithm

In addition to the Greedy, HGS and ROCKET algorithms, the framework features a custom prioritisation algorithm as well. This algorithm starts by inspecting the changed code lines to obtain the affected test cases ATS . Among ATS , the algorithm selects every test case that has failed at least once in its previous three executions and sorts those by increasing execution time. Next, the algorithm selects the remaining test cases from ATS and sorts those equivalently. After these two steps, the algorithm proceeds like the greedy algorithm until it has processed every test case.

V. RESULTS

The framework has been installed on two existing test subjects. The first project is the Ghent University Dodona project¹, on which the performance of the prediction algorithms has been evaluated. Since the provided version of the agent only supports Java and the Dodona project uses Ruby-on-Rails, a second test subject is required to verify the architecture. For this purpose, the Stratego² project was used. Additionally, this thesis answers three research questions to obtain insights into a typical test suite.

A. Performance

Table 1 compares the performance of the four discussed algorithms on two aspects. The first aspect is the amount of executed test cases until the first failure. Secondly, the duration until the first failed test case is measured. These results indicate that the Alpha algorithm executes around 30 times fewer test cases and that the first failure is observed almost instantaneously. The Greedy and HGS algorithms reduce the amount of executed test cases by almost half and offer a significant speed-wise improvement as well. The performance of the ROCKET algorithm is remarkable. The algorithm executes much more test cases than the original median but does detect a failing test case almost four times faster.

B. Research questions

This thesis answers three research questions using data from the Travis CI service. This data has been provided by the TravisTorrent project [?] (3 702 595 jobs)

¹<https://dodona.ugent.be/>

²Java Spring application created for the Software Engineering 2 course.

Algorithm	Median (tests)	Median (time)
<i>Original</i>	78	123 s
Alpha	3	1 s
Greedy	33	12 s
HGS	10	6 s
ROCKET	170	32

Table 1: Performance on the Dodona project.

and by Durieux et al. [?] (35 793 144 jobs).

RQ1: Probability of failure. The first research question analyses the probability that a test run will fail. In the provided datasets, a combined 15 % of all the runs have failed.

RQ2: Average test run duration. After inspecting the TravisTorrent dataset, the timing information was proven inaccurate. Therefore, only the dataset provided by Durieux et al. has been used to answer this question. This dataset has revealed that Travis CI is used primarily for small projects, with an average execution time of 385 s. The maximum duration is more than 26 h.

RQ3: Consecutive failure. The final research question investigates the probability that a test run will fail multiple times in a row. According to the TravisTorrent dataset, which contains the mandatory identifier of the previous test run of the same project, the probability of consecutive failure is more than 51.76 %.

VI. CONCLUSION AND FUTURE WORK

This thesis has proven that software projects can benefit from an optimised test suite, in particular, using Test Case Prioritisation. The proposed framework has been implemented successfully on two projects, and its results are promising, but it still has some limitations and room for improvements.

Agent. The provided implementation of the Java agent does not support parallel test case execution. Besides the technical difficulties, a coordination mechanism is required to enable this behaviour, to schedule test cases across multiple threads effectively.

Predictor. The prediction algorithms do not currently interact with one another. However, there might be hidden potential in merging the output sequences of multiple algorithms, possibly by using weights.

Meta predictor. The elementary meta predictor could be modified to use a saturating counter or Machine Learning to cope with an evolving codebase. Additionally, Machine Learning algorithms could potentially be used as prediction algorithms.

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—Sinds de introductie van traditionele softwareontwikkelingsmodellen in de vorige eeuw is de omvang en de complexiteit van software alsmaar toegenomen. Deze ontplooiingen hebben softwareontwikkelaars ertoe aangezet om over te schakelen op zogenaamde Agile ontwikkelingsmethoden, waarin frequent samenvoegen van code en automatisch testen centraal staan. Op lange termijn zal de stijgende omvang van software echter een bijkomend negatief effect hebben op de grootte van het testpakket¹. Om dit schaalbaarheidsprobleem op te lossen, stelt deze masterproef een framework en een nieuw prioriteringsalgoritme voor. Hierbij worden tests gerangschikt volgens de kans op falen. Het framework is geëvalueerd op twee bestaande applicaties en de resultaten zijn veelbelovend.

Trefwoorden—Continue Integratie, test suite, prestatie, optimalisatie, prioritering

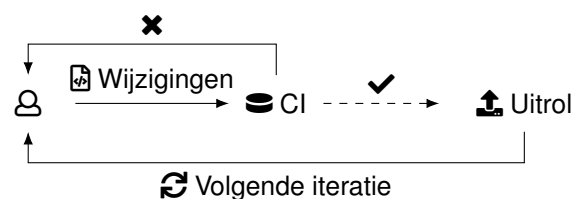
I. INTRODUCTIE

Kenmerkend aan de hedendaagse wereld is de verbaazingwekkende snelheid waartegen alles in deze wereld verandert. Dit geldt in het bijzonder voor de informatica, waar elke dag nieuwe ontwikkelingen plaatsvinden. Hoewel deze ontwikkelingen vooral hardware gerelateerd ogen, denk maar aan smartwatches, zelfrijdende auto's, biologische technologie, kunnen ze niet functioneren zonder een nog meer geavanceerde softwarecomponent. Bijgevolg is zowel de omvang als de complexiteit van software de laatste decennia exponentieel toegenomen.

Softwareontwikkelaars hebben ondervonden dat de traditionele ontwikkelingsmethoden deze evolutie niet kunnen bijbenen en hebben hun focus verlegd naar andere strategieën. Eerder dan de volledige applicatie in één keer te implementeren, verkiezen ontwikkelaars vandaag de dag de Agile methoden [1]. Deze aanpak schrijft voor dat het primaire doel moet zijn om zo snel mogelijk een minimale versie van de applicatie op de markt te brengen, om de financiële risico's te verkleinen, en achteraf extra functionaliteit toe te voegen. Een rapport van The Standish Group bevestigt dat de kans op falen aanzienlijk kleiner is bij het hanteren van een Agile ontwikkelingsmethode [2].

Deze evolutie draagt echter ook negatieve gevolgen met zich mee. Het is één zaak om een project succesvol af te leveren, maar daarmee is de betrouwbaarheid

nog niet gegarandeerd. Meer complexe software leidt onweerlegbaar tot software die meer vatbaar is voor fouten. De Agile aanpak tracht dit probleem op te lossen door middel van Continue Integratie (CI) [3]. Dit idee vereist dat het volledige testpakket (succesvol) wordt uitgevoerd na elke aanpassing aan de code (Figuur 1). Vandaag de dag bestaan een legio aan CI-services die dit proces vergemakkelijken door middel van automatisatie. Als onderdeel van dit automatisatieproces kan optioneel worden aangevuld met het automatisch uitrollen van nieuwe versies naar de eindgebruikers (Continuous Deployment).



Figuur 1: Continue Integratie.

Desalniettemin zal er zich op lange termijn nog een ander probleem voordoen. Aangezien de omvang van software exponentieel toeneemt en elke aanpassing in de code minstens één nieuwe test vereist, zal het aantal tests nog sneller stijgen. Dit leidt andermaal tot een schaalbaarheidsprobleem. Deze masterproef tracht dit probleem op te lossen door het testpakket te optimaliseren en de schaalbaarheid te verhogen.

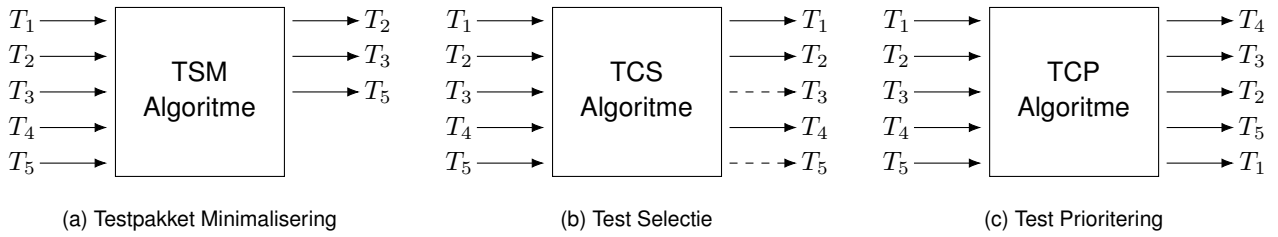
II. TECHNIEKEN

Deze masterproef presenteert drie technieken om dit schaalbaarheidsprobleem op te lossen.

A. Testpakket Minimalisering (TSM)

De eerste techniek is Testpakket Minimalisering (TSM) [4]. TSM probeert de grootte van het testpakket te verkleinen door redundante tests permanent te verwijderen, volgens volgende definitie:

¹De verzameling van alle tests in de applicatie.



Figuur 2: Overzicht van de technieken.

Gegeven:

- $T = \{t_1, \dots, t_n\}$ een testpakket bestaande uit tests t_j .
- $R = \{r_1, \dots, r_m\}$ een verzameling vereisten die voldaan moeten zijn om te stellen dat een applicatie grondig getest is.
- $\{T_1, \dots, T_m\}$ deelverzamelingen van tests uit T . Elke deelverzameling T_i wordt geassocieerd met een vereiste r_i , zodanig dat eender welke test $t_j \in T_i$ kan worden uitgevoerd om te voldoen aan vereiste r_i .

Het probleem van TSM is vervolgens gedefinieerd als het vinden van een minimale deelverzameling T' van tests $t_j \in T$, zodanig dat aan elke vereiste $r_i \in R$ voldaan is.

B. Test Selectie (TCS)

In plaats van tests permanent te verwijderen, is het ook mogelijk om de veranderingen aan de code te analyseren om zo te bepalen welke tests zeker uitgevoerd moeten worden. Analooog kunnen andere tests mogelijk worden uitgesloten, omdat ze (waarschijnlijk) niet zullen falen [4].

Gegeven:

- T het testpakket.
- P de vorige versie van de code.
- P' the huidige (aangepaste) versie van de code.

TCS vindt een deelverzameling $T' \subseteq T$ die gebruikt kan worden om P' adequaat te testen.

C. Test Prioritering (TCP)

TSM en TCS voeren zo weinig mogelijk tests uit om de omvang van het testpakket te verkleinen. Soms kan het echter gewenst zijn om toch elke test uit te voeren, bijvoorbeeld bij kritische software voor medische doeleinden. In dit geval kan het testpakket nog steeds geoptimaliseerd worden, door de uitvoeringsvolgorde aan te passen. Test Prioritering (TCP) [4] rangschikt de tests zodanig dat een vooropgesteld doel zo snel mogelijk bereikt wordt. In deze masterproef zal het doel steeds zijn om zo snel mogelijk een falende test te detecteren.

Given:

- T het testpakket.
- PT de verzameling van alle permutaties van T .
- $f : PT \mapsto \mathbb{R}$ een functie die een deelverzameling afbeeldt op een reëel getal. Deze functie wordt gebruikt om permutaties met elkaar te vergelijken.

TCP bepaalt de optimale permutatie $T' \in PT$ zodanig dat $\forall T'' \in PT : f(T') \geq f(T'') \Rightarrow (T' \neq T'')$.

III. ALGORITMEN

Deze masterproef focust op de TCP techniek, aangezien deze techniek elke test uitvoert en daardoor geen risico loopt om falende tests over te slaan. Om de optimale uitvoeringsvolgorde te bepalen worden drie algoritmen voorgesteld.

De algoritmen kunnen gebruik maken van drie gegevensreeksen:

- **Getroffen tests:** Door gebruik te maken van eerdere codebedekkingsresultaten² en de lijst met aanpassingen aan de code kan het framework bepalen welke tests mogelijks getroffen zijn door deze aanpassingen. Bijgevolg kan aan deze tests een hogere prioriteit toegewezen worden.
- **Historische uitvoeringsdata:** Vervolgens kan een algoritme gebruikmaken van historische uitvoeringsdata. Stel dat een test tijdens de vorige uitvoering gefaald is, dan bestaat de kans dat deze nu opnieuw zal falen. Hoe dan ook moet deze test een hoge prioriteit krijgen om te controleren of het onderliggend probleem inmiddels is opgelost.
- **Uitvoeringstijden:** Ten slotte kan de gemiddelde uitvoeringstijd van een test gebruikt worden. Wanneer twee tests evenveel kans maken om te falen, moet de test met de laagste uitvoeringstijd de voorkeur genieten om sneller een potentieel falende test te detecteren.

A. Greitig algoritme

Het eerste algoritme is een greitige heuristiek die oorspronkelijk ontworpen is om het *set-covering probleem* op te lossen [5]. De heuristiek start met een lege

²de *code coverage* geeft aan welke instructies in de code worden uitgevoerd bij het uitvoeren van een test

verzameling tests en de verzameling C van alle code-lijnen. Vervolgens selecteert het algoritme iteratief de test die het meeste nog onbedekte codelijnen zal bijdragen. Na elke geselecteerde test wordt C bijgewerkt. Het algoritme herhaalt deze stappen totdat elke test geselecteerd is, of tot dat C leeg is. Dit algoritme kan aangepast worden naar TCP door de selectievolgorde bij te houden en die te gebruiken als uitvoeringsvolgorde.

B. HGS-algoritme

Het tweede algoritme is bedacht door Harrold, Gupta en Soffa [6]. In tegenstelling tot het gretige algoritme, gebruikt dit algoritme een andere invalshoek. Het algoritme begint door de lijst met codelijnen te sorteren op stijgend aantal tests waardoor ze bedekt zijn. De reden voor deze sortering is dat sommige tests hoe dan ook moeten uitgevoerd worden, aangezien ze mogelijks de enige tests zijn die een bepaalde codeliijn bedekken. Deze tests kunnen echter ook codelijnen bedekken die door andere tests worden bedekt, waardoor deze redundant worden. Het algoritme overloopt alle codelijnen in deze volgorde en selecteert steeds een bijbehorende test. Na elke geselecteerde test wordt de lijst met codelijnen bijgewerkt om lijnen en wordt elke lijn die bedekt is door de geselecteerde test verwijderd. Dit proces herhaalt zich tot elke codeliijn bedekt is.

C. ROCKET-algoritme

Ten slotte beschouwt deze masterproef het ROCKET-algoritme [7]. Dit algoritme ordent tests door aan elke test een score toe te kennen, op basis van historische uitvoeringsgegevens. Daarna wordt de cumulatieve score CS_t voor elke test t berekent en wordt de objectiefunctie g gedefinieerd. In deze functie stelt E_t de uitvoeringstijd van de test voor:

$$g = (\text{maximaliseer}(CS_t), \text{minimaliseer}(E_t))$$

Vervolgens optimaliseert het algoritme deze functie om de ideale uitvoeringsvolgorde S te bepalen, als volgt:

$$(\forall i \in 1 \dots n)(g(S_i) \geq g(S_{i+1}))$$

IV. FRAMEWORK

Deze masterproef stelt het VeloClty framework voor. Dit programmertaal-onafhankelijk framework laat toe om Test Prioritering te gebruiken met bestaande softwareprojecten. De architectuur bestaat uit drie hoofdcomponenten, een meta predictor en een nieuw prioriteringsalgoritme.

A. Agent

De eerste component is de agent. Deze component wordt geïntegreerd in het testframework van de applicatie en is verantwoordelijk voor het uitvoeren van de tests in de optimale volgorde. Deze volgorde wordt opgevraagd aan de volgende component, de controller.

B. Controller

De controller is een server die twee taken uitvoert. De eerste taak is het afhandelen van verzoeken door de agent en deze door te sturen naar de *voorspeller*. Daarnaast ontvangt de controller ook feedback van de agent na elk uitgevoerd testpakket. Deze informatie wordt gebruikt om de meta predictor bij te werken, hierover later meer.

C. Voorspeller

Het laatste onderdeel van de architectuur is de voorspeller. Deze component inspecteert de aanpassingen aan de code om de optimale uitvoeringsvolgorde te bepalen. Deze volgorde wordt bepaald met behulp van tien ingebouwde voorspellingsalgoritmen. Deze algoritmen zijn varianten van de drie eerder besproken algoritmen, aangevuld met het Alfa-algoritme (zie verder).

D. Meta predictor

Aangezien de voorspeller tien algoritmen bevat, is er een extra onderdeel nodig dat bepaalt welk van deze tien voorspelde volgordes de uiteindelijke uitvoeringsvolgorde moet worden. De meta predictor is een tabel die een score toekent aan elk voorspellingsalgoritme. De volgorde van het algoritme met de hoogste score wordt gekozen als finale uitvoeringsvolgorde. Nadat de tests zijn uitgevoerd evalueert de controller de volgorde van de andere algoritmen en worden de scores bijgewerkt.

E. Alfa algoritme

Naast de Gretig, HGS- en ROCKET-algoritmen bevat dit framework ook een eigen algoritme. Dit algoritme start met het analyseren van de tests om de verzameling ATS van getroffen tests te bepalen. Binnen ATS , beschouwt het algoritme elke test die ten minste één keer gefaald is in de laatste drie uitvoeringen. Deze tests worden geselecteerd in stijgende uitvoeringstijd. Daarna worden de overblijvende tests in ATS geselecteerd, eveneens volgens stijgende uitvoeringstijd. Na deze twee stappen worden de resterende tests geselecteerd volgens het eerder besproken gretig algoritme.

V. EVALUATIE

Het framework is geïnstalleerd in twee bestaande applicaties. Het eerste project is het Dodona project³ van de Universiteit Gent. Dit project is gebruikt om de performantie van de voorspellingsalgoritmen te evalueren. Aangezien de agent in deze masterproef enkel compatibel is met Java applicaties en Dodona gebouwd is in Ruby-on-Rails, is een tweede testapplicatie gebruikt om de installatie te verifiëren. Hiervoor werd het Stratego⁴ project gebruikt. Aansluitend beantwoordt deze masterproef drie onderzoeksvragen, met als doel waardevolle inzichten te vergaren over testpakketten.

A. Performantie

Tabel 1 vergelijkt de vier besproken algoritmen op twee vlakken. Deze vlakken zijn respectievelijk het aantal uitgevoerde tests tot de eerste gefaalde test en de bijbehorende uitvoeringstijd van het partiële testpakket. Deze resultaten tonen aan dat het Alfa algoritme bijna 30 keer minder tests uitvoert en dat de eerste falende test bijna onmiddellijk gedetecteerd wordt. Het Greedy en HGS-algoritme halveren het aantal uitgevoerde tests en zorgen tegelijk voor een sterke reductie van de uitvoeringstijd. De performantie van het ROCKET-algoritme is opmerkelijk. Dit algoritme voert veel meer tests uit dan de mediaan originele, niet-geprioriteerde volgorde, maar detecteert een falende test vier keer sneller.

Algoritme	Mediaan (tests)	Mediaan (tijd)
Origineel	78	123 s
Alfa	3	1 s
Greedy	33	12 s
HGS	10	6 s
ROCKET	170	32

Tabel 1: Performantie op het Dodona project.

B. Onderzoeksvragen

Deze masterproef beantwoordt drie onderzoeksvragen door gebruik te maken van gegevens van Travis CI. Deze gegevens zijn verzameld en gepubliceerd door het TravisTorrent project [8] (3 702 595 uitvoeringen) en door Durieux et al. [9] (35 793 144 uitvoeringen).

OV1: Kans op falen. De eerste onderzoeksvraag beschouwt de kans dat de uitvoering van een testpakket zal falen. Volgens beide bronnen is deze kans 15 %.

OV2: Gemiddelde uitvoeringstijd. Na het inspecteren van de TravisTorrent gegevens werd duidelijk dat

³<https://dodona.ugent.be/>

⁴Java Spring applicatie ontwikkeld voor het vak Software Engineering 2.

de tijdsinformatie onvolledig en inaccuraat was, waardoor deze gegevens niet konden gebruikt worden om een betrouwbaar antwoord op deze vraag te bieden. De andere bron geeft aan dat Travis CI hoofdzakelijk gebruikt wordt voor kleinere projecten, met een gemiddelde uitvoeringstijd van 385 s per testpakket en een maximum van 26 h.

OV3: Opeenvolgend falen. De laatste onderzoeksvraag betreft de kans dat de tests van een project meer dan twee keer na elkaar falen. Hiervoor kan enkel de TravisTorrent bron worden gebruikt aangezien deze bron een koppeling bevat tussen alle uitvoeringen van hetzelfde project. Volgens deze bron is de kans op opeenvolgend falen 51.76 %.

VI. CONCLUSIE EN AANVULLEND WERK

In deze masterproef is aangetoond dat het testpakket van softwareprojecten geoptimaliseerd kan worden, in het bijzonder met behulp van Test Prioritering. Het voorgestelde framework is succesvol geïmplementeerd in twee projecten en de resultaten zijn veelbelovend. Er zijn echter nog een aantal limitaties, alsook ruimte voor verbetering.

Agent. Voor de implementatie van de agent in deze masterproef werd gekozen voor Java, meer specifiek het JUnit testframework. Dit heeft als nadeel dat tests niet in parallel kunnen worden uitgevoerd. Om dit mogelijk te maken is er, naast de technische moeilijkheden, nood aan een coördinatiemechanisme om tests over meerdere processorthreads te plannen.

Voorspeller. Momenteel werkt elk voorspellingsalgoritme volledig op zichzelf. Mogelijks kan er verborgen potentieel verschuilen in de combinatie van verschillende algoritmen, bijvoorbeeld door gewichten toe te kennen aan elk algoritme en op basis daarvan de voorspellingen te mengen.

Meta predictor. De eenvoudige meta predictor die in deze masterproef wordt gebruikt kan aangepast worden naar bijvoorbeeld een saturerende teller, of naar een Machine Learning algoritme. Analooq kan Machine Learning eventueel worden gebruikt als voorspellingsalgoritme.

REFERENTIES

- [1] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas, "Manifesto for agile software development," 2001.
- [2] Standish Group et al., "Chaos report 2015," *The Standish Group International*, 2015.
- [3] John Ferguson Smart, *Jenkins: The Definitive Guide*, O'Reilly, Beijing, 2011.

- [4] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Softw. Test. Verif. Reliab.*, vol. 22, no. 2, pp. 67–120, Mar. 2012.
- [5] Raphael Noemmer and Roman Haas, *An Evaluation of Test Suite Minimization Techniques*, pp. 51–66, 12 2019.
- [6] M. Jean Harrold, Rajiv Gupta, and Mary Lou Soffa, "A methodology for controlling the size of a test suite," *ACM Trans. Softw. Eng. Methodol.*, vol. 2, no. 3, pp. 270–285, July 1993.
- [7] 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.
- [8] Moritz Beller, Georgios Gousios, and Andy Zaidman, "Travis-torrent: Synthesizing travis ci and github for full-stack research on continuous integration," in *Proceedings of the 14th working conference on mining software repositories*, 2017.
- [9] Thomas Durieux, Rui Abreu, Martin Monperrus, Tegawendé F. Bissyandé, and Luís Cruz, "An analysis of 35+ million jobs of travis ci," 2019.

Lay summary

Software

Computers, smartphones, cars, or even much simpler devices like alarm clocks and microwaves, every digital device that exists today consists of two distinct parts. The first, physical part is the hardware, which is the combination of mechanical bits and electrical wiring that enable a device to interact with the real world. The type of interaction can range from either very primitive to extremely complex, such as emitting an LED-light, producing a sound, or launching a rocket. The second part is the software, which is installed on the hardware of the device. Software is developed by software engineers using programming languages and instructs the hardware on what to do, and when.

Testing

Deciding on what is the "best" approach towards the development of software is an entire science on its own with two main conceptions. The traditional approach starts with a thinking phase, followed by a programming phase and finalised by a testing phase. In the first phase, the developers create a detailed design document that describes the required functionality of the final application. Next, the developers write computer code that implements the desired functionality. When this process is completed, the quality assurance team thoroughly tests the application. This testing phase exists in hardware as well. Consider, for example, crash tests conducted by car manufacturers. The purpose of these tests is to detect potential issues and anomalies (bugs) in the application before its end-users do. This phase is critical because bugs can result in financial loss or incur other disastrous effects, such as the explosion of two space rockets in the previous decade, mere seconds after ignition.

Continuous Integration

The urge to confine financial losses is even more prominent today, in the wake of the world economic crisis and the more recent COVID-19 induced crisis. While the aforementioned traditional approach works well for small projects, it suffers from severe scalability issues when the size of the application increases at today's pace, since the testing phase consumes valuable time, and time equals money. As a result, software

developers have shifted towards an Agile development approach. This approach encourages software developers not to release the entire application at once, but release an initial version with a reduced functionality set as soon as possible and add extra features iteratively. Additionally, the developers must include automated software tests and execute these every time they make a change, to reduce the probability of introducing bugs. Because this is a tedious task, additional software has been created that automatically executes these tests after every change, under the name of Continuous Integration (CI), as illustrated in Figure 1.

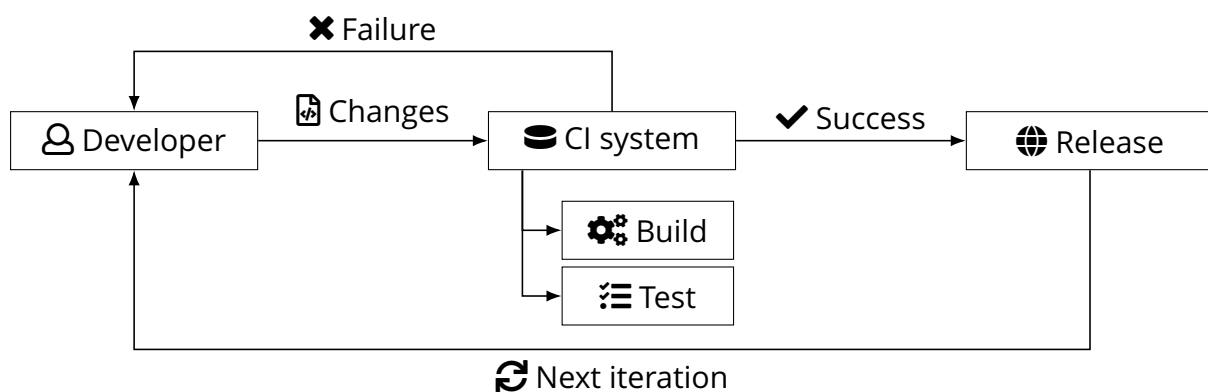


Figure 1: Continuous Integration (simplified).

Scalability

Nevertheless, Continuous Integration is not a silver bullet. In the initial stage of the project, the number of test cases will be rather small, therefore providing fast feedback to the developers in case of failure. However, as time progresses and the application grows, more test cases will be added that all need to be executed after every change. Eventually, this will consume a significant amount of time as well, thereby nullifying these benefits.

Solution

This thesis focuses on resolving this problem by introducing three techniques. The first two techniques are *Test Suite Minimisation* and *Test Case Selection*. These techniques attempt to predict which test cases are likely to fail, and as such, only execute those test cases with a high probability of failing. The third technique is *Test Case Prioritisation (TCP)*. As the name suggests, this technique will execute every test case in a specific sequence. The order of this sequence is determined by the predicted chance that the test case will fail, executing the most likely failing test cases as soon as possible.

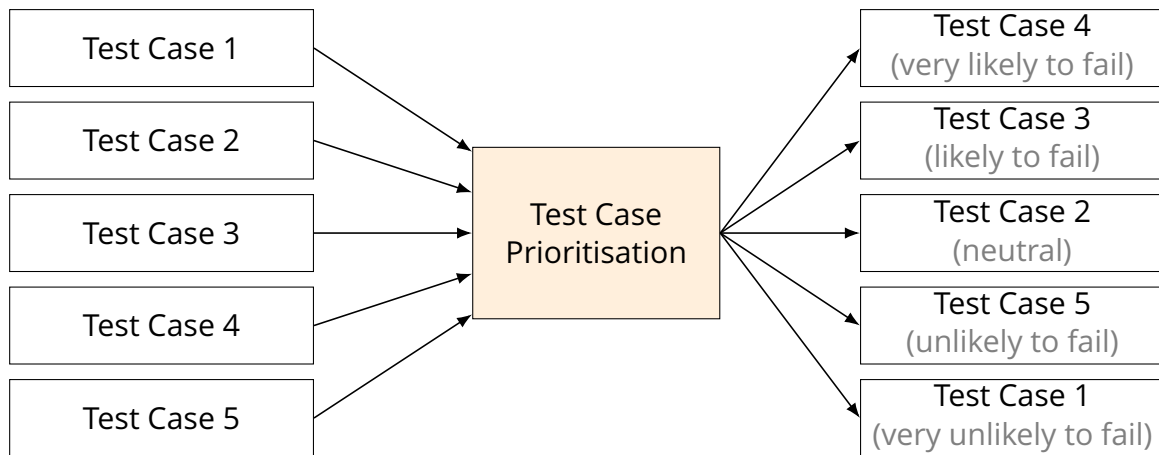


Figure 2: Test Case Prioritisation.

This thesis concentrates on TCP (Figure 2) since this technique ensures that every failing test case will eventually be executed. The other two techniques cannot guarantee this because a failing test case might accidentally be omitted.

Prediction

In order to estimate which test cases might fail, the TCP implementation in this thesis consists of ten prediction algorithms, referred to as predictors. Every predictor uses the same input data but with a different interpretation, which is out-of-scope for this summary. This input data is threefold:

1. **Affected test cases:** The predictors contain a mapping that links every test case to the corresponding tested lines of code in the application. If the developer modifies a line of code, every *affected* test case is considered a potential failure.
2. **Historical data:** Next, the predictors can examine whether or not a test case has recently failed. Research has indicated that test cases tend to fail consecutively.
3. **Duration data:** Finally, the predictors can use the average duration of a test case as a tie-breaker. If two test cases are equally likely to fail, the test case with the lowest duration should be preferred to speed up the execution.

Results

The benefit of applying TCP on two existing applications has been analysed. The results are promising, the implemented optimisation framework executes, on average, only between 3% – 5% of the test cases. When examining the time it takes to detect a failing test case, the results indicate a reduction of more than 30 – 50 times compared to the original, unprioritised execution.

Contents

Summary	iv
Extended abstract	vi
1 Introduction	2
2 Software Engineering	4
2.1 Software Development Life Cycle	4
2.1.1 Taxonomy of test cases	6
2.2 Test Suite Assessment	9
2.2.1 Coverage	9
2.2.2 Mutation testing	12
2.3 Agile Software Development	14
2.3.1 Agile Manifesto	14
2.3.2 The need for Agile	14
2.3.3 Continuous Integration	15
3 Related work	19
3.1 Classification of approaches	20
3.1.1 Test Suite Minimisation	20
3.1.2 Test Case Selection	21
3.1.3 Test Case Prioritisation	22
3.2 Algorithms	23
3.2.1 Greedy algorithm	24
3.2.2 HGS	25
3.2.3 ROCKET algorithm	27
3.3 Adoption in testing frameworks	29
3.3.1 Gradle and JUnit	29
3.3.2 Maven Surefire	30
3.3.3 OpenClover	30
4 Proposed framework: VeloCity	31
4.1 Design goals	31
4.2 Architecture	32
4.2.1 Agent	32
4.2.2 Controller	32
4.2.3 Predictor and Metrics	32

4.3	Pipeline	34
4.3.1	Initialisation	34
4.3.2	Prediction	35
4.3.3	Test case execution	36
4.3.4	Post-processing and analysis	37
4.4	Alpha algorithm	38
4.5	Analysis	41
5	Evaluation	43
5.1	Test subjects	43
5.1.1	Dodona	43
5.1.2	Stratego	43
5.2	Research questions	44
5.3	Data collection	44
5.3.1	Travis CI build data	44
5.3.2	Dodona data	46
5.3.3	Stratego data	46
5.4	Results	47
5.4.1	RQ1: Probability of failure	47
5.4.2	RQ2: Average duration of a test run	47
5.4.3	RQ3: Consecutive failure probability	48
5.4.4	RQ4: Applying Test Case Prioritisation to Dodona	49
5.4.5	RQ5: Integrate VeloCity with Stratego	52
6	Conclusion	55
6.1	Future work	56
6.1.1	Java agent	56
6.1.2	Predictions	56
6.1.3	Meta predictor	57
6.1.4	Final enhancements	58
	List of Figures	62
	List of Tables	63
	List of Examples	64
A	TravisTorrent queries	67

Terms

Acronyms

CI Continuous Integration. 2, 15, 19

IEEE The Institute of Electrical and Electronics Engineers. 4

SDLC Software Development Life Cycle. 4

TCP Test Case Prioritisation. 2, 19, 22, 23

TCS Test Case Selection. 2, 19, 21, 22

TDD Test-driven development. 14

TSM Test Suite Minimisation. 2, 19, 20, 22, 23

VCS Version Control System. 2

Glossary

black-box test a test case that was constructed without any knowledge of the function(s) under test. 7

MapReduce a programming paradigm that allows large amounts of data to be processed in a distributed manner. 45

REST Representational State Transfer is an architectural design pattern used by modern web applications. This design pattern encourages standardised communication using existing HTTP methods. 32

test suite the collection of all test cases in an application. 6

white-box test a test case that was constructed after fully inspecting the function(s) under test. 6

Chapter 1

Introduction

Given the complexity and rapid pace at which software is being built today, it is inevitable that sooner or later, 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, we require an adequate *testing infrastructure*.

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 process 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. Every addition or modification to the source code must be followed by at least one test case 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.

We can take three approaches to resolve this issue and reduce 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.

The structure of this thesis is as follows. The next chapter will introduce essential concepts used in modern software engineering. Chapter 3 will elaborate more on the three mentioned 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 the “Application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software; that is, the application of engineering to software” [21, p. 421]. The word “systematic” in this definition emphasises the need for a structured process, depicting guidelines and models that describe how we should develop software in the most efficient way possible. Such a process does exist under the name of the Software Development Life Cycle (SDLC) [21, p. 420]. If a developer prefers not to abide by any model and act as they deem correct without following any guidelines, we employ the term *Cowboy coding* [23, p. 34].

2.1 Software Development Life Cycle

An implementation of the SDLC typically consists of several phases and a model that describes the transition from every phase to another. Depending on the nature of the software, we can either omit some or add more phases. The five phases below were compiled from multiple sources [15, 20] and describe a generic approach to which most software projects adhere.

1. **Requirements phase:** In the first phase of the development process, the developers acquaint themselves with the project and compile a list of the desired functionalities [20]. Subsequently, the developers can decide on the financial details, the required hardware specifications as well as which external software libraries 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 construct an architectural design of the application. This design consists of multiple documents, such as user stories and UML-diagrams. A user story describes which actions can be performed by which users, whereas a UML-diagram specifies the technical interaction between the individual components.
3. **Implementation phase:** In the third phase, the developers will write code according to the specifications defined in the architectural designs.

4. **Testing phase:** The fourth phase is the most critical. This phase will require the developers and quality assurance managers to test the implementation of the application thoroughly. The goal of this phase is to identify potential bugs before the application is made available to other users.
5. **Operational phase:** The final phase marks the completion of the project, after which the developers can integrate it into the existing business environment of their customer.

After we have identified the phases, we must define the transition from one phase into another phase using a model. Multiple models exist in the literature [15], with each model having its advantages and disadvantages. This thesis will consider the traditional model, which is still widely used as of today. The base of this model is the Waterfall model by Benington [6]. Similar to a real waterfall, this model executes every phase in cascading order. However, this imposes several restrictions. The most prevalent issue is the inability to revise a design decision when performing the actual implementation. To mitigate this problem, Royce has proposed an improved version of the Waterfall model [32], which does allow a phase to transition back to any preceding phase. Figure 2.1 illustrates this updated model.

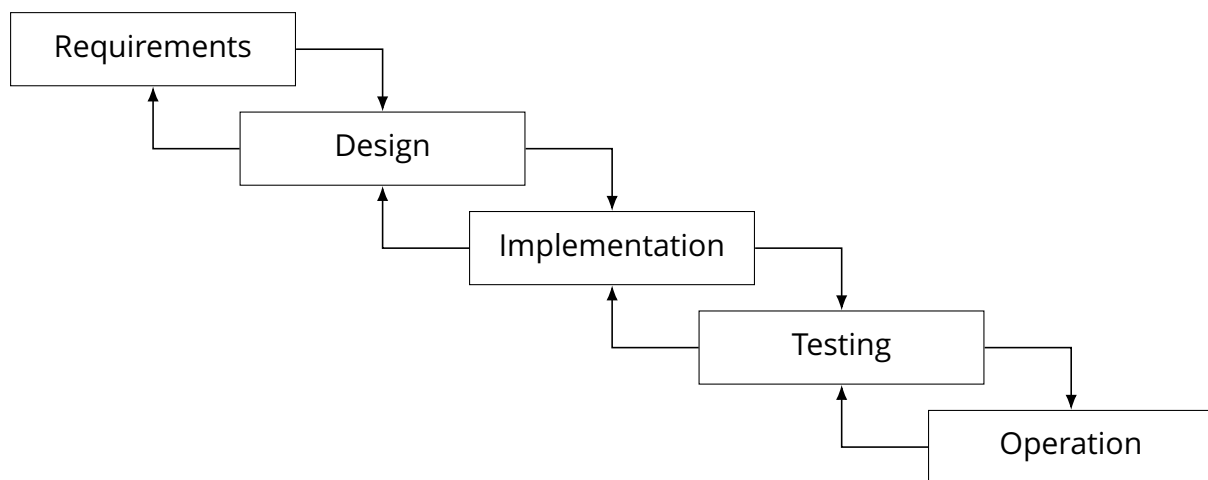


Figure 2.1: Improved Waterfall model by Royce

The focus of this thesis will be on the implementation and testing phases, as these are the most time-consuming phases of the entire process. The modification that Royce has applied to the Waterfall model is particularly useful when applied to these two phases in the context of *software regressions* [29]. We employ the term “regression” when a feature that was once working as intended is suddenly malfunctioning. The culprit of this problem can be a change in the code, but this behaviour can also have an external cause, such as a change in the system clock due to daylight saving time. Sometimes, a regression can even be the result of a change to another, seemingly unrelated part of the application code [19].

2.1.1 Taxonomy of test cases

Software regressions and other functional bugs can ultimately incur disastrous effects, such as severe financial loss or permanent 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 [24]. In order to reduce the risk of bugs, we should be able to detect malfunctioning components as soon as possible to warden the application against potential failures before they occur. Consequently, we must consider the testing phase as the most critical phase of the entire development process and therefore include sufficient test cases in the application. The collection of all the test cases in an application is referred to as the *test suite*. We can distinguish many different types of test cases. This thesis will consider three categories in particular.

Unit tests

This is the most basic test type. The purpose of a unit test is to verify the behaviour of an individual component [36]. As a result, the scope of a unit test is limited to a small and isolated piece of code, e.g. one function. Implementation-wise, a unit test is typically an example of a white-box test [19, p. 12]. The term white-box indicates that the creator of the test case can manually inspect the code before constructing the test. As such, they can identify necessary edge values or corner cases. Common examples of these edge values include zero, negative numbers, empty arrays or array boundaries that might cause an overflow. Once the developer has identified the edge cases, they can construct the unit test by repeatedly calling the function under test, each time with a different (edge) argument value, and afterwards verifying its behaviour and result. These verifications are referred to as *assertions*. Example 2.1 contains a unit test written in Java using the popular JUnit test framework.

```
1 public class ExampleUnitTest {  
2     static int square(int base) { return base * base; }  
3  
4     @Test  
5     public void testSquare() {  
6         Assert.assertEquals(25, square(5));  
7         Assert.assertEquals(4, square(-2));  
8     }  
9 }
```

Example 2.1: Java unit test in JUnit.

Integration tests

The second category involves a more advanced type of tests. An integration test validates the interaction between two or more individual components [36]. Ideally, accompanying unit tests should exist that test these components as well. As opposed to the previous unit tests, a developer will usually implement an integration test as a black-box test [19, p. 6]. A black-box test differs from the earlier white-box tests in the fact that the implementation details of the code under test are irrelevant for the construction of the test. Since a black-box test does not require any details about the code, we can, in fact, construct the integration tests before we implement the actual feature itself. A typical example of an integration test is the communication between the front-end and the back-end side of a web application. Another example is illustrated in Example 2.2.

```
1 public class ExampleIntegrationTest {
2     @Test
3     public void testOrderPizza() {
4         // Authenticate a test user.
5         Session session = UserSystem.login("JohnDoe", "password");
6         session.wallet.balance = 1000.0;
7         Assert.assertNotNull(session);
8
9         // Find an item to order.
10        Pizza pizza = new Pizza(Flavour.PEPPERONI);
11        Assert.assertNotNull(pizza);
12
13        // Create an order.
14        Order order = OrderSystem.createOrder(session, pizza);
15        Assert.assertNotNull(order);
16
17        // Checkout.
18        double oldBalance = session.wallet.balance;
19        order.checkout(session.wallet);
20        double newBalance = session.wallet.balance;
21        Assert.assertEquals(oldBalance - pizza.price, newBalance);
22    }
23 }
```

Example 2.2: Java integration test in JUnit.

Regression tests

After a developer has detected a regression in the application, they will add a regression test [21, p. 372] to the test suite. This regression test must replicate the exact conditions and sequence of actions that have triggered the failure. The goal of this test is to prevent similar failures to occur in the future if the same conditions would reapply. An example is provided in Example 2.3.

```
1 public class ExampleRegressionTest {
2     // Regression #439: A user cannot remove their comments after
3     they have changed their first name.
4     @Test
5     public void testRegression439() {
6         // Authenticate a test user.
7         Session session = UserSystem.login("johndoe", "password");
8         Assert.assertNotNull(session);
9
10        // Create a comment.
11        String content = "This is a comment by John Doe.";
12        Comment comment = CommentSystem.create(content, session);
13        Assert.assertNotNull(comment);
14
15        // Change the first name of the user.
16        Assert.assertEquals("Bert", session.user.firstName);
17        session.user.setFirstName("Matthew");
18        Assert.assertEquals("Matthew", session.user.firstName);
19        Assert.assertEquals("Matthew", comment.user.firstName);
20
21        // Try to remove the comment.
22        CommentSystem.remove(comment);
23        Assert.assertTrue(comment.removed);
24    }
25 }
```

Example 2.3: Java regression test in JUnit.

2.2 Test Suite Assessment

2.2.1 Coverage

The most frequently used metric to measure the quantity and thoroughness of a test suite is the *code coverage* or *test coverage* [21, p. 467]. The test coverage indicates which fraction of the application code is hit by at least one test case in the test suite. Internally, a coverage framework calculates the coverage ratio by augmenting every application statement using binary instrumentation. A hook is inserted before and after every statement to detect which statements are executed by the test cases. Many different criteria exist to interpret these results and thus to express the fraction of covered code [28]. The two most commonly used criteria are *statement coverage* and *branch coverage*.

Statement coverage Statement coverage expresses the fraction of statements that are executed by any test case in the test suite, over the total amount of statements in the code [19]. Similarly, we can calculate the *line coverage* as the fraction of covered code lines. Since one statement can span multiple lines and one line may also contain more than one statement, both of these criteria are intrinsically related. Statement coverage is heavily criticised in literature [28, p. 37] since it is possible to achieve a statement coverage percentage of 100 % on code of which we can prove it is malfunctioning. Consider example 2.4. If a test case would call the `example`-function twice with the arguments $\{a = 1, b = 2\}$ and $\{a = 5, b = 0\}$, then both test cases will pass and every statement will be covered, resulting in a statement coverage of 100 %. However, suppose we would call the function with arguments $\{a = 0, b = 0\}$. The first argument matches the first condition of the branch but will trigger a division-by-zero error, even though the previous combination of arguments reported a complete statement coverage. This simple example was already sufficient to illustrate that statement coverage is not trustworthy. However, statement coverage may still prove useful for other purposes, such as detecting unreachable code which we may safely remove.

```
1 int example(int a, int b) {  
2     if (a == 0 || b != 0) {  
3         return a / b;  
4     }  
5  
6     return 0;  
7 }
```




Example 2.4: Irrelevant statement coverage in C.

Branch coverage requires that the test cases traverse every branch of a conditional statement at least once [28, p. 37]. For an if-statement, we require two test cases, one for every possible outcome of the condition (`true` or `false`). For a loop-statement, we require at least two test cases as well. One test case should never execute the loop, and the other test case should execute every iteration. Optionally, we can add additional test cases for specific iterations. Observe that, while this criterion is stronger than statement coverage, it will still not detect the bug in Example 2.4. In order to mitigate this, we can use *multiple-condition coverage* [28, p. 40]. This criterion requires that for every conditional expression, every possible combination of subexpressions is evaluated at least once. If we apply this requirement to Example 2.4, the if-statement will only be covered if we test the following four cases.

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

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 every line of code, every statement or every branch must be explicitly covered [9]. Some parts of the code might be irrelevant or untestable, such as wrapper or delegation methods that only call a library function. All major programming languages have frameworks and libraries that enable the collection of coverage information during test execution, and each of these frameworks allows the exclusion parts of the code from the final coverage calculation. As of today, the most popular options are JaCoCo¹ for Java, coverage.py² for Python and simplecov³ for Ruby. These frameworks report in-depth statistics on the covered code and indicate which parts require more extensive testing, as illustrated in Figure 2.2.

io.github.thepieterdc.http.impl

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

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

¹<https://www.jacoco.org/jacoco/>

²<https://github.com/nedbat/coveragepy>

³<https://github.com/colszowka/simplecov>

Coverage report: 75%

Module ↓	statements	missing	excluded	coverage
awesome/__init__.py	4	1	0	75%
<pre> 1 def smile(): 2 return ":)" 3 4 def frown(): 5 return ":(" </pre>				
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 2.2: Statistics from Code coverage frameworks.

2.2.2 Mutation testing

The previous section has explained how we can identify which parts of the code require additional test cases. However, we cannot yet measure the quality and resilience of the test suite nor its ability to detect future failures. To accomplish this, we can employ *mutation testing*. This technique creates several *mutants* of the application under test. A mutant is a syntactically different instance of the source code. We can create a mutant by applying one or more *mutation operators* to the original source code. These mutation operators attempt to simulate typical mistakes that developers tend to make, such as the introduction of off-by-one errors, removal of statements and the replacement of logical connectors [31]. 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 [22].

Creating and evaluating the mutant versions of the code is a computationally expensive process which typically requires human intervention. As a result, very few software developers have managed to employ this technique in practice. Figure 2.3 illustrates the process of applying mutation testing. The first step is to consider the original program P and a set of test cases TS , to which we apply mutation operators to construct a broad set of mutants P' . Next, we evaluate every test case $t \in TS$ on the original program P to determine the correct behaviour. Note that this step assumes that if the original source code passes the test cases, it is correct.

This assumption will only be valid if the test suite contains sufficient thorough unit tests. If at least one of these test cases fails, we have found a bug which we must first resolve before continuing with the mutation analysis. When P successfully passes every test case, we evaluate every test case for each of the mutants. A mutant p' is said to be “killed” if its outcome is different from P for at least one test case. Otherwise, we refer to the mutant as “surviving”. After we have finished executing the test cases on every mutant, we analyse the set of surviving mutants. Every mutant that managed to survive implies a change in the source code that did not trigger any failure in the test cases. As a result, we need to introduce subsequent test cases until we have killed every mutant. However, it is also possible that the surviving mutants are functionally equivalent to P and are, therefore, correct. Since the detection of program equivalence is an impossible problem, we need to verify this manually [22, 31]. Finally, note that although we can use mutation testing to estimate the adequacy of the test suite, it is not flawless, as several mutation operators can cancel each other out [30].

After every mutant has either been killed or marked equivalent to the original problem, we can calculate the *mutation score* of the test suite using Equation (2.1). In an adequate test suite, this score is equal to 1, which indicates that the test suite was able to detect every mutant instantly.

$$\text{Mutant Score} = \frac{\text{killed mutants}}{\text{non-equivalent mutants}} \quad (2.1)$$

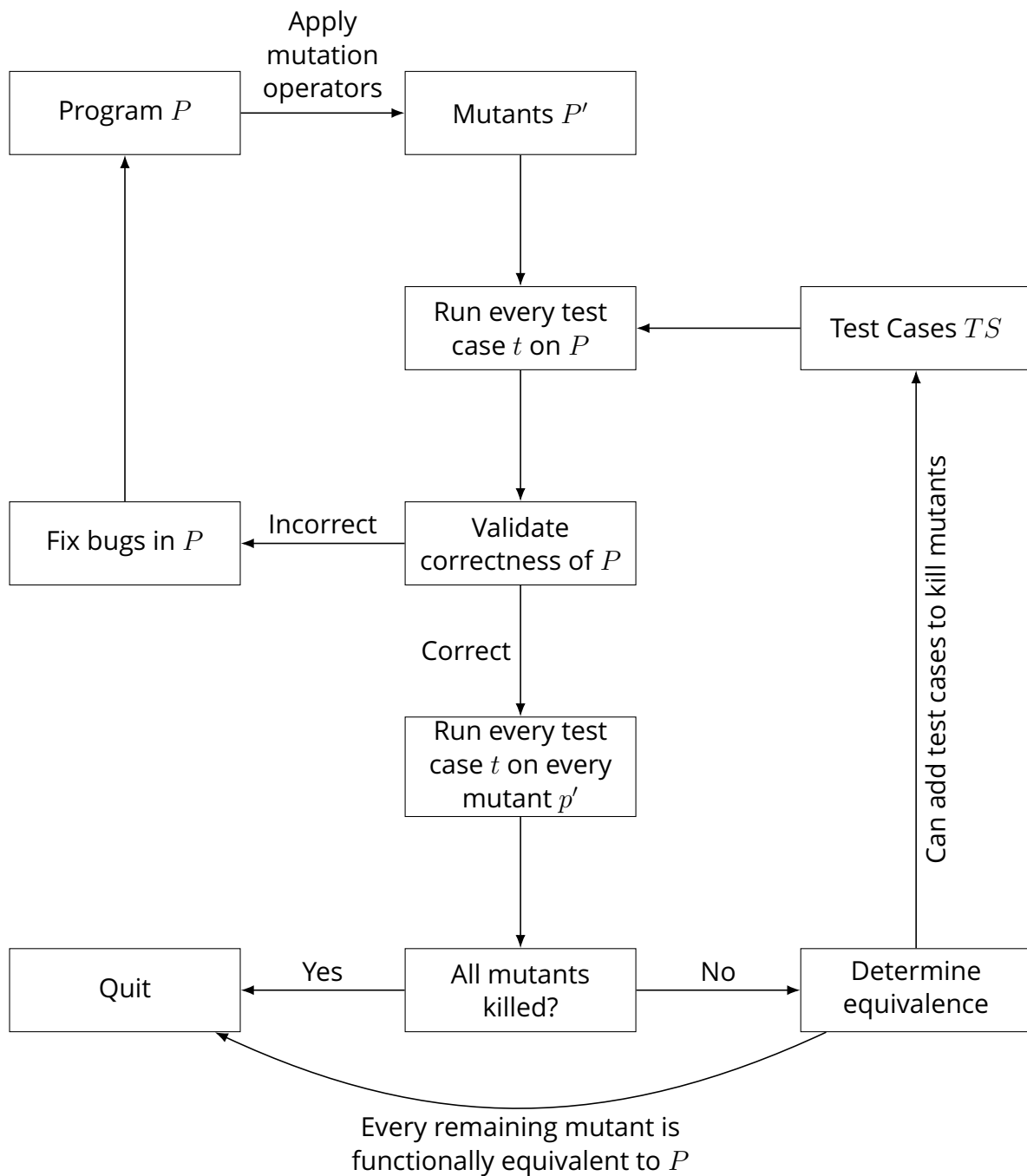


Figure 2.3: Process of Mutation Testing (based on [31]).

2.3 Agile Software Development

2.3.1 Agile Manifesto

Since the late 1990s, developers have tried to reduce the time occupied by the implementation and testing phases. As a result, several software pioneers have proposed new implementations of the SDLC, which were later collectively referred to as the *Agile development methodologies*. This term was coined during a meeting of seventeen prominent software developers, in which they have defined the following four fundamental values of Agile development in the *Agile Manifesto* [4].

1. *Individuals and interactions* over processes and tools.
2. *Working software* over comprehensive documentation.
3. *Customer collaboration* over contract negotiation.
4. *Responding to change* over following a plan.

According to the authors, we should interpret these values as follows: “While there is value in the items on the right, we value the items on the left more” [4]. When we examine these values more closely, we can observe that they all share a common philosophy, which is that software engineering should be a fast process in which communication and a short feedback loop is critical to avoid missteps. Since 2001, a variety of different programming models have arisen, each incorporating these Agile principles in their own way. The most remarkable new practice is Test-driven development (TDD). Recall that an integration test is a black-box test and that as such, we can actually construct the test case in advance and write the implementation afterwards. This concept is also prevalent in TDD. This practice depicts that if we want to extend the functionality of the application, we should first modify the test cases (or add new test cases) and then modify the application code until every test case is passing [3].

2.3.2 The need for Agile

In the wake of the world economic crisis, software companies have been forced to devote efforts into researching how they can reduce their overall expenses. This research has concluded that in order to cut financial risks, developers should reduce the *time-to-market* of their applications [18]. As a result of this, the Agile methodologies have received increased attention in scientific literature since this philosophy strives to deliver a minimal version as soon as possible. Afterwards, we can incrementally add additional features. This practice indeed results in a shorter time-to-market and lower costs, since one can decide to cancel the project much earlier in the process.

Additionally, 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 traditional methodologies (Figure 2.4).

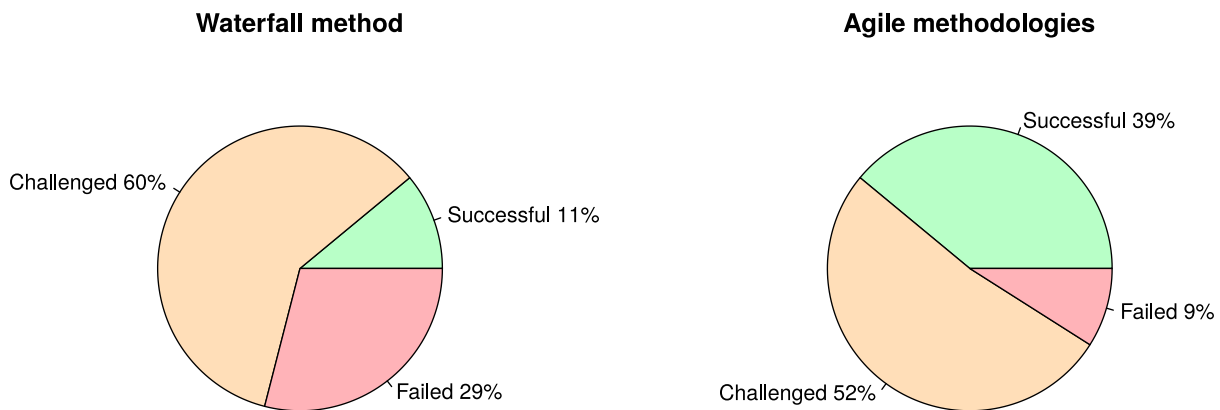


Figure 2.4: Success rate of Agile methodologies [16].

2.3.3 Continuous Integration

In traditional software development, the design phase usually leads to a representation of the required functionality in multiple, stand-alone modules. These modules are then implemented separately by the individual developers and are afterwards integrated into one monolithic application. This operation can prove to be very complicated since every developer can make their assumptions at the start of the project. Ultimately, these assumptions may render the components mutually incompatible. Furthermore, since it can take several weeks to months before this integration takes place, the developers often need to rewrite old code. Eventually, this will lead to unanticipated delays and costs [33].

Contrarily, Agile development methodologies advocate the idea of frequent, yet small deliverables. In order to obtain frequent deliverables, we require to build the code often and integrate the modules multiple times, *continuously*, rather than just once at the end. Another advantage of frequent deliverables is the early identification of problems [14]. We refer to this practice as Continuous Integration (CI) [33]. Note that this idea has existed before the creation of the Agile manifesto. The first notorious software company that has adopted CI is Microsoft in 1989 [8]. Cusumano reports that Microsoft typically builds the entire application at least once per day and as such requires developers to integrate and test their changes multiple times per day.

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

1. **Implementation:** In the first phase, every developer writes code individually for their assigned module. At a regular interval, the code is committed to the remote repository.
2. **Integration:** When the developer commits their changes, they simultaneously fetch the changes other developers have made to their modules. Afterwards, the developer must integrate these remote changes with their local module to ensure these remain compatible. In case of a conflict, the developer is responsible for resolving this locally [27].
3. **Test:** After the developer has successfully integrated the remote modules to the locale module, the test suite must be executed to verify that no regressions have been introduced.

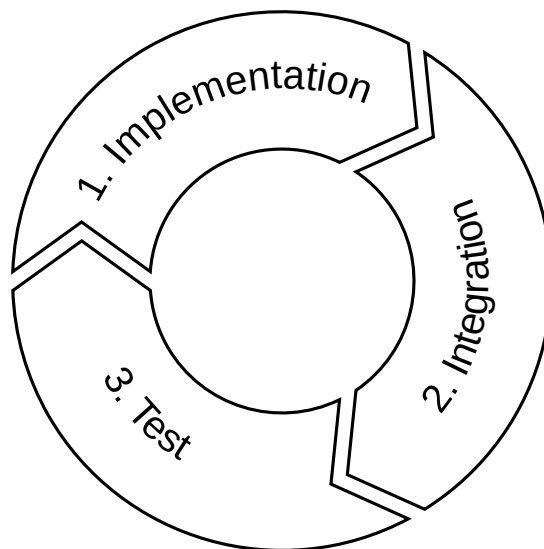


Figure 2.5: Development Life Cycle with Continuous Integration.

Adopting Continuous Integration can prove to be a lengthy and repetitive task. Luckily, a variety of tools and frameworks exist to automate this process. These tools are typically attached to a version control system (e.g. Git, Mercurial, ...) using a *post-receive* hook [33]. Every time the developers push a commit to the repository, the hook will notify the CI system. The CI system will respond accordingly by automatically building the code and executing the test cases. Optionally, we can configure the CI system to automatically publish successful builds to the end-users, a process referred to as *Continuous Delivery*. This paper will now proceed by discussing four prominent CI systems.

Jenkins

Jenkins CI⁴, “the leading open source automation server”, was started in 2004 by Kohsuke Kawaguchi as a hobby project. Initially launched as Hudson, the project was later re-named to Jenkins due to trademark issues [33]. Jenkins is programmed in Java and is currently maintained by volunteers. 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 in a private environment. Furthermore, Jenkins provides an open ecosystem that encourages developers to create new plugins and extend its functionality. Market research conducted by ZeroTurnaround in 2016 revealed that Jenkins is the preferred CI tool by 60 % of the developers [25].



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

GitHub Actions

Following the successful beta of GitHub Actions which had started in August 2019, GitHub launched its Continuous Integration system later that year in November⁵. GitHub Actions executes builds in the cloud on servers owned by GitHub, restricting its use to GitHub repositories. Support for GitHub Enterprise repositories is not currently available. Developers can define builds using workflows, which can run both on Linux, Windows as well as macOS hosts. Private repositories are allowed a fixed amount of free build minutes per month, while builds of public repositories are always free of charges [11]. Similar to Jenkins, we can extend GitHub Actions with custom plugins. These plugins can be composed either using a Docker container or using native JavaScript [1]. However, since this service is recent, it does not yet offer many plugins.



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

⁴<https://jenkins.io/>

⁵<https://github.blog/2019-08-08-github-actions-now-supports-ci-cd/>

GitLab CI

GitLab, the main competitor to GitHub, announced its own Continuous Integration service in late 2012 named GitLab CI⁶. The build configuration is specified in a *pipeline* and is executed by *GitLab Runners*. Developers may host these runners by themselves, or use shared runners hosted by GitLab [2]. Equivalent to the previously mentioned GitHub Actions, shared runners can be used for free by public repositories and are bound by quota for private repositories [13]. A downside of using GitLab CI is the absence of a community-driven plugin system, but support for plugins is planned⁷.



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

Travis CI

The final CI platform that will be discussed is Travis CI. This CI system was launched in 2011 and is only compatible with GitHub repositories. Travis CI build tasks are configured similarly to GitLab CI but lack support for self-hosted runners. In addition to commit-triggered builds, we can also schedule daily, weekly or monthly builds using *cronjobs*. Similar to GitHub Actions, open-source projects can use this service at zero cost, and a paid plan exists for private repositories [10]. It is not possible to create custom plugins, but Travis CI already features built-in support for a variety of programming languages. In 2020, almost one million projects are using Travis CI [34].



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

⁶<https://about.gitlab.com/blog/2012/11/13/continuous-integration-server-from-gitlab/>

⁷<https://gitlab.com/gitlab-org/gitlab/issues/15067>

Chapter 3

Related work

In the previous chapter, we have stressed the paramount importance of frequently integrating one's changes into the upstream repository. This process can prove to be a complex and lengthy operation. As a result, software engineers have sought and found ways to automate this task. These solutions and practices embody Continuous Integration (CI). However, CI is not the golden bullet for software engineering, as there is a flip side to applying this practice. After every integration, we must execute the entire test suite to ensure that we have not introduced any regressions. As the project evolves and the size of the codebase increases, the number of test cases will increase accordingly to preserve a sufficiently high coverage level [30]. Walcott, Soffa and Kapfhammer illustrate the magnitude of this problem by providing an example of a project consisting of 20 000 lines of code, whose test suite requires up to seven weeks to complete [35].

Fortunately, developers and researchers have found multiple techniques to address the scalability issues of ever-growing test suites. We can classify the techniques currently known in literature into three categories [30]. These categories are Test Suite Minimisation (TSM), Test Case Selection (TCS) or Test Case Prioritisation (TCP). We can apply each technique to every test suite, but the outcome will be different. TSM and TCS will have an impact on the execution time of the test suite, at the cost of a reduced test coverage level. In contrast, TCP will have a weaker impact on the execution time but will not affect the test adequacy.

The following sections will discuss these three approaches in more detail and provide accompanying algorithms. Because the techniques are very similar, the corresponding algorithms can (albeit with minor modifications) be used interchangeably for every approach. The final section of this chapter will investigate the adoption and integration of these techniques in modern software testing frameworks.

3.1 Classification of approaches

3.1.1 Test Suite Minimisation

The first technique is called Test Suite Minimisation (TSM), also referred to as *Test Suite Reduction* in literature. This technique will try to reduce the size of the test suite by permanently removing redundant test cases. This problem has been formally defined by Rothermel [37] in definition 1 and illustrated in Figure 3.1.

Definition 1 (Test Suite Minimisation).

Given:

- $T = \{t_1, \dots, t_n\}$ a test suite consisting of test cases t_j .
- $R = \{r_1, \dots, r_m\}$ a set of requirements that must be satisfied in order to provide the desired “adequate” testing of the program.
- $\{T_1, \dots, T_m\}$ subsets of test cases in T , 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 .

Subsequently, we can define Test Suite Minimisation as the task of finding a subset T' of test cases $t_j \in T$ that satisfies every requirement r_i .

If we apply the concepts of the previous chapter to the above definition, we can interpret the set of requirements R as source code lines that must be covered. A requirement r_i can subsequently be satisfied by any test case $t_j \in T$ that belongs to the subset T_i . Observe that the problem of finding T' is closely related to the *hitting set problem* (definition 2) [37].

Definition 2 (Hitting Set Problem).

Given:

- $S = \{s_1, \dots, s_n\}$ a finite set of elements.
- $C = \{c_1, \dots, c_n\}$ 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' corresponds to the hitting set of T_i s. In order to effectively minimise the amount of tests in the test suite, T' should be the minimal hitting set [37]. Since we can reduce this problem to the NP-complete *Vertex Cover*-problem, we know that this problem is NP-complete as well [12].

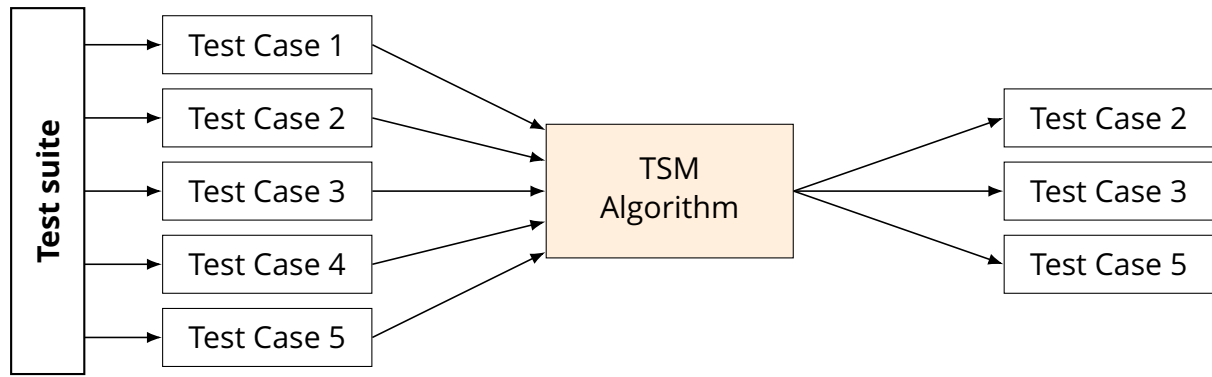


Figure 3.1: Test Suite Minimisation.

3.1.2 Test Case Selection

The second approach closely resembles the previous one. However, instead of permanently removing redundant test cases, Test Case Selection (TCS) has a notion of context. In this algorithm, we will not calculate the minimal hitting set at runtime, but before executing the test suite, we will perform a *white-box static analysis* of the source code. This analysis identifies which parts of the source code have been changed and executes only the corresponding test cases. Subsequent executions of the test suite will require a new analysis, thus making the selection temporary (Figure 3.2) and modification-aware [37]. Rothermel and Harrold define this formally in definition 3.

Definition 3 (Test Case Selection).

Given:

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

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

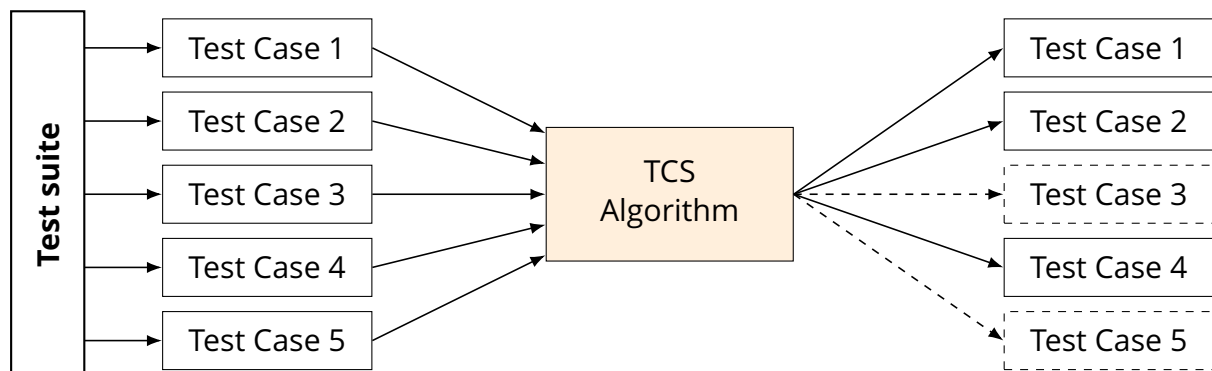


Figure 3.2: Test Case Selection.

3.1.3 Test Case Prioritisation

Both TSM and TCS attempt to execute as few tests as possible to reduce the execution time of the test suite. Nevertheless, in some cases, we may require to execute every test case to guarantee correctness. In this situation, we can still optimise the test suite. Test Case Prioritisation (TCP) aims to find a permutation of the sequence of test cases, rather than eliminating specific tests from being executed (Figure 3.3). We choose the order of the permutation in such a way that we can complete a predefined objective as soon as possible. Once we have achieved our objective, we can early terminate the execution of the test suite. In the worst-case scenario, we will still execute every test case. Some examples of objectives include covering as many lines of code as fast as possible or executing tests ordered on their probability of failure [37]. Definition 4 provides a formal definition of this approach.

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 test cases to find the optimal permutation.

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

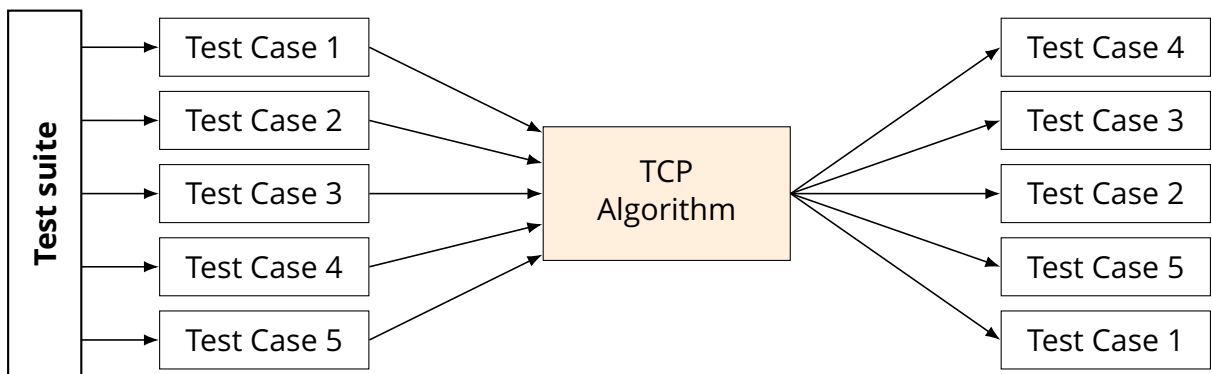


Figure 3.3: Test Case Prioritisation.

3.2 Algorithms

TCP is essentially an extended version of TSM since we can first execute the minimised test suite and afterwards the remaining test cases. Additionally, section 3.1.1 has explained that TSM is an instance of the minimal hitting set problem, which is an NP-complete problem. Consequently, we know that both TSM and TCP are NP-complete problems as well and therefore, we require the use of *heuristics*. A heuristic is an experience-based method that can be applied to solve a hard to compute problem by finding a fast approximation [19]. However, the found solution will mostly be sub-optimal, or sometimes the algorithm might even fail to find any solution at all. Given the relation between TSM and the minimal hitting set problem, we can implement an optimisation algorithm by modifying any known heuristic that finds the minimal hitting set. This paper will now proceed by discussing a selection of these heuristics. The used terminology and the names of the variables have been changed to ensure mutual consistency between the algorithms. Every algorithm has been adapted to adhere to the conventions provided in definitions 5 and 6.

Definition 5 (Naming convention).

- $TS = \{T_1, \dots, T_n\}$: the set of all test cases t in the test suite.
- $RS = \{T_1, \dots, T_n\} \subseteq TS$: the representative set of test cases t that have been selected by the algorithm.
- $C = \{c_1, \dots, c_m\}$: the set of all source code lines in the application, that are covered by at least one test case $T \in TS$.
- $CT = [CT_1 \ \dots \ CT_m]$: the list of test groups.
 - $CT_c = \{T_1, \dots, T_n\} \subseteq TS$: the test group c , which corresponds to the set of all test cases $T \in TS$ that cover the source code line $c \in C$.
- $TL = [TL_1 \ \dots \ TL_n]$: the list of coverage groups.
 - $TL_t = \{c_1, \dots, c_m\} \subseteq C$: the set of all source code lines $c \in C$ that are covered by test case $t \in TS$.

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, we can use $Card(S)$ to denote the cardinality of S .

3.2.1 Greedy algorithm

The first algorithm is a *greedy* heuristic, which was initially designed by Chvatal to find an approximation for the set-covering problem [30]. A greedy algorithm always makes a locally optimal choice, assuming that this will eventually lead to a globally optimal solution [7]. Algorithm 1 presents the Greedy algorithm for Test Suite Minimisation. The objective of the algorithm is to construct a set of test cases that cover every line in the code, by requiring as few test cases as possible.

Initially, the algorithm starts with an empty representative set RS , the set TS of all test cases and the set C of all coverable source code lines. Furthermore, TL denotes the set of coverage groups as specified in the definition. In essence, the algorithm will iteratively select test cases from TS and add them to RS . The locally optimal choice is always to select the test case that will contribute the most still uncovered lines, ergo the test case t for which the cardinality of the intersection between C and TL_t is maximal. After every iteration, we remove the code lines TL_t from C , since these are now covered. We repeat this selection process until C is empty, which indicates that we have covered every source code line. Afterwards, when we execute the test suite, we only need to execute test cases in RS . We can apply this algorithm to Test Case Prioritisation as well, by changing the type of RS to a list instead. We require a list to maintain the insertion order since this is equivalent to the ideal order of execution.

Algorithm 1 Greedy algorithm for Test Suite Minimisation.

Input: the test suite TS , all coverable lines C , the list of coverage groups TL

Output: representative set $RS \subseteq TS$ of test cases to execute

```

1: procedure GREEDYTSM( $TS, C, TL$ )
2:    $RS \leftarrow \emptyset$ 
3:   while  $C \neq \emptyset$  do
4:      $t_{max} \leftarrow 0$ 
5:      $tl_{max} \leftarrow \emptyset$ 
6:     for all  $t \in TS$  do
7:        $tl_{current} \leftarrow C \cap TL[t]$ 
8:       if  $|tl_{current}| > |tl_{max}|$  then
9:          $t_{max} \leftarrow t$ 
10:         $tl_{max} \leftarrow tl_{current}$ 
11:     $RS \leftarrow RS \cup \{t_{max}\}$ 
12:     $C \leftarrow C \setminus tl_{max}$ 
13:  return  $RS$ 

```

3.2.2 HGS

The second algorithm is the HGS algorithm. The algorithm was named after its creators Harrold, Gupta and Soffa [17]. Similar to the Greedy algorithm (section 3.2.1), this algorithm will also iteratively construct the minimal hitting set. However, instead of considering the coverage groups TL , the algorithm uses the test groups CT . More specifically, we will use the distinct test groups, denoted as CTD . We consider two test groups CT_i and CT_j as distinct if they differ in at least one test case. The pseudocode for this algorithm is provided in Algorithm 2.

The algorithm consists of two main phases. The first phase begins by constructing an empty representative set RS in which we will store the selected test cases. Subsequently, we iterate over every source code line $c \in C$ to create the corresponding test groups CT . As mentioned before, we will reduce this set to CTD for performance reasons and as such, only retain the distinct test groups. Next, we select every test group of which the cardinality is equal to 1 and add these to RS . The representative set will now contain every test case that covers precisely one line of code, which is exclusively covered by that single test case. Afterwards, we remove every covered line from C . The next phase consists of repeating this process for increasing cardinalities until C is empty. However, since the test groups will now contain more than one test case, we need to make a choice on which test case to select. The authors prefer the test case that covers the most remaining lines. In the event of a tie, we defer the choice until the next iteration.

The authors have provided an accompanying calculation of the computational time complexity of this algorithm [17]. In addition to the naming convention introduced in definition 5, 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 test group with the most test cases. In the HGS algorithm we need to perform two steps 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 we all need to examine once, the computational cost of this step is equal to $O(n * MAX_CARD)$. For the next step, in order to determine which test case we should include in the representative set RS , we need 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 ([17]).**Input:** distinct test groups CTD , total amount of test cases $nt = Card(TS)$ **Output:** representative set $RS \subseteq TS$ of test cases to execute

```

1: function SELECTTEST( $CTD, nt, MAX\_CARD, size, list, marked$ )
2:    $count \leftarrow array[1 \dots nt]$  ▷ initially 0
3:   for all  $t \in list$  do
4:     for all  $group \in CTD$  do
5:       if  $t \in group \wedge \neg marked[group] \wedge Card(group) = size$  then
6:          $count[t] \leftarrow count[t] + 1$ 
7:    $max\_count \leftarrow MAX(count)$ 
8:    $tests \leftarrow \{t | t \in list \wedge count[t] = max\_count\}$ 
9:   if  $|tests| = 1$  then return  $tests[0]$ 
10:  else if  $|tests| = MAX\_CARD$  then return  $tests[0]$ 
11:  else return SELECTTEST( $CTD, nt, MAX\_CARD, size + 1, tests, marked$ )
12: procedure HGSTSM( $CTD, nt$ )
13:    $n \leftarrow Card(CTD)$ 
14:    $marked \leftarrow array[1 \dots n]$  ▷ initially false
15:    $MAX\_CARD \leftarrow MAX(\{Card(group) | group \in CTD\})$ 
16:    $RS \leftarrow \bigcup \{singleton | singleton \in CTD \wedge Card(singleton) = 1\}$ 
17:   for all  $group \in CTD$  do
18:     if  $group \cap RS \neq \emptyset$  then
19:        $marked[group] \leftarrow true$ 
20:    $current \leftarrow 1$ 
21:   while  $current < MAX\_CARD$  do
22:      $current \leftarrow current + 1$ 
23:      $list \leftarrow \{t | t \in grp \wedge grp \in CTD \wedge Card(grp) = current \wedge \neg marked[grp]\}$ 
24:     while  $list \neq \emptyset$  do
25:        $next \leftarrow SELECTTEST(current, list)$ 
26:        $reduce \leftarrow false$ 
27:       for all  $group \in CTD$  do
28:         if  $next \in group$  then
29:            $marked[group] = true$ 
30:           if  $Card(group) = MAX\_CARD$  then
31:              $reduce \leftarrow true$ 
32:       if  $reduce$  then
33:          $MAX\_CARD \leftarrow MAX(\{Card(grp) | grp \in CTD \wedge \neg marked[grp]\})$ 
34:          $RS \leftarrow RS \cup \{next\}$ 
35:          $list \leftarrow \{t | t \in grp \wedge grp \in CTD \wedge Card(grp) = current \wedge \neg marked[grp]\}$ 
36:   return  $RS$ 

```

3.2.3 ROCKET algorithm

The third and final algorithm is the ROCKET algorithm. This algorithm has been presented by Marijan, Gotlieb and Sen [26] as part of a case study to improve the testing efficiency of industrial video conferencing software. Contrarily to the previous algorithms, which attempted to execute as few test cases as possible, this algorithm does execute the entire test suite. 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 successive failure rate, while also maximising the number of executed test cases in a limited time frame. In the implementation below, we will consider an infinite time frame as this is a domain-specific constraint and irrelevant for this thesis. This algorithm will yield a total ordering of all the test cases in the test suite, ordered using a weighted function.

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

- $TS = \{T_1, \dots, T_n\}$: the set of test cases to prioritise.
- $E = \begin{bmatrix} E_1 & \dots & E_n \end{bmatrix}$: the execution time of each test case.
- $F = \begin{bmatrix} F_1 & \dots & F_n \end{bmatrix}$: the failure statuses of each test case.
 - $F_t = \begin{bmatrix} f_1 & \dots & f_m \end{bmatrix}$: the failure status of test case t over the previous m successive executions. $F_{ij} = 1$ if test case i has failed in execution ($current - j$), 0 if it has passed.

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, we construct an $m \times n$ failure matrix MF and fill it using the following formula.

$$MF[i, j] = \begin{cases} 1 & \text{if } F_{ji} = 1 \\ -1 & \text{otherwise} \end{cases}$$

Table 3.1 contains an example of this matrix MF . In this table, we consider the hypothetical failure rates of the last two 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

Table 3.1: Example of the failure matrix MF .

Afterwards, we fill P with the cumulative priority of each test case. We can calculate the priority of a test case by multiplying its failure rate with a domain-specific weight heuristic ω . This heuristic reflects the probability of repeated failures of a test case, given earlier failures. In their paper [26], the authors apply the following weights:

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

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

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 test cases 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 for Test Case Prioritisation.

Input: the test suite TS , the execution times of the test cases E , the amount of previous executions to consider m , the failure statuses F for each test case over the previous m executions

Output: priority P of the test cases

```

1: procedure ROCKETTCP( $TS, E, m, F$ )
2:    $n \leftarrow \text{Card}(TS)$ 
3:    $P \leftarrow \text{array}[1 \dots n]$  ▷ initially 0
4:    $MF \leftarrow \text{array}[1 \dots m]$ 
5:   for all  $i \in 1 \dots m$  do
6:      $MF[i] \leftarrow \text{array}[1 \dots n]$ 
7:     for all  $j \in 1 \dots n$  do
8:       if  $F[j][i] = 1$  then  $MF[i][j] \leftarrow -1$ 
9:       else  $MF[i][j] \leftarrow 1$ 
10:  for all  $j \in 1 \dots n$  do
11:    for all  $i \in 1 \dots m$  do
12:      if  $i = 1$  then  $P[j] \leftarrow P[j] + (MF[i][j] * 0.7)$ 
13:      else if  $i = 2$  then  $P[j] \leftarrow P[j] + (MF[i][j] * 0.2)$ 
14:      else  $P[j] \leftarrow P[j] + (MF[i][j] * 0.1)$ 
15:   $Q \leftarrow \{P[j] | j \in 1 \dots n\}$  ▷ distinct priorities
16:   $G \leftarrow \text{array}[1 \dots \text{Card}(Q)]$  ▷ initially empty sets
17:  for all  $j \in 1 \dots n$  do
18:     $p \leftarrow P[j]$ 
19:     $G[p] \leftarrow G[p] \cup \{j\}$ 
20:  Sort every group in  $G$  based on ascending execution time in  $E$ .
21:  Sort  $P$  according to which group it belongs and its position within that group.
22:  return  $P$ 

```

3.3 Adoption in testing frameworks

In the final section of this chapter, we will investigate how existing software testing frameworks have implemented these and other optimisation techniques.

3.3.1 Gradle and JUnit

Gradle¹ is a dependency manager and development suite for Java, Groovy and Kotlin projects. It supports multiple plugins to automate tedious tasks, such as configuration management, testing and deploying. One of the supported testing integrations is JUnit², which is the most widely used testing framework by Java developers. JUnit 5 is the newest version which is still under active development as of today. Several prominent Java libraries and frameworks, such as Android and Spring have integrated JUnit as the preferred testing framework. The testing framework offers mediocre support for features that optimise the execution of the test suite, primarily when used in conjunction with Gradle. The following three key elements are available:

1. **Parallel test execution:** The Gradle implementation of JUnit features multiple *test class processors*. A test class processor is a component which processes Java classes to find all the test cases, and eventually to execute them. One of these processors is the `MaxNParallelTestClassProcessor`, which is capable of running a configurable amount of test cases in parallel. Concurrently executing the test cases results in a significant speed-up of the overall test suite execution.
2. **Prioritise failed test cases:** Gradle provides a second useful test class processor: the `RunPreviousFailedFirstTestClassProcessor`. This processor will prioritise test cases that have failed in the previous run. This practice is similar to the ROCKET-algorithm (section 3.2.3), but the processor does not take into account the duration of the test cases.
3. **Test order specification:** JUnit allows us to specify the sequence in which it will execute the test cases. By default, it uses a random yet deterministic order³. The order can be manipulated by annotating the test class with the `@TestMethodOrder`-annotation, or by applying the `@Order(int)`-annotation to an individual test case. However, we can only use this feature to alter the order of test cases within the same test class. JUnit does not support inter-test class reordering. We could use this feature to (locally) 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/>

3.3.2 Maven Surefire

A commonly used alternative to Gradle is Apache Maven⁴. This framework also supports executing JUnit test cases using the Surefire plugin. As opposed to Gradle, Surefire does offer multiple options to specify the order in which the test cases will be executed using the `runOrder` property. Without any configuration, Maven will run the test cases in alphabetical order. By switching the `runOrder` property to `failedFirst`, we can tell Maven to prioritise the previously failed test cases. Another supported value is `balanced`, which orders test cases based on their duration. Finally, we can choose to implement a custom ordering scheme for absolute control.

3.3.3 OpenClover

OpenClover⁵ is a code coverage tool for Java and Groovy projects. It was created by Atlassian and open-sourced in 2017. OpenClover 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. These features include powerful integrations with development software and prominent Continuous Integration systems. Furthermore, OpenClover can automatically analyse 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, we can interpret these predictions to implement Test Case Selection and therefore reduce the test suite execution time.

⁴<http://maven.apache.org/>

⁵<https://openclover.org>

Chapter 4

Proposed framework: VeloCity

The implementation part of this thesis will provide 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 efficiency and speed of Continuous Integration. This paper will now proceed by first describing the design goals of the framework, after which a high-level schematic overview of the implemented architecture will be provided. The architecture consists of a seven-step pipeline, divided into three individual components. These steps will be elaborated in more detail in section 4.3. Subsequently, the next section will present the *Alpha* algorithm as a novel prioritisation algorithm, and this chapter will be concluded with an overview of the analytical features.

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 CI systems, programming languages and test frameworks. Similarly, a clear interface must be provided to integrate new 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 extensibility of the framework. 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 must support multiple algorithms. However, the performance of an algorithm might be dependent on the nature of the source code. An algorithm may offer a very high accuracy on one project but fall short on another. The framework should decide by itself which algorithm it should prefer, by measuring the performance of a prediction and subsequently infer which algorithm to use for future predictions.

4.2 Architecture

4.2.1 Agent

The first component that we will consider is the agent. The agent interacts directly with the source code and the test suite and is, therefore, the only component that is specific to the programming language and the test framework. Every programming language and test framework requires a different implementation of the agent, although these implementations are strongly related. This thesis provides a Java agent, which is available as a plugin for the Gradle and JUnit test framework, a combination which has previously been described in section 3.3.1. When the test suite is started, the plugin will contact the controller (section 4.2.2) to obtain the prioritised test case order and subsequently execute the test cases in that order. Afterwards, the plugin will send 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 one side and the predictor (section 4.2.3) on the other side. In order to satisfy the second design goal and as such allow language agnosticism, the controller exposes a REST-interface, to which the agent can communicate using the HTTP protocol. On the other side, the controller does not communicate directly with the predictor but stores prediction requests in a shared database instead. The predictor will periodically poll this database and update the request with the predicted order. Besides providing routing functionality between the agent and the predictor, the controller is additionally responsible for updating the meta predictor (section 4.3.4) by evaluating the accuracy of earlier predictions.

4.2.3 Predictor and Metrics

The final component is the predictor. The predictor is responsible for applying the prioritisation algorithms to predict the optimal execution order of the test cases. This order is calculated by first executing ten prioritisation algorithms and subsequently consulting the meta predictor to determine the preferred sequence. The predictor has been implemented in Python, because of its accessibility and compatibility with various existing libraries such as NumPy¹ and TensorFlow², to allow advanced prioritisation algorithms.

¹<https://numpy.org/>

²<https://www.tensorflow.org/>

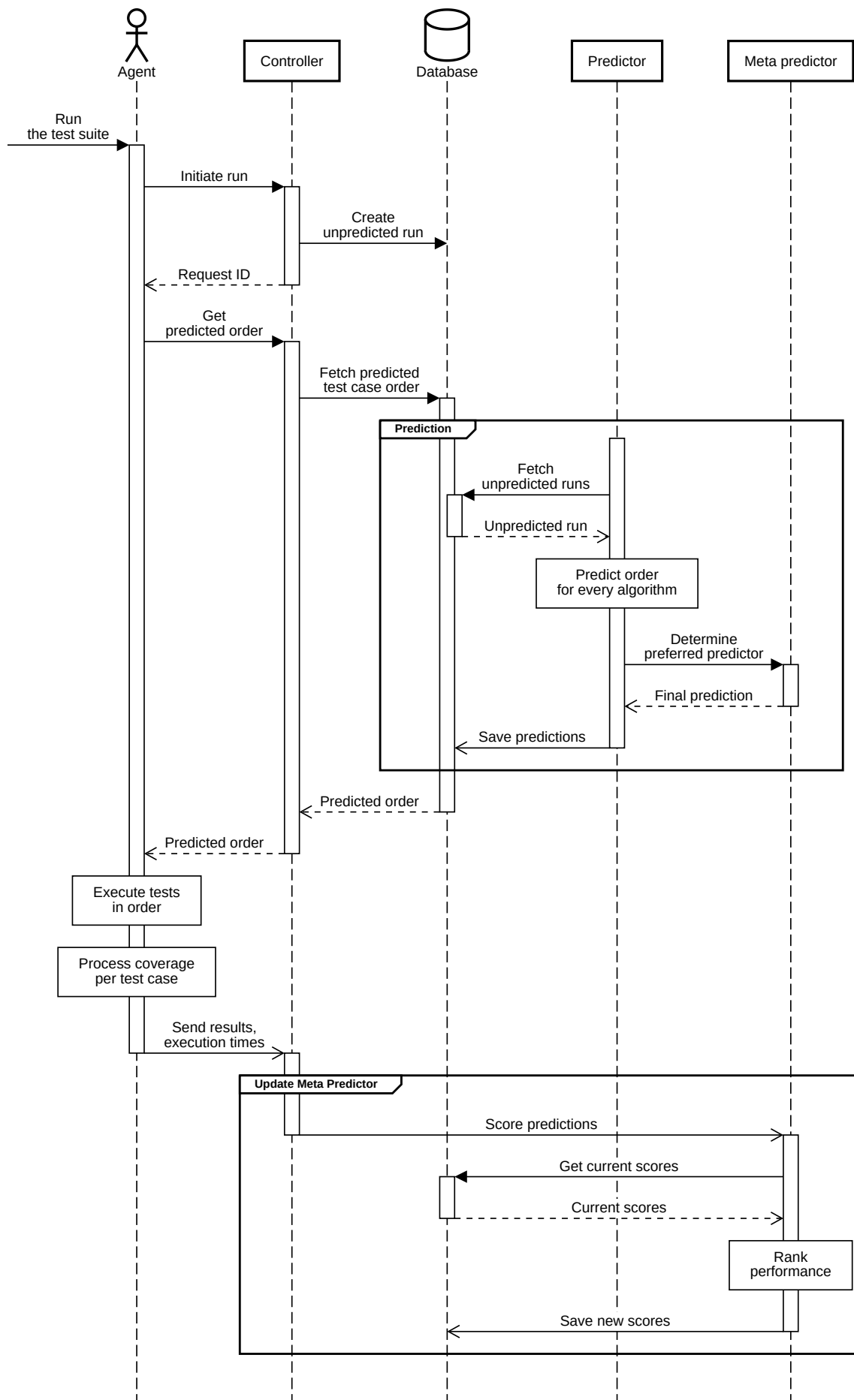


Figure 4.1: Sequence diagram of VeloCity.

4.3 Pipeline

This section will elaborate on the individual steps of the pipeline. We will review these steps by manually applying the pipeline on a hypothetical Java project. The very first prediction of a project will not use the meta predictor since this requires a previous prediction. Therefore, for the sake of simplicity, we will assume a steady-state situation, which implies that we have previously executed the prediction at least once.

4.3.1 Initialisation

In order to integrate VeloCity in an existing Gradle project, we must modify the build script (`build.gradle`) in two places. The first change is to include and apply the plugin in the header of the file. Afterwards, we must configure the following three properties:

- `base`: the path to the Java source files, relative to the location of the build script. This path will typically resemble `src/main/java`.
- `repository`: the URL to the git repository that hosts the project. 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 to the controller.

Example 4.1 contains a minimal integration of the agent in a Gradle build script. The controller is hosted at the same machine as the agent and is reachable at port 8080.

```
1 buildscript {  
2     dependencies {  
3         classpath 'io.github.thepieterdc.velocity:velocity-junit  
4             :0.0.1-SNAPSHOT'  
5     }  
6 }  
7 apply plugin: 'velocity-junit'  
8  
9 velocity {  
10     base 'src/main/java/'  
11     repository 'https://github.com/user/my-java-project'  
12     server 'http://localhost:8080'  
13 }
```

Example 4.1: Minimal Gradle buildsript.

After we have configured the build script, we can execute the test suite. For the Gradle agent, this involves executing the `velocity` task, which commences the pipeline. This task requires an additional argument to be passed, which is the commit hash of the changeset to prioritise. In every discussed CI system, this commit hash is available as an environment variable.

The first step in the pipeline is for the agent to initiate a new test run at the controller. The agent can accomplish this 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 enqueue a new prioritisation request in the database. At some point in time, the controller, which is continuously polling the database, will see the request and process it in the next step.

4.3.2 Prediction

The predictor is, as was mentioned before, continuously monitoring the database for unpredicted test runs. When a new test run is detected, the predictor will execute every available prediction algorithm in order to obtain multiple prioritised test sequences. The following algorithms are available:

AllInOrder The first algorithm will generate a straightforward execution sequence by ranking every test case alphabetically. We will mainly use this sequence for benchmarking purposes in chapter 5.

AllRandom We will use the second algorithm for benchmarking purposes as well. This algorithm will shuffle the list of test cases and return an arbitrary ranking.

AffectedRandom This algorithm is similar to the previous algorithm, but it will only consider the test cases that cover modified source code lines. These test cases will be ordered arbitrarily, followed by the other test cases in the test suite in no particular order.

GreedyCoverAll This algorithm is the first of three implementations of the Greedy algorithm (section 3.2.1). This variant will execute the standard algorithm on 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. After it has prioritised these test cases, it will order the remaining test cases in the test suite 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.

HGSAII This algorithm is an implementation of the algorithm presented by Harrold, Gupta and Soffa (section 3.2.2). Similar to the `GreedyCoverAll` algorithm, this algorithm will prioritise every test case in the test suite.

HGSAffected This algorithm is identical to the previous `HGSAII` algorithm, apart from the fact 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 section 3.2.3.

Alpha The final algorithm is a custom algorithm inspired by the other implemented algorithms. Section 4.4 will further elaborate on the details.

Afterwards, the predictor will apply the meta predictor to determine the final prioritisation sequence. In its most primitive form, we can compare the meta predictor to a table which assigns a score to every algorithm. This score reflects the performance of the algorithm on this particular project. Section 4.3.4 will explain how this score is updated. Eventually, the meta predictor will elect the sequence of the algorithm with the highest score as the final prioritised order and persist this to the database.

4.3.3 Test case execution

While the predictor is determining the test execution order, the agent will poll the controller using the previously acquired identifier by sending a `GET` request to the `/runs/id` endpoint. Should the prioritisation order already be available, the controller will return this. One of the discussed features of Gradle in section 3.3.1 was the possibility to execute test cases in a chosen order by adding annotations. However, we cannot use this feature to implement the Java agent, since it only supports ordering test cases within the same test class. In order to allow complete control over the order of execution, we require a custom `TestProcessor` and `TestListener`.

A `TestProcessor` is responsible for processing every test class in the classpath and forwarding it, along with configurable options, to a delegate processor. The final processor in the chain will eventually perform the actual execution of the test cases. By default, the built-in processors will execute every test case in the test class it receives.

Every built-in processor will, by default, immediately execute every test case in the test class it processes. Since we want to execute test cases across test classes, the custom processor needs to work differently. The provided implementation of the agent will first load every received test class to obtain all test cases in the class using reflection. Afterwards, it will store every test case in a list and iterate this list in the prioritised order. For every test case t in the list, the custom processor will call the delegate processor with a tuple containing the test class and an array of options. This array will exclude every test case in the class except for t . This practice will forward the same test class to the delegate processor multiple times using a different option that restricts test execution to the chosen test case, resulting in the desired behaviour.

Furthermore, the agent calls a custom `TestListener` before and after every executed test case. This listener allows the agent to calculate the duration of the 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 the agent has finished executing all the test cases, it will send the test 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`.

When the controller receives this feedback information, it will update the meta predictor as follows. If every test case has passed, we do not update the meta predictor. The explanation for this choice is obvious. Since the objective of Test Case Prioritisation is to detect failures as fast as possible, every prioritised sequence is equally good if there are no failures at all. On the other hand, if a test case did fail, the meta predictor will inspect the predicted sequences. For every sequence, the meta predictor will calculate the duration until the first failed test case. Subsequently, it calculates the average of all these durations. Finally, the meta predictor will update the score of every algorithm by comparing its duration until the first failed test case to the average duration. For every algorithm that has a below-average duration, we increase the score and else decrease it. This process will eventually lead to the most accurate algorithms having a higher score, and these will, therefore, be preferred in following test runs.

4.4 Alpha algorithm

Besides the earlier presented Greedy, HGS and ROCKET algorithms (section 3.2), a custom algorithm has been implemented. The *Alpha* algorithm has been constructed by examining the individual strengths of the three preceding algorithms and subsequently combining their philosophies into a novel prioritisation algorithm. This paper will now proceed by providing its specification in accordance with the conventions described in definition 5. The corresponding pseudocode is listed in Algorithm 4.

The algorithm consumes the following inputs:

- $TS = \{T_1, \dots, T_n\}$: the set of test cases to prioritise.
- $AS = \{T_1, \dots, T_m\} \subseteq TS$: the set of *affected* test cases. A test case t is considered “affected” if t covers any modified source code line in the current commit.
- $C = \{c_1, \dots, c_m\}$: the set of all source code lines in the application, that are covered by at least one test case $t \in TS$.
- $F = \begin{bmatrix} F_1 & \dots & F_n \end{bmatrix}$: the failure statuses of each test case.
 - $F_t = \begin{bmatrix} f_1 & \dots & f_m \end{bmatrix}$: the failure status of test case t over the previous m successive executions. $F_{ij} = 1$ if test case i has failed in execution (*current* – j), 0 if it has passed.
- $D = \begin{bmatrix} D_1 & \dots & D_n \end{bmatrix}$: the execution times of each test case.
 - $D_t = \begin{bmatrix} d_{t1} & \dots & d_{tm} \end{bmatrix}$: the execution times of test case t . D_{ij} corresponds to the duration (in milliseconds) of test case i in execution (*current* – j).
- $TL = \begin{bmatrix} TL_1 & \dots & TL_n \end{bmatrix}$: the list of coverage groups.
 - $TL_t = \{c_1, \dots, c_o\} \subseteq C$: the set of all source code lines $c \in C$ that are covered by test case $t \in TS$.

The algorithm begins by determining the execution time E_t of every test case t . To calculate the execution time, we distinguish two cases. If t has passed at least once, we calculate the execution time as the average duration of every successful execution of t . In the unlikely event that t has always failed, we compute the average over every execution of t . This distinction is mandatory, since a failed test case might have been aborted prematurely, which introduces a bias in the timings.

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

Next, the algorithm executes every affected test case that has also failed at least once in its three previous executions. Consecutive failures reflect the behaviour of a developer attempting to resolve the bug that caused the test case failure in the first run of the chain. By particularly executing the affected failing test cases as early as possible, we anticipate a developer that resolves multiple failures one by one. This idea is also used by the ROCKET algorithm (section 3.2.3). If multiple affected test cases are failing, the algorithm sorts those test cases by assigning a higher priority to the test case with the lowest execution time. After every selection, we update C by subtracting the source code lines that are now covered by that test case.

Afterwards, we repeat the same procedure for every failed (yet unaffected) test case and likewise use the execution time as a tie-breaker. Where the previous phase helps a developer to get fast feedback about whether or not they have resolved the issue, this phase ensures that the other failing test cases are executed early as well. Similar to the previous step, we again update C after every prioritised test case.

Research (section 5.4.1) has indicated that on average, a minor fraction (10 % – 20 %) of all test runs will contain a failed test case. As a result, the previous two phases will often not select any test case at all. Therefore, we will now focus on executing test cases that cover affected code lines. More specifically, the third phase of the algorithm will execute every affected test case, sorted by decreasing cardinality of the intersection between C and the test group of that test case.

After every selection, we will update C as well. Since this phase uses C in the comparison, every selected test case can influence the next selected test case. The update process of C will ultimately lead to some affected test cases not strictly requiring to be executed, similar to the Greedy algorithm (section 3.2.1).

In the last phase, the algorithm will select the test cases based on the cardinality of the intersection between C and their test group. We repeat this process until C is empty and update C accordingly. When C is empty, the algorithm will yield the remaining test cases. Notice that these test cases will not contribute to the test coverage whatsoever since the previous iteration would already have selected every test case that would incur additional coverage. Subsequently, these test cases are de facto redundant and are therefore candidates for removal by TSM. 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: the test suite TS , the affected test cases AS , all coverable lines C , the failure statuses F for each test case over the previous m executions, the execution times D for each test case over the previous m executions, the list of coverage groups TL

Output: ordered list P , sorted by descending priority

```

1: procedure ALPHATCP( $TS, AS, C, F, D, TL$ )
2:   Construct  $E$  using  $D$  as described above.
3:    $P \leftarrow \text{array}[1 \dots n]$  ▷ initially 0
4:    $i \leftarrow n$ 
5:    $FTS \leftarrow \{t | t \in TS \wedge (F[t][1] = 1 \vee F[t][2] = 1 \vee F[t][3] = 1)\}$ 
6:    $AFTS \leftarrow AS \cap FTS$ 
7:   for all  $t \in AFTS$  do ▷ sorted by execution time in  $E$  (ascending)
8:      $C \leftarrow C \setminus TL[t]$ 
9:      $P[t] \leftarrow i$ 
10:     $i \leftarrow i - 1$ 
11:    $FTS \leftarrow FTS \setminus AFTS$ 
12:   for all  $t \in FTS$  do ▷ sorted by execution time in  $E$  (ascending)
13:      $C \leftarrow C \setminus TL[t]$ 
14:      $P[t] \leftarrow i$ 
15:      $i \leftarrow i - 1$ 
16:    $AS \leftarrow AS \setminus FTS$ 
17:   while  $AS \neq \emptyset$  do
18:      $t_{max} \leftarrow AS[1]$  ▷ any element from  $AS$ 
19:      $tl_{max} \leftarrow \emptyset$ 
20:     for all  $t \in AS$  do
21:        $tl_{current} \leftarrow C \cap TL_t$ 
22:       if  $|tl_{current}| > |tl_{max}|$  then
23:          $t_{max} \leftarrow t$ 
24:          $tl_{max} \leftarrow tl_{current}$ 
25:      $C \leftarrow C \setminus tl_{max}$ 
26:      $P[t] \leftarrow i$ 
27:      $i \leftarrow i - 1$ 
28:    $TS \leftarrow TS \setminus (AS \cup FTS)$ 
29:   while  $TS \neq \emptyset$  do
30:      $t_{max} \leftarrow TS[1]$  ▷ any element from  $TS$ 
31:      $tl_{max} \leftarrow \emptyset$ 
32:     for all  $t \in TS$  do
33:        $tl_{current} \leftarrow C \cap TL_t$ 
34:       if  $|tl_{current}| > |tl_{max}|$  then
35:          $t_{max} \leftarrow t$ 
36:          $tl_{max} \leftarrow tl_{current}$ 
37:      $C \leftarrow C \setminus tl_{max}$ 
38:      $P[t] \leftarrow i$ 
39:      $i \leftarrow i - 1$ 
40:   return  $P$ 

```

4.5 Analysis

In this last section, we will take a look at the analytical features of the framework. Since the predictor already generates various statistics about the project which are required by the prioritisation algorithm, we can reuse these. The implementation of the analysis tool comprises a stand-alone version of the predictor daemon and supports the following six commands:

affected: The first command will determine which test cases have been affected by the changes in the given commit. This information is calculated based on the coverage information that the predictor has obtained from its last execution. Example 4.2 contains an example output of this command.

```
1 $ predictor affected https://github.com/author/project f5a23e0  
2 FooTest.bar  
3 FooTest.foo
```

Example 4.2: Output of the affected-command.

durations: The second command will compute the mean execution time of every test case in the repository and return the test cases from slowest to fastest.

```
1 $ predictor durations https://github.com/author/project  
2 FooTest.foo: 200s  
3 OtherBarTest.bar: 100s
```

Example 4.3: Output of the durations-command.

failures: Similar to the previous command, this command will determine the failure ratio of every test case in the repository. This ratio is equivalent to the number of failures, divided by the total amount of executions. Note that this denominator is not the same for every test case, since the test suite may be extended with new test cases. The output (Example 4.4) will list the test cases from the highest to the lowest failure rate.

```
1 $ predictor failures https://github.com/author/project  
2 HelloWorldTest.hello: 25.00%  
3 FooBarTest.bar: 10.00%
```

Example 4.4: Output of the failures-command.

predict: This command allows the user to invoke the predictor by hand for the given test run. We can use this to test new algorithms, as opposed to the usual predictor daemon which does not support repeated predictions of the same test run. The result will contain the prioritised order as predicted by every available algorithm. An example output of this command is listed in Example 4.5.

```
1 $ predictor predict 1  
2 HGS: [FooTest.bar, OtherBarTest.bar, HelloWorldTest.hello]  
3 Alpha: [HelloWorldTest.hello, FooTest.bar, OtherBarTest.bar]
```

Example 4.5: Output of the predict-command.

predictions: This command allows the user to retrieve historical prediction results. For deterministic algorithms, this will result in the same output as the previous command (which will rerun the algorithms). However, since some algorithms contain a random factor, we do require a separate command that fetches the prediction of the given run from the database.

```
1 $ predictor predictions 3  
2 HGS: [FooTest.bar, OtherBarTest.bar, HelloWorldTest.hello]  
3 AllRandom: [HelloWorldTest.hello, OtherBarTest.bar, FooTest.bar]
```

Example 4.6: Output of the predictions-command.

scores: The final command yields the current score of every prediction algorithm in the meta predictor table, for the given project. The predictor will always return the test sequence that has been predicted by the algorithm with the current highest score. In Example 4.7 below, this would be the ROCKET algorithm.

```
1 $ predictor scores https://github.com/author/project  
2 AllInOrder: -3  
3 ROCKET: 7  
4 GreedyCoverAffected: 4
```

Example 4.7: Output of the scores-command.

Chapter 5

Evaluation

This chapter will evaluate the performance of the framework presented in the previous chapter. The first section introduces the two test subjects that will be used in subsequent experiments. The next section will restate the research questions formally and extend these. Afterwards, we will elaborate on the procedure of the data collection. 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

Dodona¹ is an open-source online learning environment created by Ghent University, which allows students from secondary schools and universities in Belgium and South-Korea to submit solutions to programming exercises and receive instant, automated feedback. The application is built on top of the Ruby-on-Rails web framework. To automate the testing process of the application, Dodona employs GitHub Actions (section 2.3.3) which executes the more than 450 test cases in the test suite and performs static code analysis afterwards. The application is tested using the default MiniTest testing framework and SimpleCov² is used to record the coverage of the test suite. Currently, the coverage ratio is approximately 89%. This analysis will consider builds between January 1 and May 17, 2020.

5.1.2 Stratego

The second test subject has been created for the Software Engineering Lab 2 course at Ghent University in 2018. The application was created for a Belgian gas transmission system operator and consists of two main components: a web frontend and a backend. This thesis will test the backend in particular since it is written in Java using the Spring framework. Furthermore, the application uses Gradle and JUnit to execute the 300 – 400 test cases in the test suite, allowing the Java agent (section 4.2.1) to be applied directly.

¹<https://dodona.ugent.be/>

²<https://github.com/colszowka/simplecov>

5.2 Research questions

We will answer the following research questions 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 tends to fail or not. The expectancy is that the probability of failure will be rather low, indicating that it is not strictly necessary to execute every test case and therefore making a case for Test Suite Minimisation.

RQ2: 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. We will only consider successful test runs, to reduce bias introduced by prematurely aborting the execution.

RQ3: Suppose that a test run has failed, what is the probability that the next run will fail as well? The ROCKET algorithm (section 3.2.3) relies on the assumption that if a test case has failed in a given test run, it is likely to fail in the subsequent run as well. This research question will investigate the likelihood of this hypothesis.

RQ4: How can Test Case Prioritisation be applied to Dodona and what is the resulting performance benefit? This research question will investigate the possibility to apply the VeloCity framework to the Dodona project and analyse how quickly the available predictors can discover a failing test case.

RQ5: Can the Java agent be applied to Stratego? Since the testing framework used by Stratego should be supported natively by the Java agent, this research question will verify its compatibility. Furthermore, we will analyse the prediction performance, albeit with a small number of relevant test runs.

5.3 Data collection

5.3.1 Travis CI build data

We can answer the first three research questions by analysing data from projects hosted on Travis CI (section 2.3.3). This data has been obtained from two sources.

The first source comprises a database [10] of 35 793 144 log files of executed test runs, which has been contributed by Durieux et al. The magnitude of the dataset (61.11 GiB)

requires a big data approach to parse these log files. Two straightforward MapReduce pipelines (Figure 5.1) have been created using the Apache Spark³ engine, to provide an answer to the first and second research question.

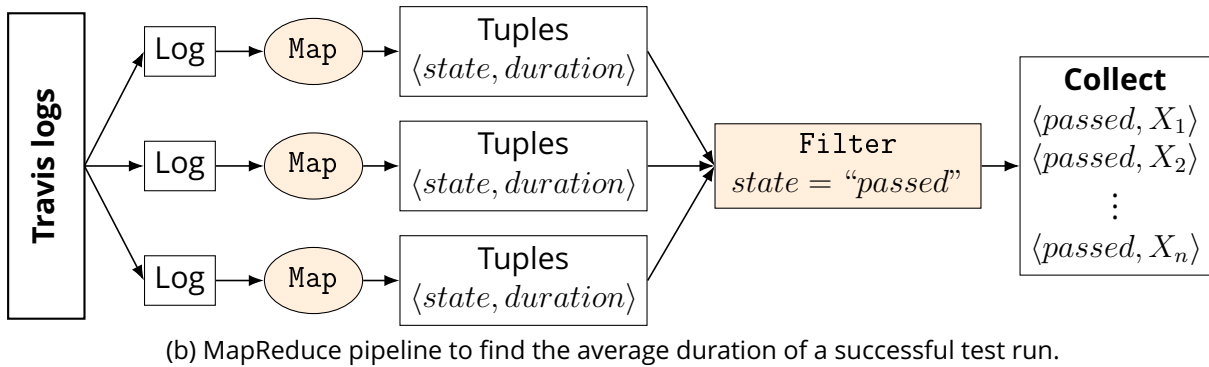
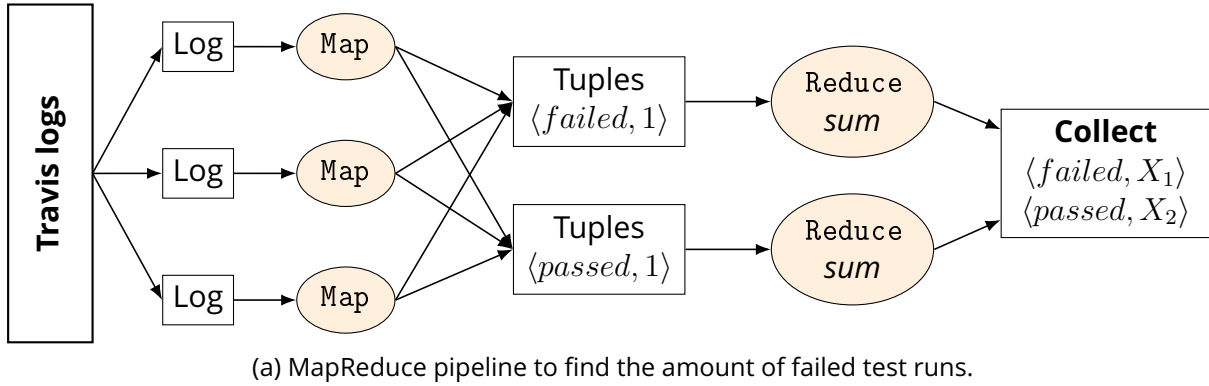


Figure 5.1: MapReduce pipelines for Travis CI data.

In addition to the first source, another 3 702 595 jobs have been analysed from the *TravisTorrent* project [5]. To identify which projects are using Travis CI, the authors have crawled the GitHub API and examined the build status of every commit to retrieve the run identifier. Subsequently, the Travis CI API is used to obtain information about both the build, as well as the project itself. This information includes the programming language, the amount of source code lines and the amount of failed test cases. The latter value provides an accurate answer to the first research question since it indicates why the test run has failed. Without this information, the test suite might have failed to compile as opposed to an actual failure in the test cases. Furthermore, the dataset includes the identifier of the previously executed test run, which we can use to answer the third research question. Additionally, the information contains the build duration. This dataset has been excluded from the second research question however, as the included execution time does not correspond to the actual duration reported on the webpage of Travis CI. The authors have provided a Google BigQuery⁴ interface to allow querying the dataset more efficiently. Appendix A contains the executed queries.

³<https://spark.apache.org/>

⁴<https://bigquery.cloud.google.com/>

5.3.2 Dodona data

As mentioned before, Dodona utilises the MiniTest testing framework in conjunction with SimpleCov to calculate the coverage. MiniTest will by default only emit the name of every failed test case, without any further information. Furthermore, SimpleCov can only calculate the coverage for the entire test suite and does not allow us to retrieve the coverage on a per-test basis. To answer the fourth research question and apply the VeloClty predictors to Dodona, a Python script has been created to reconstruct the conditions of every failed test run. The script first queries the API of GitHub Actions to find which test runs have failed. This thesis will consider 120 failed runs. For every failed commit, the script retrieves the parent commit and calculates the coverage on a per-test basis. This thesis will assume that the coverage of the parent commit resembles the coverage of the failed commit. The coverage is calculated by applying the following two transformations to the parent commits and subsequently rescheduling these in GitHub Actions:

- **Cobertura formatter:** The current SimpleCov reports can only be generated as HTML reports, preventing convenient analysis. We can resolve this problem by using the Cobertura formatter instead, which generates XML reports. The controller already supports the structure of these reports, as this formatter is commonly used by Java testing frameworks as well.
- **Parallel execution:** The Dodona test suite currently executes the test cases by four processes concurrently, to reduce the execution time. Every process individually records the code coverage, and at the end of the test suite, SimpleCov merges these separate reports into one. However, this process is not entirely thread-safe since the test suite requires shared resources. We do not require thread-safety to calculate the total coverage, but we do require this to generate the coverage on a per-test basis. As a result, parallel execution has been disabled in these experiments.

5.3.3 Stratego data

To integrate VeloClty with the existing Stratego codebase, we can use the instructions described in chapter 4. Afterwards, to analyse the prediction performance, we can take an approach similar to the previous test subject. The GitHub API has been used to identify the failed commits and to find their parent (successful) commits. The parent commits have subsequently been modified to use the VeloClty Java agent and have been executed using GitHub Actions.

5.4 Results

5.4.1 RQ1: Probability of failure

The two pie charts in Figure 5.2 illustrate the amount of failed and successful test runs. The leftmost chart visualises the failure rate in the dataset [10] by Durieux et al. 4 558 279 test runs out of the 28 882 003 total runs have failed, which corresponds to a failure probability of 18.74 %. The other pie chart uses data from the TravisTorrent [5] project. Since we can infer the cause of failure from this dataset, it is possible to obtain more accurate results. 42.89 % of the failed runs are due to a compilation failure where the test suite did not execute. For the remaining part of the runs, 225 766 out of 2 114 920 executions contain at least one failed test case, corresponding to a failure percentage of 10.67 %.

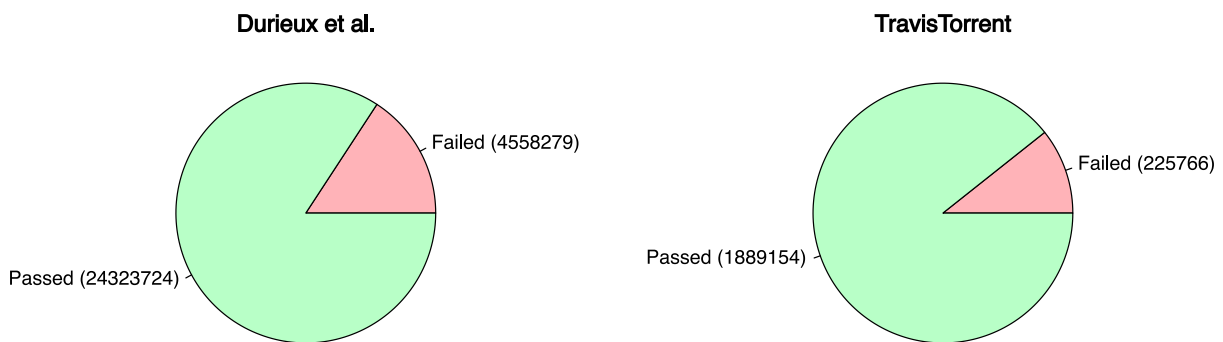


Figure 5.2: Probability of test run failure.

5.4.2 RQ2: Average duration of a test run

The dataset by Durieux et al. [10] has been refined to only include test runs that did not finish within 10 s. A lower execution time generally indicates that the test suite did not execute and that a compilation failure has occurred instead. Table 5.1 contains the characteristics of the remaining 24 320 504 analysed test runs. The median and average execution times suggest that primarily small projects are Travis CI, yet the maximum value is very high. Figure 5.3 confirms that 71 378 test runs have taken longer than one hour to execute. Further investigation has revealed that these are typically projects which are using mutation testing, 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 [10].

⁵A Ruby library for hypermedia (<https://github.com/plexus/yaks>).

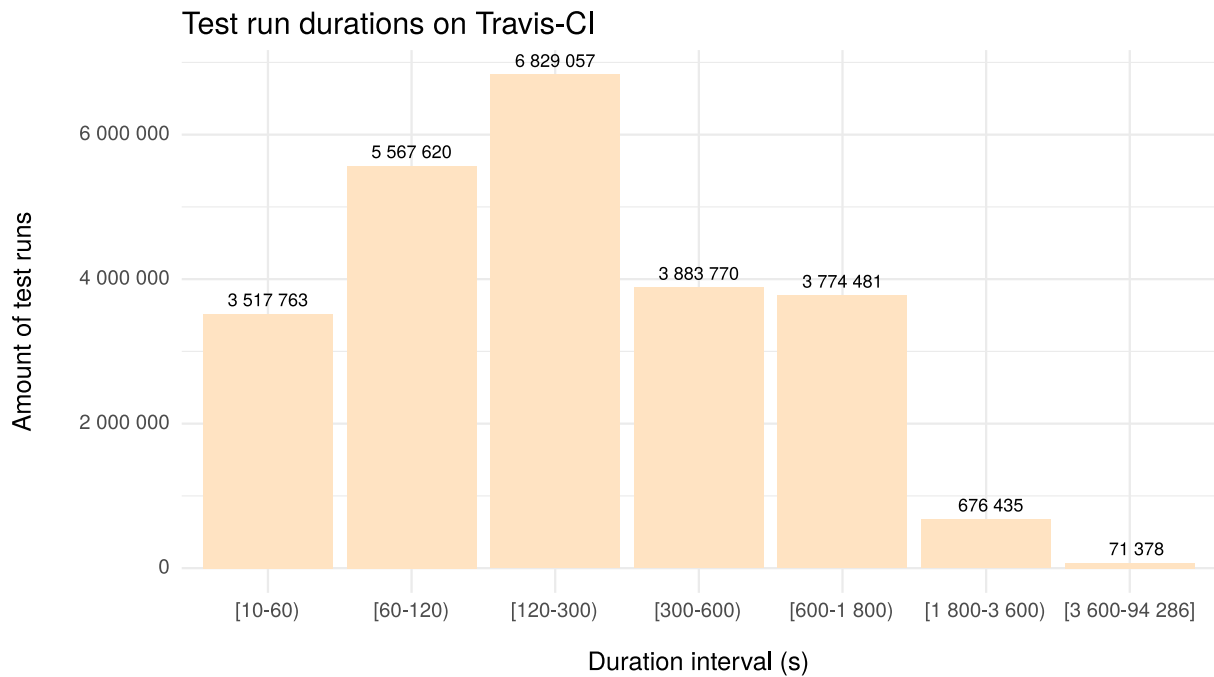


Figure 5.3: Test run durations on Travis CI.

5.4.3 RQ3: Consecutive failure probability

Because the TravisTorrent project is the only dataset that contains the identifier of the previous run, only runs from this project have been used. This dataset consists of 211 040 test runs, immediately following a failed execution. As illustrated in Figure 5.4, 109 224 of these test runs have failed as well, versus 101 816 successful test runs (51.76 %).

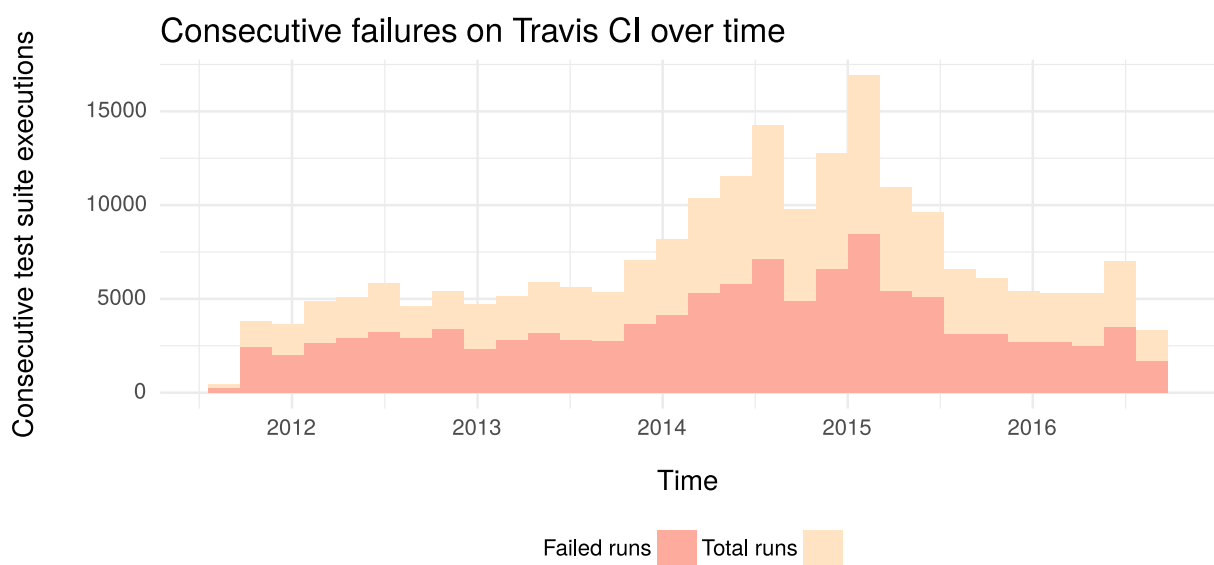


Figure 5.4: Consecutive test run failures on Travis CI.

5.4.4 RQ4: Applying Test Case Prioritisation to Dodona

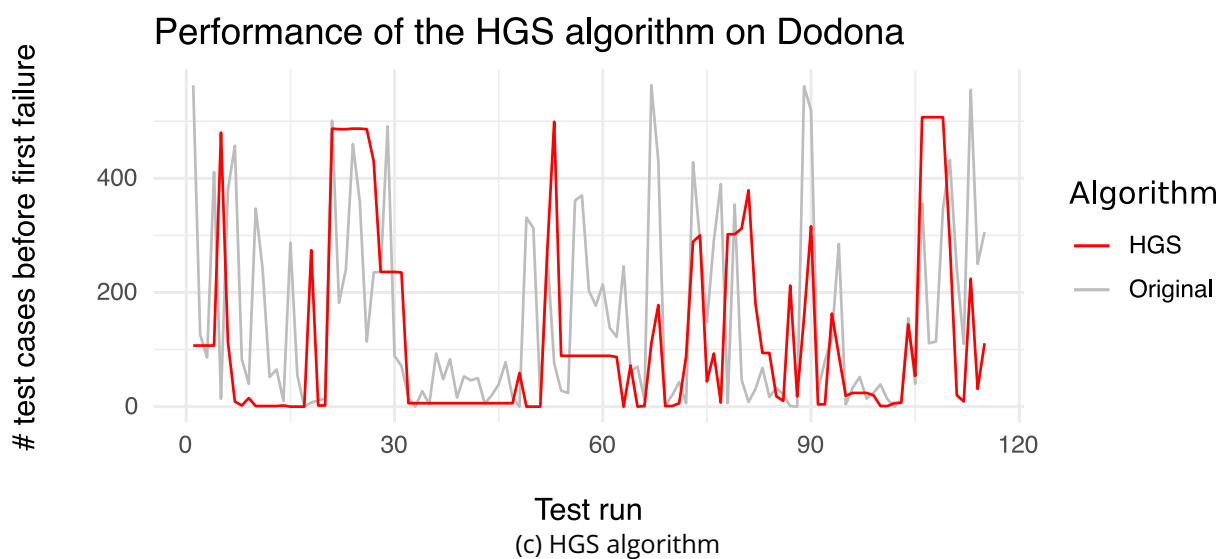
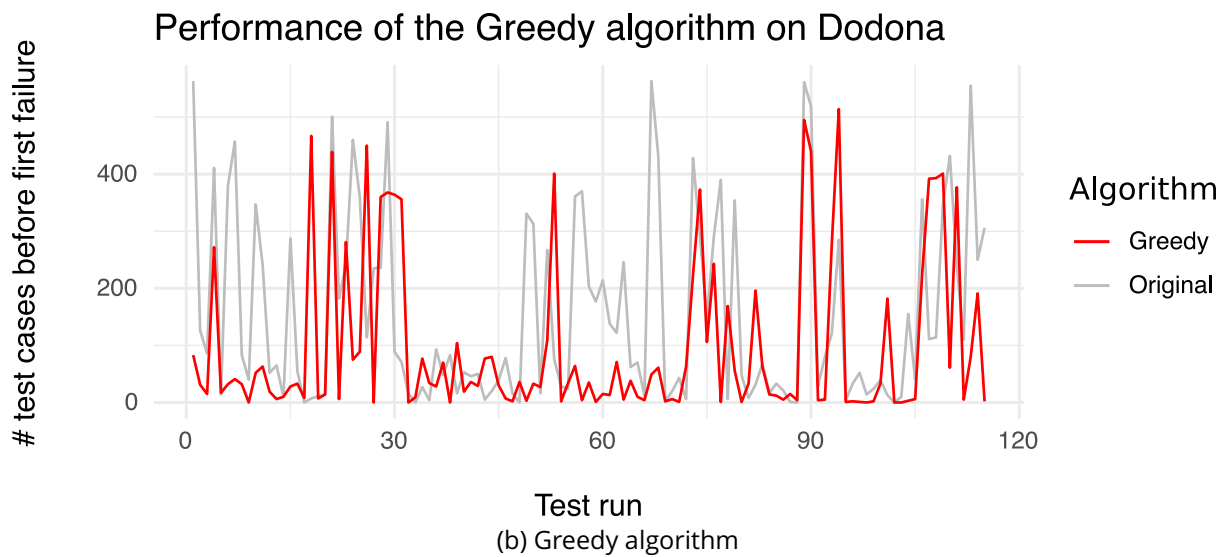
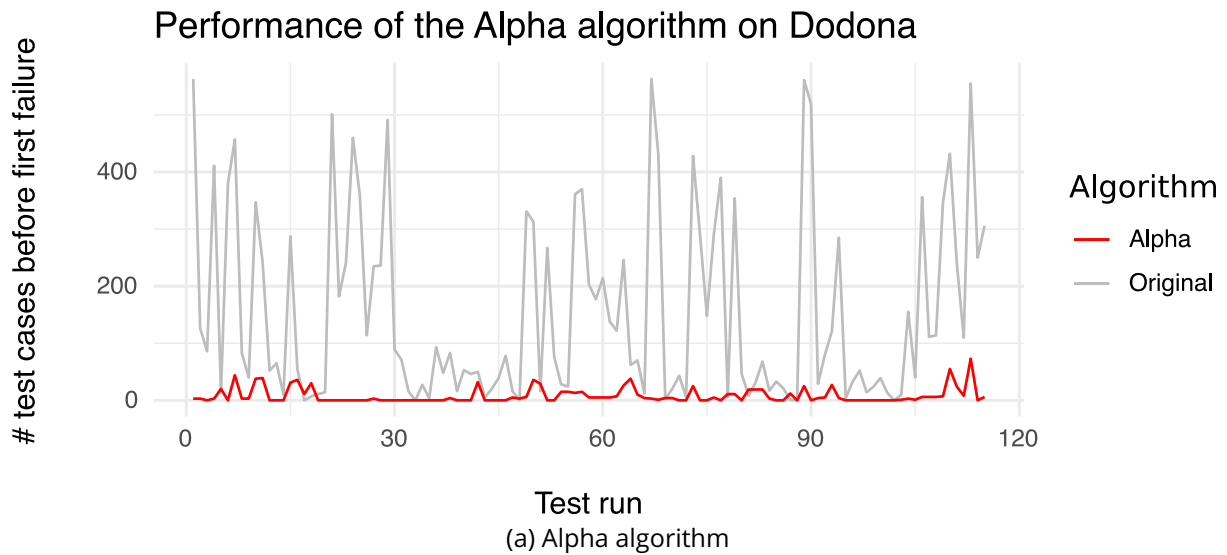
After executing the 120 failed test runs, the log files have been inspected. These log files have revealed that an error in the configuration was the actual culprit of five failed test runs, rather than a failed test case. These test runs have therefore been omitted from the results because the test suite did not execute. Since configuration-related problems require in-depth contextual information about the project, we cannot automatically predict these.

Table 5.2 contains the amount of executed test cases until we observe the first failure. These results indicate that every predictor is capable of performing at least one successful prediction. Furthermore, the maximum amount of executed test cases is lower than the original value, which means that every algorithm is a valid predictor. The data suggests that the Alpha algorithm and the HGS algorithm are the preferred predictors for the Dodona project. In contrast, the performance of the ROCKET algorithm is rather low.

Algorithm	Minimum	Mean	Median	Maximum
<i>Original</i>	0	155	78	563
Alpha	0	8	3	73
AffectedRandom	0	54	10	428
AllInOrder	0	119	82	460
AllRandom	0	90	27	473
GreedyCoverAffected	0	227	246	494
GreedyCoverAll	0	98	33	514
GreedyTimeAll	0	210	172	482
HGSAffected	0	61	10	511
HGSAll	0	124	54	507
ROCKET	0	210	170	482

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

The previous results have been visualised in Figure 5.5. These charts confirm the low accuracy of the ROCKET algorithm. The Alpha algorithm and the HGS algorithm offer the most accurate predictions, with the former algorithm being the most consistent. Observe the chart of the Greedy algorithm. This algorithm manages to very accurately predict some of the test runs while failing to predict others anywhere near, a behaviour which is specific to a greedy heuristic.



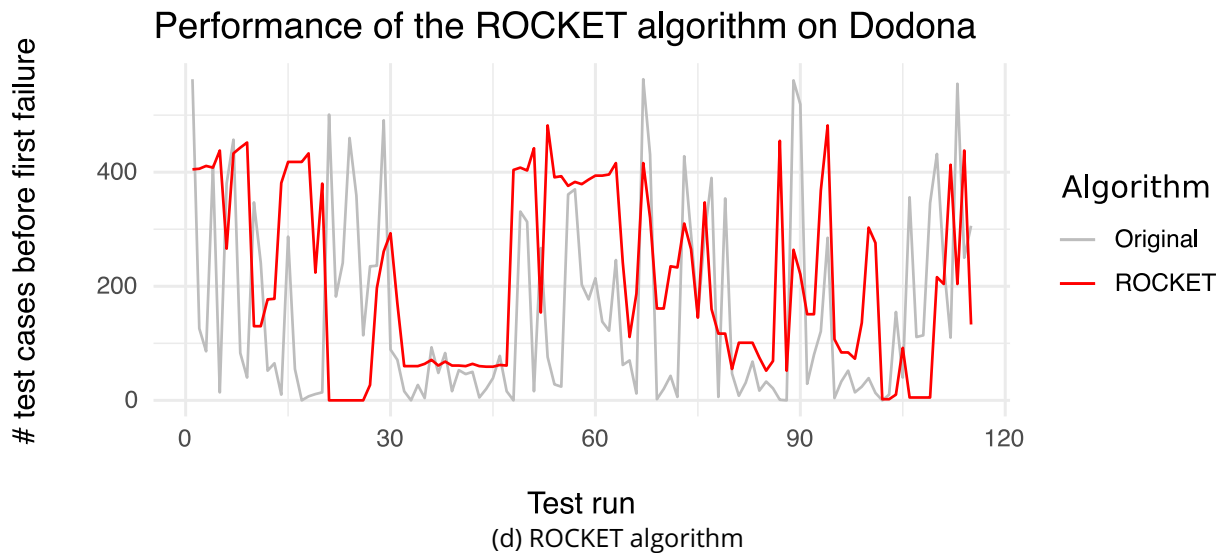


Figure 5.5: Prediction performance on the Dodona project.

The duration until the first observed failure is reported in Table 5.3. Observe that the previous table indicates that the ROCKET algorithm does not perform well, while this table suggests otherwise. We can explain this behaviour by examining the objective function of this algorithm. This function prioritises cases with a low execution time to be executed first.

Algorithm	Minimum	Mean	Median	Maximum
<i>Original</i>	0 s	135 s	123 s	380 s
Alpha	0 s	3 s	1 s	33 s
AffectedRandom	0 s	28 s	5 s	190 s
AllInOrder	0 s	82 s	71 s	270 s
AllRandom	0 s	43 s	11 s	270 s
GreedyCoverAffected	0 s	88 s	86 s	314 s
GreedyCoverAll	0 s	46 s	12 s	280 s
GreedyTimeAll	0 s	55 s	32 s	175 s
HGSAffected	0 s	35 s	6 s	356 s
HGSAII	0 s	75 s	34 s	377 s
ROCKET	0 s	54 s	32 s	175 s

Table 5.3: Duration until the first failure for the Dodona project.

5.4.5 RQ5: Integrate VeloCity with Stratego

The data collection phase has already proven that the Java agent is compatible with Stratego. Since VeloCity is not yet able to predict test cases which have been added in the current commit, we can only use 35 of the 54 failed test runs.

Similar to the previous test subject, Table 5.4 lists how many test cases have been executed before the first observed failure. The table only considers the four main algorithms, since the actual prediction performance was only secondary to this research question, and we have only analysed a small number of test runs. The results suggest that every algorithm except the ROCKET achieves a high prediction accuracy on this project.

Algorithm	Minimum	Mean	Median	Maximum
<i>Original</i>	0	68	2	278
Alpha	0	10	2	57
GreedyCoverAll	0	11	3	57
HGSAll	0	9	4	50
ROCKET	0	42	27	216

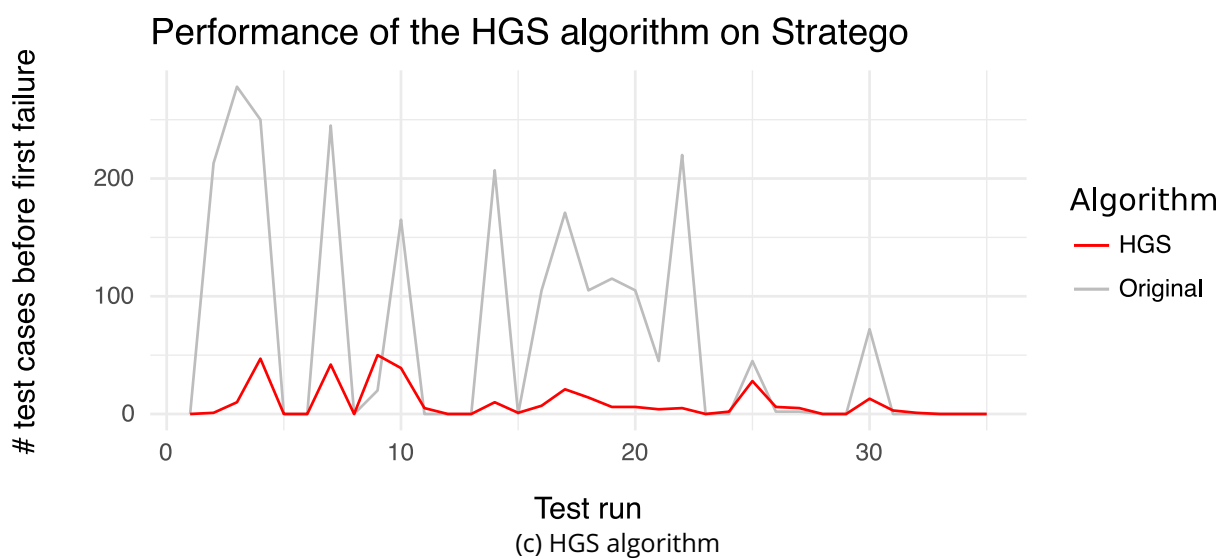
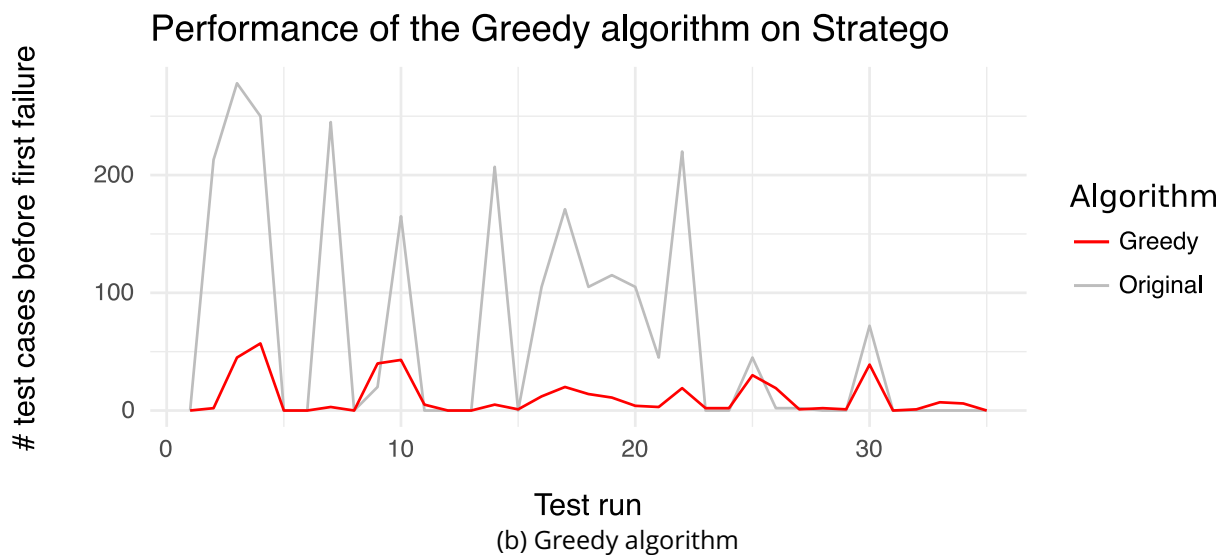
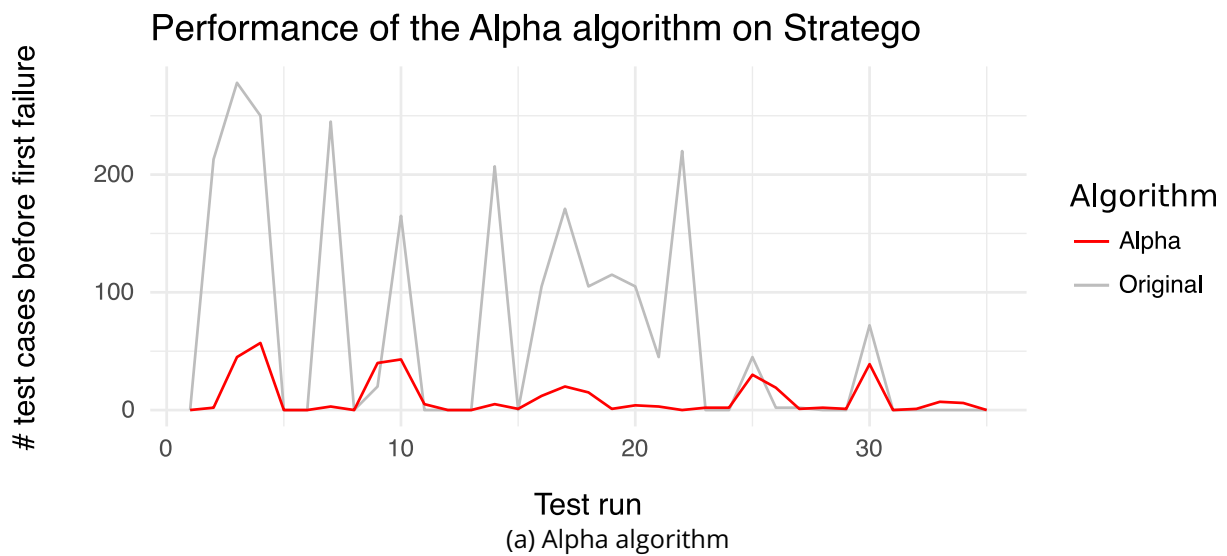
Table 5.4: Amount of executed test cases until the first failure.

Even though the performance of the ROCKET algorithm is suboptimal in the previous table, Table 5.5 does indicate that it outperforms every other algorithm time-wise. Notice that the predicted sequence of the HGS algorithm takes the longest to execute, while the previous table suggested a good prediction accuracy. The Alpha and Greedy algorithms seem very similar on both the amount of executed test cases, as well as the execution time.

Algorithm	Minimum	Mean	Median	Maximum
<i>Original</i>	0 s	62 s	8 s	233 s
Alpha	0 s	11 s	2 s	103 s
GreedyCoverAll	0 s	12 s	2 s	103 s
HGSAll	0 s	19 s	1 s	130 s
ROCKET	0 s	6 s	0 s	85 s

Table 5.5: Amount of executed test cases until the first failure.

Figure 5.6 further confirms the above statements. Notice the close resemblance of the charts of the Greedy algorithm and the Alpha algorithm, which indicates that a different failing test case is the cause of every test run failure. The ROCKET algorithm performs better on this project than on Dodona, yet not accurate.



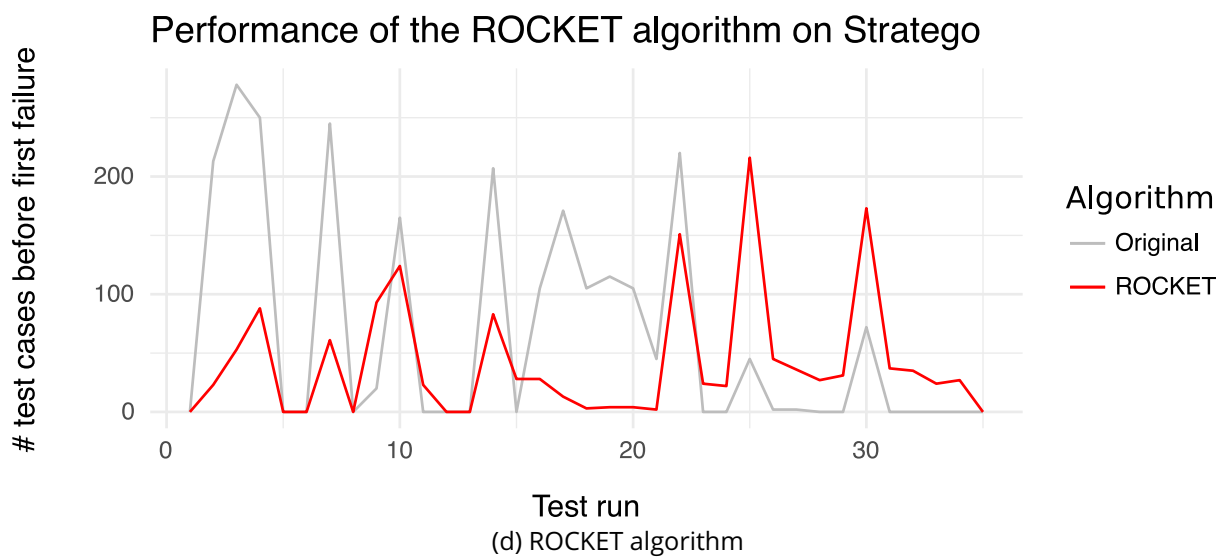


Figure 5.6: Prediction performance on the Stratego project.

Chapter 6

Conclusion

The main objective of this thesis was to study different approaches to optimise the test suite of a typical software project. Three approaches have been introduced to this extend: Test Suite Minimisation, Test Case Selection and Test Case Prioritisation. We have successfully implemented the latter approach using the VeloCity framework. Furthermore, this framework features the Alpha algorithm as a novel prioritisation algorithm. The performance of the Alpha algorithm has been evaluated, mainly on the UGent Dodona project. The results are promising, resulting in 95 % fewer executed test cases and 97 % less time spent waiting for the first test case to fail.

The second purpose of this thesis was to gain useful insights into the characteristics of a regular test suite, formalised into three research questions. The first question was to estimate the expected failure probability of a test run. To answer this question, we have analysed more than 28 million test runs on Travis-CI. This analysis has indicated that 18 % of those test runs have failed. Additionally, we have used this dataset to answer another question, which was to determine the typical duration of a test run. Statistical analysis has revealed that developers mainly use Travis-CI for small projects, with an average test suite duration of seven minutes. 0.20 % of the test suites take longer than one hour to execute, and some projects use mutation testing. The final question was to examine the probability of consecutive failing test runs. This probability was estimated at 52 % using a second Travis-CI dataset from the TravisTorrent project[5].

6.1 Future work

The proposed architecture currently features a Java agent, which supports the prediction of Gradle projects using ten available predictors. However, there is still room for improvements. The paragraphs below will suggest some ideas for possible enhancements.

6.1.1 Java agent

We can extend the functionality of the Java agent in multiple ways. Its current biggest weakness is the lack of support for parallel test case execution. To allow parallel testing, we must first solve a problem related to the scheduling process. Since the execution time of a test case can vary significantly, a coordination mechanism is required to schedule which test case should be executed on which thread. One possibility would be to consider the average execution time per test case, which we can obtain by examining prior runs. Alternatively, the scheduling can be performed at runtime using an existing inter-thread communication paradigm, such as message passing. Specific to the Java agent, implementing parallel execution requires us to modify the current `TestProcessor` to extend the `MaxNParallelTestClassProcessor` instead. A thread pool should ideally be used to diminish the overhead of restarting a new thread for every test case.

6.1.2 Predictions

We can make four different enhancements to the predictors.

For the first enhancement, the predictor should be able to discriminate between a unit test, an integration test or a regression test. Recall that the scope of a unit test is limited to a small fraction of the application code and that its execution time is usually low. Contrarily, an integration test and a regression test usually take much longer to execute and test multiple components of the application at once. The predictor should ideally make use of this distinction and assume that a failing unit test will almost certainly affect a regression or integration test as well. Consequently, the predictor should prioritise unit tests over other types of tests.

Secondly, the prediction algorithms currently take into account which source code lines have either been modified or removed to identify which test cases have been affected. Likewise, a change in the code of the test case itself should also consider that test case affected, as the change might have introduced a bug as well.

A third possible improvement would be to examine the performance of combining multiple prediction algorithms. Currently, the algorithms operate independently from each other, but there might be hidden potential in combining the individual strengths of these algorithms dynamically at runtime. A simple implementation is possible by modifying the existing meta predictor. Instead of assigning a score to the entire prediction, we could combine several predictions using predefined weights from earlier predictions.

Finally, the predictors do not currently consider branch coverage in addition to statement coverage. Not every coverage tool is capable of accurately reporting which branches have been covered, therefore this has not been implemented. Branch coverage can alternatively be supported by instrumenting the source code and rewriting every conditional expression as separate if-statements.

6.1.3 Meta predictor

The current implementation of the meta predictor increments the score of the predictor if the prediction was above-average, and decreases the score otherwise. However, a possible problem with this approach is that the nature of the source code might evolve and change as time progresses. As a result, it might take several test suite invocations for the meta predictor to prefer an alternative predictor. We can mitigate this effect if we would use a saturating counter (Figure 6.1) instead. This idea is also used in branch predictors of microprocessors and allows a more versatile meta predictor.

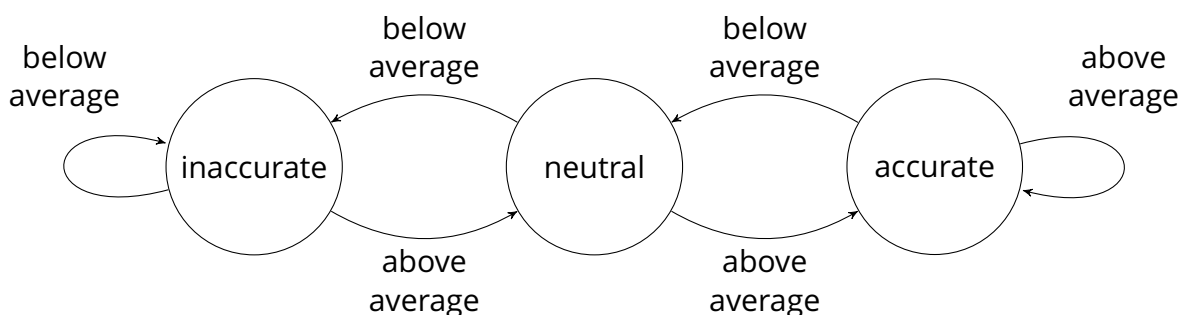


Figure 6.1: Saturating counter with three states.

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 we can apply every implemented algorithm to Test Suite Minimisation as well, we might extend the architecture to support this technique explicitly. Executing fewer test cases will result in even lower execution times.

Support for new programming languages and frameworks is possible by implementing a new agent. A naive implementation would be 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 (URN CST) Journal* 3 (Sept. 2019), pp. 1–6. DOI: [10.26685/urncst.152](https://doi.org/10.26685/urncst.152).
- [3] Beck. *Test Driven Development: By Example*. USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 0321146530.
- [4] Kent Beck et al. *Manifesto for Agile Software Development*. 2001. URL: <https://www.agilemanifesto.org/>.
- [5] 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.
- [6] 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>.
- [7] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844.
- [8] Michael Cusumano, Akindutire Michael, and Stanley Smith. "Beyond the waterfall : software development at Microsoft". In: (Feb. 1995).
- [9] Charles-Axel Dein. *dein.fr*. Sept. 2019. URL: <https://www.dein.fr/2019-09-06-test-coverage-only-matters-if-at-100-percent.html>.
- [10] Thomas Durieux et al. "An Analysis of 35+ Million Jobs of Travis CI". In: (2019). DOI: [10.1109/icsme.2019.00044](https://doi.org/10.1109/icsme.2019.00044). eprint: [arXiv:1904.09416](https://arxiv.org/abs/1904.09416).
- [11] *Features • GitHub Actions*. URL: <https://github.com/features/actions>.
- [12] 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.
- [13] *GitLab CI/CD*. URL: <https://docs.gitlab.com/ee/ci/>.
- [14] *GitLab Continuous Integration & Delivery*. URL: <https://about.gitlab.com/stages-devops-lifecycle/continuous-integration/>.
- [15] 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.

- [16] Standish Group et al. "CHAOS report 2015". In: *The Standish Group International* (2015). URL: https://www.standishgroup.com/sample_research_files/CHAOSReport2015-Final.pdf.
- [17] 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](https://doi.org/10.1145/152388.152391). URL: <https://doi.org/10.1145/152388.152391>.
- [18] 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.
- [19] "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](https://doi.org/10.1109/IEEESTD.2013.6588537).
- [20] "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](https://doi.org/10.1109/IEEESTD.2015.7106435).
- [21] "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](https://doi.org/10.1109/IEEESTD.2017.8016712).
- [22] 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.
- [23] 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>.
- [24] 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](https://doi.org/10.1109/ECBS.1997.581900).
- [25] Simon Maple. *Development Tools in Java: 2016 Landscape*. July 2016. URL: <https://www.jrebel.com/blog/java-tools-and-technologies-2016>.
- [26] 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.
- [27] Robert C. Martin and Micah Martin. *Agile Principles, Patterns, and Practices in C#* (Robert C. Martin). USA: Prentice Hall PTR, 2006. ISBN: 0131857258.

- [28] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. 3rd. Wiley Publishing, 2011. ISBN: 1118031962.
- [29] 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.
- [30] 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](https://doi.org/10.1007/978-3-030-35510-4_4).
- [31] 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](https://doi.org/10.1007/978-1-4757-5939-6_7). URL: https://doi.org/10.1007/978-1-4757-5939-6_7.
- [32] 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>.
- [33] 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/>.
- [34] Travis. *Travis CI - Test and Deploy Your Code with Confidence*. Feb. 2020. URL: <https://travis-ci.org>.
- [35] 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](https://doi.org/10.1145/1146238.1146240). URL: <https://doi.org/10.1145/1146238.1146240>.
- [36] 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](https://doi.org/10.1109/52.819971).
- [37] 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](https://doi.org/10.1002/stv.430). URL: <https://doi.org/10.1002/stv.430>.

List of Figures

1	Continuous Integration (simplified).	xvi
2	Test Case Prioritisation.	xvii
2.1	Improved Waterfall model by Royce	5
2.2	Statistics from Code coverage frameworks.	11
2.3	Process of Mutation Testing (based on [31]).	13
2.4	Success rate of Agile methodologies [16].	15
2.5	Development Life Cycle with Continuous Integration.	16
2.6	Logo of Jenkins CI (https://jenkins.io/).	17
2.7	Logo of GitHub Actions (https://github.com/features/actions).	17
2.8	Logo of GitLab CI (https://gitlab.com/).	18
2.9	Logo of Travis CI (https://travis-ci.com/).	18
3.1	Test Suite Minimisation.	21
3.2	Test Case Selection.	21
3.3	Test Case Prioritisation.	22
4.1	Sequence diagram of VeloCity.	33
5.1	MapReduce pipelines for Travis CI data.	45
5.2	Probability of test run failure.	47
5.3	Test run durations on Travis CI.	48
5.4	Consecutive test run failures on Travis CI.	48
5.5	Prediction performance on the Dodona project.	51
5.6	Prediction performance on the Stratego project.	54
6.1	Saturating counter with three states.	57

List of Tables

3.1	Example of the failure matrix MF	27
5.1	Characteristics of the test run durations in [10].	47
5.2	Amount of executed test cases until the first failure.	49
5.3	Duration until the first failure for the Dodona project.	51
5.4	Amount of executed test cases until the first failure.	52
5.5	Amount of executed test cases until the first failure.	52

List of Examples

2.1	Java unit test in JUnit.	6
2.2	Java integration test in JUnit.	7
2.3	Java regression test in JUnit.	8
2.4	Irrelevant statement coverage in C.	9
4.1	Minimal Gradle buildsript.	34
4.2	Output of the affected-command.	41
4.3	Output of the durations-command.	41
4.4	Output of the failures-command.	41
4.5	Output of the predict-command.	42
4.6	Output of the predictions-command.	42
4.7	Output of the scores-command.	42
A.1	TravisTorrent query: Find the amount of failed runs.	67
A.2	TravisTorrent query: Find the probability of consecutive failures.	67

Appendices

Appendix A

TravisTorrent queries

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

Example A.1: TravisTorrent query: Find the amount of failed runs.

```
1 (  
2   SELECT gh_build_started_at, true as failed  
3   FROM `travistorrent`  
4   WHERE  
5     tr_build_id IN (  
6       SELECT DISTINCT(tr_prev_build)  
7       FROM `travistorrent`  
8       WHERE tr_log_bool_tests_ran=true AND  
9         tr_log_bool_tests_failed=true  
10    )  
11   AND gh_build_started_at IS NOT null AND  
12     tr_log_bool_tests_failed=true  
13 ) UNION ALL (  
14   SELECT gh_build_started_at, false as failed  
15   FROM `travistorrent`  
16   WHERE tr_build_id IN (  
17     SELECT DISTINCT(tr_prev_build)  
18     FROM `travistorrent`  
19     WHERE tr_log_bool_tests_ran=true AND tr_log_bool_tests_failed=  
20       false  
21   )  
22   AND gh_build_started_at IS NOT null AND tr_log_bool_tests_failed  
23     =true  
24 )
```

Example A.2: TravisTorrent query: Find the probability of consecutive failures.