# Test Generation Techniques for Combinatorial Geospatial Computations

Comparing test techniques in an applied domain

Master's Thesis in Computer Science and Engineering

Marcus Schagerberg
Jakob Windt

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Master's Thesis 2025

# Test Generation Techniques for Combinatorial Geospatial Computations

Comparing test techniques in an applied domain

Marcus Schagerberg
Jakob Windt

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Test Generation Techniques for Combinatorial Geospatial Computations
Marcus Schagerberg and Jakob Windt

© Marcus Schagerberg and Jakob Windt, 2025.

Supervisor: Robert Feldt, Department of Computer Science and Engineering
Examiner: Francisco Gomes de Oliveira Neto, Department of Computer Science and Engineering
Company Involved: Carmenta
Company Contact: Jesper Holm

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2025

Marcus Schagerberg and Jakob Windt
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg


# Abstract

Software in the industry is usually tested using old proven techniques. State-of-the-art software testing technologies are often reserved for research. This study aims to investigate the advantages and drawbacks of three test generation techniques in an applied domain, where an entire tool may be ruled out due to incompatibilities with the software and its stack. They are tested on a software development kit for real-time, geospatial applications that is written in C++. The study performs a quantitative comparison using mutation testing-inspired fault injection, and a qualitative comparison focused on the applicability, ease of use and usefulness during its entire development and integration process. Property-based testing, model-based testing and combinatorial interaction testing are implemented in the software, and the learnings are presented in this study.


Keywords: Software testing, test generation, geospatial, test technique evaluation, industry

# Acknowledgements

We would like to thank our industry supervisor Jesper Holm for his help during the study. A special thanks to the developers at Carmenta, mainly Marcus Lundberg, who guided and assisted us with challenges related to the software. We would also like to take this opportunity to thank Patrik Ellrén for making the study happen, Kristian Samuelsson for his very welcoming attitude and the rest of the employees at Carmenta for hosting us. Finally, we would like to express our gratitude to our supervisor Robert Feldt, and our examiner in practice Ranim Khojah for their guidance.

Marcus Schagerberg and Jakob Windt, Gothenburg, June 2025

# Contents

**List of Figures** **xiii**

**List of Tables** **xv**

**1 Introduction** **1**
1.1 Problem Description . . . 1
1.2 Purpose of the Study . . . 2
1.3 Research Questions . . . 2
1.4 Significance of the Study . . . 2
1.5 Scope Delimitations . . . 3

**2 Background** **5**
2.1 Software under Test . . . 5
2.1.1 Context from Carmenta . . . 5
2.1.2 Carmenta Engine . . . 5
2.1.3 SDK Interfaces . . . 6
2.1.4 Operators . . . 7
2.2 Testing Structure at Carmenta . . . 7
2.2.1 Image Tests . . . 8
2.2.2 Integration Tests . . . 9
2.2.3 Release Testing . . . 9
2.3 Software Testing Strategies . . . 9
2.3.1 Mutation Testing . . . 9
2.3.2 Fault Injection Testing . . . 10
2.3.2.1 Fault Categories . . . 11
2.3.3 Combinatorial Interaction Testing . . . 12
2.3.4 Model-based Testing . . . 12
2.3.5 Property-based Testing . . . 13
2.4 Technologies under Investigation . . . 14
2.4.1 Hypothesis . . . 14
2.4.2 PICT . . . 14

**3 Related Work** **17**
3.1 Mutation Testing on C++ Software . . . 17
3.2 Test Generation Methods . . . 17

3.3 SDK Testing . . . . . . . . . . 18

**4 Methods 21**
4.1 Evaluating the results . . . . . . . . . . 21
4.2 Technique Selection . . . . . . . . . . 21
4.2.1 Choice of Data Collection Method . . . . . . . . . . 22
4.2.2 Interviews . . . . . . . . . . 23
4.2.3 Focus Group . . . . . . . . . . 24
4.2.4 Anonymization . . . . . . . . . . 24
4.3 Technique Implementation . . . . . . . . . . 25
4.3.1 Choosing the tools . . . . . . . . . . 25
4.3.2 Compatibility with the Environment . . . . . . . . . . 26
4.3.3 Implementation phase . . . . . . . . . . 26
4.4 Scenarios . . . . . . . . . . 27
4.5 Technique Evaluation . . . . . . . . . . 28
4.5.1 Why Fault Injection was Chosen over Mutation Testing . . . . 29
4.5.2 Fault Injection . . . . . . . . . . 30
4.6 Implementation of the Fault Injection Tool . . . . . . . . . . 30

**5 Results 33**
5.1 Results from the Initial Interview . . . . . . . . . . 33
5.2 Determining Relevant Testing Techniques . . . . . . . . . . 33
5.2.1 Justifications of Conclusions . . . . . . . . . . 35
5.2.2 Decisions . . . . . . . . . . 36
5.3 Comparison of Test Generation Techniques . . . . . . . . . . 37
5.3.1 Fault Injection . . . . . . . . . . 37
5.3.1.1 Missing Test Suites . . . . . . . . . . 37
5.3.1.2 Caught Faults . . . . . . . . . . 38
5.3.1.3 Fault Detection by Category . . . . . . . . . . 39
5.3.1.4 Fault Detection Frequency . . . . . . . . . . 39
5.3.1.5 Performance Metrics . . . . . . . . . . 40
5.3.2 Fault Injection using Cache . . . . . . . . . . 41
5.3.3 Results from the Focus Group Session . . . . . . . . . . 43
5.3.4 Reflections during Implementation . . . . . . . . . . 45
5.3.4.1 Ease of Use . . . . . . . . . . 45
5.3.4.2 Applicability in Different Contexts . . . . . . . . . . 46
5.3.4.3 Relevance . . . . . . . . . . 46

**6 Discussion 47**
6.1 Answering the Research Questions . . . . . . . . . . 47
6.1.1 Research Question 1 . . . . . . . . . . 47
6.1.2 Research Question 2 . . . . . . . . . . 48
6.1.3 Research Question 3 . . . . . . . . . . 49
6.2 Flaky Tests . . . . . . . . . . 50
6.3 Drawbacks of the Techniques . . . . . . . . . . 51
6.4 Impact of Test Generation Cache . . . . . . . . . . 52
6.5 Recommendations . . . . . . . . . . 53

6.5.1 Regression Testing . . . 53
6.5.2 Nightly Testing . . . 53
6.5.3 Release Testing . . . 54
6.6 Threats to Validity . . . 54
6.6.1 Internal Validity . . . 54
6.6.2 External Validity . . . 55
6.7 Future Work . . . 55
6.7.1 Mutation Testing . . . 55
6.7.2 Improvements to the Fault Injection Tool . . . 56
6.7.3 Combinations of Techniques . . . 56
6.7.4 Implementation at Another Company . . . 56

**7 Conclusion** **57**

**Bibliography** **59**

# List of Figures


2.1 Carmenta Explorer, Reprinted from Carmenta[2] . . . . . . . . . . . . 6
2.2 Carmenta Studio, Reprinted from Carmenta[2] . . . . . . . . . . . . . 6
2.3 A typical operator chain. Note that the data travels from right to left. Reprinted from Carmenta[3]. . . . . . . . . . . . . . . . . . . . . . . 7
2.4 Generated Image . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
2.5 Reference Image . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
2.6 Zoomed in on section in Generated Image . . . . . . . . . . . . . . . . 8
2.7 Zoomed in on section in Reference Image . . . . . . . . . . . . . . . . 8
2.8 Model-based Testing flow-chart. . . . . . . . . . . . . . . . . . . . . . 13

4.1 Technique and tool selection process . . . . . . . . . . . . . . . . . . 22
4.2 SplitFilterFunnel Operator Chain . . . . . . . . . . . . . . . . . . . . 28
4.3 ThinInsideBuffer Operator Chain . . . . . . . . . . . . . . . . . . . . 28
4.4 Fault injection tool process . . . . . . . . . . . . . . . . . . . . . . . 32

5.1 How experienced are you with software testing in general? . . . . . . 34
5.2 How experienced are you with software testing techniques? . . . . . . 34
5.3 How much do you trust the testing of the SDK? . . . . . . . . . . . . 35
5.4 Number of caught faults per test method . . . . . . . . . . . . . . . . 39
5.5 Elapsed time per test suite, with and without outliers. The medians, in orange, are presented . . . . . . . . . . . . . . . . . . . . . . . . . . 41
5.6 Number of caught faults per test method, using cache . . . . . . . . . 42
5.7 Elapsed time per test suite with cache, with and without outliers. The medians, in orange, are presented . . . . . . . . . . . . . . . . . . 43

6.1 PBT failing connect line tests. The asterisk marks the start of the line. 52

# List of Tables


4.1 Interview questions regarding general information . . . . . . . . . . . 24
4.2 General information of interviewees . . . . . . . . . . . . . . . . . . . . 24
4.3 Interview questions regarding software testing knowledge . . . . . . . 25
4.4 Interview questions regarding the current testing of the SDK . . . . . 25

5.1 Faults not tested by techniques . . . . . . . . . . . . . . . . . . . . . . 38
5.2 The number of faults caught by each test generation technique . . . . 38
5.3 Fault detection count by category . . . . . . . . . . . . . . . . . . . . . 39
5.4 Fault detection frequency from all five runs . . . . . . . . . . . . . . . 40
5.5 The total time to run the custom fault injection tool . . . . . . . . . 40
5.6 Fault detection count by category, using cache . . . . . . . . . . . . . 41
5.7 Fault detection frequency for all five runs, using cache . . . . . . . . . 42
5.8 Positive and negative feedback for each technique, from the focus group session . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44

6.1 Example input for PICT . . . . . . . . . . . . . . . . . . . . . . . . . . 47
6.2 Output generated by PICT from the input data in table 6.1 . . . . . 48

# Glossary

**API**: Application Programming Interface is a connection between computers or computer programs.

**BST**: Binary Search Tree is a data structure used to store data. It contains a number of nodes connected in a tree-like structure.

**CI**: Continuous Integration

**CIT**: Combinatorial Interaction Testing is a black box sampling technique.

**GUI**: Graphical User Interface is a type of user interface where users can interact with graphical/visual indicators.

**HPC**: High-Performance Computing is the practice of combining computational resources to solve complex problems.

**LoC**: Lines of Code.

**MBT**: Model-based Testing is a test generation method, where tests are derived from an abstract model of the software-under-test.

**MSVC**: Microsoft Visual C++ is a C++ compiler developed by Microsoft.

**PBT**: Property-based Testing is a test generation method, where tests are generated from a property that should hold.

**SUT**: Software Under Test refers to the system under evaluation by testers.

**SDK**: Software Development Kit is a collection of tools within an installable software package.

**UAS**: Unmanned Aircraft System, also commonly known as a drone.

# 1

# Introduction

Test case generation is a useful tool when testing large and complex systems. However, there are many methods for generating tests. Carmenta is a company working with geospatial information who offer a Software Development Kit, SDK, that is highly customizable, similar to a toolbox. The components of this SDK can be combined in different ways by customers to develop a product tailored to them. An example of a customer product would be taking a terrain map, a start and end point as input, and calculating the fastest route between them based on the characteristics of the vehicle and terrain. The characteristics are for example maximum capable incline and the type of surface. Testing all combinations and configurations manually is daunting, if not impossible, which is why they are interested in finding strategies to improve their test suite with the use of generated test cases.

## 1.1 Problem Description

A software test is when a "program is executed with desired input(s) and the output(s) is/are observed accordingly. The observed output(s) is/are compared with expected output(s). If both are same, then the program is said to be correct" [24, p. 4]. A simple statement that fits most scenarios, but what happens when the desired inputs are unknown, or in this case too many to test?

Carmenta operates in the geospatial field, with real-time computations related to the real world. This distinguishes the study, as common testing goals such as resource efficiency is not the main focus. The product is to be used by humans, where real life human consequences could be the end result of a faulty code base. Therefore, it is important that the tests are directed to cover the relevant geospatial relationships in order to create a trustworthy product. The human element, and the domain knowledge, therefore play an important part in designing the testing methodologies to target correctness, safety and error handling. Geospatial computations are an approximate environment, where input and output are often inexact. As an example, the input may be coordinates in the world or a terrain map, which naturally contain fluctuations and deviations. It is closely connected to the real world, and therefore shares properties with a natural setting. In many cases, it is not possible to make clear specifications, as the input and output can be difficult to narrow down, meaning specified oracles useful in testing can be hard to come by. The geospatial domain

will thus have an effect on the possible testing methodologies and their effectiveness.

## 1.2 Purpose of the Study

The purpose of the study is to trial and assess different test generation techniques on the Carmenta SDK. With the vast amount of operator combinations that a customer can create, there are a lot of potential problems that can occur. By using the current test suite as a baseline, the objective is to evaluate test generation techniques on the SDK. This field experiment will provide valuable data regarding the performance of different test generation techniques which will both benefit Carmenta and researchers. Based on the results, we aim to provide recommendations for real-time geospatial system testing.

## 1.3 Research Questions

**RQ 1:** *What testing strategies and/or tools would be beneficial in the geospatial technology field with a highly configurable product?* Compare the current testing efforts at Carmenta with contemporary research to find recommendations and potential improvements to their testing methodology.

**RQ 2:** *Which test generation methods are useful for testing operators?* Investigate how 3-5 different methods compare and how useful the developers consider them for a specific set of operators. Usefulness in this case is determined by the developers.

**RQ 3:** *How well do the different test generation methods withstand manually introduced faults in operators?* If faults are introduced into an operator, how do different methods compare in which types of faults they catch and miss?

## 1.4 Significance of the Study

The study aims to explore test case generation in a setting where it has no prior exposure. Retrofitting has different challenges than extending a platform that had it in mind, and may therefore lead to techniques that have been deemed successful in research not being effective or even applicable in our environment. It is a study comparing test generation techniques in an applied setting, within the geospatial domain, aiming to answer concrete questions and problems arising in its context, rather than an objective and abstract direct comparison of the techniques. This context separates the study from other research, and this uniqueness is in turn what can make it especially useful for other projects in a similar domain. The findings of the study will be beneficial to the awareness and development at Carmenta, but can also highlight the role of a long company history, where methods and patterns were developed during a period where many test generation tools were absent, and how these can be reinvented.

## 1.5 Scope Delimitations

The project will be performed and evaluated within a single company, which may have implications for external validity. The company specializes in the field and has been working in the geospatial domain for 40 years[1]. However, the software solution and testing methodology in place may not reflect the domain as a whole. The results will also be tied to the context of the study, and may therefore not be applicable in some external cases.

Cost efficiency will not be taken into consideration throughout the course of the project. Both within research and in practice, cost could be a major factor depending on, for example, time and resources. For this project, it would add additional complexity and introduce another factor of potential bias.

Due to the confidentiality of the software source code, techniques and tools using large language models or other forms of generative AI will not be taken into consideration throughout the project.

Commercial tools that require licenses or other forms of paid tools will not be considered. However, for many techniques there are free or open source alternatives. This limitation will therefore most often not rule out entire techniques, only certain tools. Note that we are trying to compare the techniques, not the tools.

[1] https://carmenta.com/about

# 2

# Background

This chapter aims to give the necessary background information related to the project. It will cover areas such as Carmenta's software and testing structure and then explore potential test generation tools and techniques.

## 2.1 Software under Test

The software under test, SUT, chosen for this study is based on the Carmenta Engine SDK developed by the company Carmenta, based in Gothenburg, Sweden.

### 2.1.1 Context from Carmenta

Carmenta is a company working with geospatial information who offer a Software Development Kit that is highly customizable, similar to a toolbox. The components of this SDK can be combined in different ways by customers to develop a product tailored to them. The SUT being an SDK plays an important role in the study, and is a distinction from most other software where you can trace the dependencies and execution orders and use as context when writing tests. With an SDK, you do not know exactly how the component will be used, or in which context.

### 2.1.2 Carmenta Engine

Carmenta offers a few different products, each with differences in the code base and testing. They consist of Carmenta Engine, Carmenta Server, Carmenta Map Builder and Carmenta UAS Services[1].

Their main product is Carmenta Engine, which is the product that will act as the main SUT during this study. It is an SDK that can be used for real-time visualization and analysis of geospatial information[2]. Developers can use the SDK to create interactive 2D and 3D applications on Windows, Linux and Android. Within the engine there is Carmenta Explorer and Carmenta Studio, shown in figure 2.1 and 2.2 respectively. Carmenta Explorer acts as a graphical user interface, or GUI, for the user designed applications where map configurations can be imported and interacted

[1] https://carmenta.com/products
[2] https://carmenta.com/products/carmenta-engine

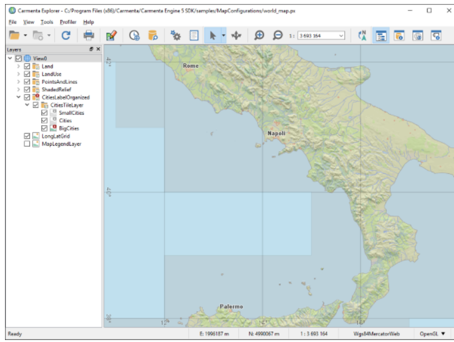


**Figure 2.1:** Carmenta Explorer, Reprinted from Carmenta[2]

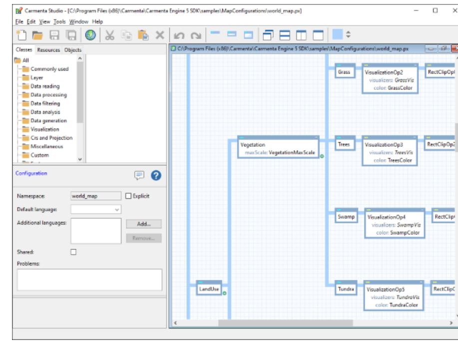


**Figure 2.2:** Carmenta Studio, Reprinted from Carmenta[2]

with in a user friendly environment. Carmenta Studio provides the functionality to interact with the map configuration in a code-less environment. The user can add, remove and modify operators and other aspects of the configuration and instantly see the changes within Carmenta Explorer.

The engine uses a dataflow model consisting of a sequence of processing steps, modifying the object representing a geographical position with its own attributes. The steps in the dataflow model are known as Layers consisting of the core building blocks of the toolkit known as Operators. At the time of writing, there are over 100 operators for the user to combine, modifying the geographical information to the user's configuration[3].

### 2.1.3 SDK Interfaces

The focus of this thesis is the SDK, which is written in C++. This presents a challenge, as the testing community and libraries for C++ are limited. However, the SDK also provides APIs, Application Programming Interfaces, which lets customers develop their products in other languages than C++ while still being able to use the SDK. The Java and Python APIs are of special interest for our purposes, as these have the most tools available and the best documentation for the testing techniques to be examined. The APIs allow us to write tests in either Java or Python, using available tools and libraries for those languages, while still testing the C++ code base.

This solution is not as plug-and-play as using the tools natively, however it is more flexible as we can leverage the greater supply available for Java or Python. After comparing them, the decision was to use Python since tools existed for all techniques under investigation. Our longer prior experience and familiarity with Python was also a contributing factor. Combined with the fact that Python is an interpreted language, makes it suitable for exploratory development since it is a quick and easy process to try something before deciding whether it will be useful for the work.

[3]https://docs.carmenta.com/?page=operator

### 2.1.4 Operators

As previously mentioned, the SDK contains software building blocks known as operators that are used to modify features. A feature represents an object with a geographical position and a set of attributes[4]. By combining and chaining operators, a user can tailor the functionality of the SDK to their specific use case.

An operator chain is usually built with four components. A chain typically starts with a ReadOperator that is used to read data from a dataset. Datasets store features and come in different variations depending on the data source. Following the ReadOperator, a more use case-specific operator is used, such as a TerrainRouteOperator which calculates the fastest route a vehicle can take between a set of waypoints. Finally, a VisualizationOperator connects visualizers to the features, and then a view component displays the visualizers within Carmenta Explorer. This simple yet powerful versatility of the SDK is what allows users to create their customized tool for their specific purpose.

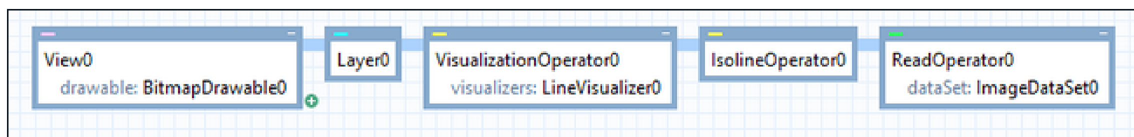



**Figure 2.3:** A typical operator chain. Note that the data travels from right to left. Reprinted from Carmenta[3].

An example can be seen in figure 2.3 that shows an operator chain in Carmenta Studio.

## 2.2 Testing Structure at Carmenta

Carmenta uses a number of different tests to validate the functionality of their products. The first category is product testing, where testing strategies differ between products. Some products do not have automated tests, as an example. Secondly, the main division is between automated and manual tests. The automated test suites run nightly together with a nightly build, testing any recent commits. Manual tests come from test plans that describe the overall goals of what needs to be tested. From the plan, a test specification is created, detailing how to perform the testing. During testing, testers follow the procedure in the specification and fill in a test report with their findings. Each result is compared against the expected result in the specification. Some deviations are expected, mainly for the image tests as the produced images may differ depending on the graphics hardware or software used. If there are unexpected deviations from the expected results, they are noted and an issue is created in order to fix the problem.

[4]https://docs.carmenta.com/?page=feature

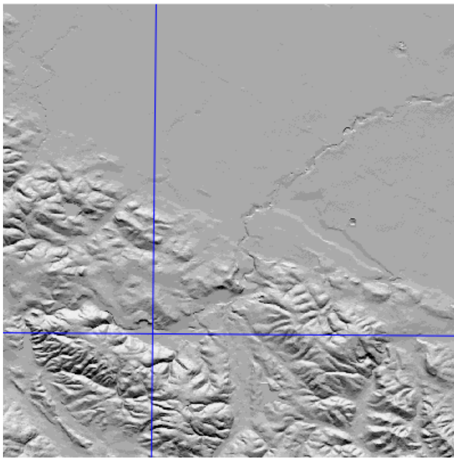


**Figure 2.4:** Generated Image

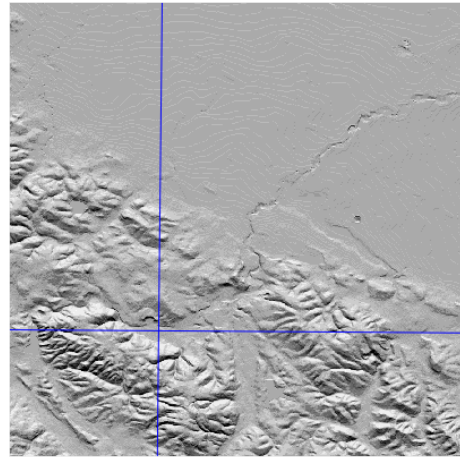


**Figure 2.5:** Reference Image

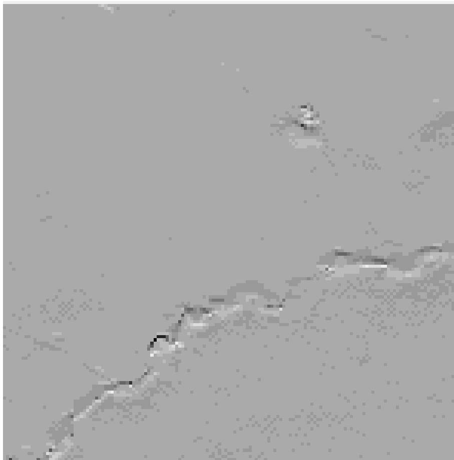


**Figure 2.6:** Zoomed in on section in Generated Image

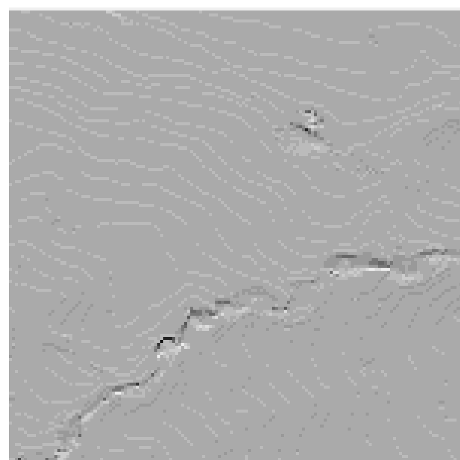


**Figure 2.7:** Zoomed in on section in Reference Image

### 2.2.1 Image Tests

The most encompassing type of automated software test at Carmenta is image testing. Its main flaw is the minor inconsistencies with image generation and image comparisons. The generated images are compared to a correct reference image, and scored with a percentage depending on the ratio of matching pixels. If the percentage is greater than the set acceptable score, the test case passes.

Depending on factors such as the type of graphics card used, image renders can vary, resulting in false positives. Even if a render is not identical to the one expected in a test, the minor differences can be negligible for real-world usage. This uncertainty with the image tests results in developers spending time manually reviewing failed test cases, to check if there are false positives.

Figures 2.4 and 2.5 represent a failed image test. According to the test runner, there are a total of 697 pixels that are different between the two images. Figure 2.5, the reference image, has thin white lines throughout the image representing

contours, while the generated image does not. This difference can be better seen in the zoomed in images in figure 2.6 and 2.7.

However, the focus of the test are the blue lines marking a location on the map. These two blue lines are identical, resulting in a test that should pass, but does not due to the pixel difference.

### 2.2.2 Integration Tests

Integration tests make up for another major part of the test suite at Carmenta. These tests are mostly written in C++ for the kernel, but also in languages such as Java, Python and C# for the APIs of the SDK. As the name suggests, an integration test should test multiple components within a system as a group to compliment other types of tests such as unit tests and system tests.

### 2.2.3 Release Testing

The final type of testing conducted at Carmenta is release testing. This is the final testing before a release, which has several stages. The automatic tests have already been completed, so release testing is manual testing of the software on multiple computers running different hardware and software combinations. The tests range from installation and setup tests, to running the software and verifying that it all works as expected. During release testing, some issues may arise for the first release candidate, which are logged and fixed if possible for the next release candidate. Then, retesting takes place, where the new version is tested to ensure that the fixes worked. Once the release testing is finished, a new software version is ready to be released to the customers.

## 2.3 Software Testing Strategies

The following section will explain the concepts mutation testing and fault injection that are relevant to this study. Thereafter, the investigated test generation techniques will be presented.

### 2.3.1 Mutation Testing

Although mutation testing is not used in this study, a very closely related method was instead used, which will be explained in section 2.3.2. There are also plenty of references to mutation testing, meaning the concept is necessary to understand when reading this thesis.

Introducing faults to validate test suites is known in software testing as mutation testing. The core strategy is generating mutants - slightly modified versions of the source code - and then measuring whether the test suite catches these mutants or not. The mutation score is the percentage of caught mutants [24, p. 216], and is the metric that is used to quantify each test generation method. Mutation testing

```
bool isBelowThreshold(int a) {
    if (a < 5)
        return true;
    return false;
}
```

**Code 1:** Original code

```
bool isBelowThreshold(int a) {
    if (a > 5)
        return true;
    return false;
}
```

**Code 2:** Mutated code

is used to determine the effectiveness of a test suite [24, p. 212], which enables comparisons of test suites generated through different test generation methods. The underlying idea is that, if the test suite is sensitive to source code changes, there is a good chance it captures its behaviour. As an example, exchanging a less than operator to a more than operator, as in Code 1 and 2, will alter the behaviour of the SUT, and is therefore expected to be caught by some test.

In this case, the change is obvious in that it will change the behaviour of the SUT and cause an incorrect return value. However, an issue with mutation testing is that there can be false positives. If the change does not alter the behaviour, a test is not needed to cover the mutation. For large systems, the number of false positives can increase to scales that makes it unusable.

Another issue with mutation testing is that the source code needs to be changed a lot, to generate many mutants. In compiled languages, like C++, the build process to compile the software can take time. Mutation testing of these types of systems would not be viable if the code had to be rebuilt for each mutant. The mutation testing tool Mull [8] solves this by injecting conditional flags into the bitcode, that allow the tool to enable mutations during the test runtime. By bundling all mutations into a single build, the tool performs well even on large systems.

### 2.3.2 Fault Injection Testing

Similarly to mutation testing, fault injection tests software by introducing faults into the SUT, either during compile-time by modifying the code or during runtime for example by responding with an incorrect API response. The main goal of fault injection testing is to test the resilience and ability to recover from failures. It is most commonly used for testing safety-critical software, which is frequent in the automotive industry.

Chaos Monkey[5] is a famous fault injection tool developed by Netflix. Netflix also

[5]https://github.com/Netflix/chaosmonkey

coined the term Chaos Engineering, which is the practice of intentionally causing failures in the software to test its resilience. Chaos Monkey randomly terminates procedures within the system causing chaos, hence its name. By randomly terminating parts of the software, the behaviour of the system during unusual operating conditions can be validated. Chaos engineering often uses fault injection to cause the failures.

#### 2.3.2.1 Fault Categories

Software faults can be categorized into different types. Several categorizations exist in the research space, of which one is presented by Grottke and Trivedi [12]. They categorize faults into two distinct categories with sub-types. These are presented below.

1. **Bohrbug**: Easily detected and reproduced fault
2. **Mandelbug**: Complex fault, either caused by interactions of conditions or with a time lag between activation and occurrence
    (a) **Heisenbug**: Elusive fault where attempting to investigate it causes it to disappear or take a different form
    (b) **Aging-related bug**: Runtime bug that causes failures after a while due to error accumulation

However, this categorization has two main flaws in the case of our study. The first is that it is too coarse, with only two categories. We need to categorize our faults into finer buckets with more precision, to see whether there are any trends in fault detections among the categories. The second issue is that it does not fit well with mutation testing, as these faults are simple and will therefore almost always result in Bohrbugs. Generating Mandelbugs from mutation testing is a result of chance, and is improbable to occur.

Another categorization is the one presented by Catolino et al. [4], which can be seen below.

1. **Configuration issue**: Bugs concerned with building configuration files
2. **Network issue**: Bugs related to having connection and/or server issues due to network problems
3. **Database-related issue**: Bugs that can appear between the main application and its database
4. **GUI-related issue**: Bugs related to the GUI. For example, screen layouts and padding issues
5. **Performance issue**: Bugs that cause memory issues, energy leaks and/or methods with infinite recursion

6. **Permission/deprecation issue**: Either bugs caused by usage of deprecated method calls or APIs, or missing API permissions

7. **Security issue**: Bugs related to vulnerability issues

8. **Program anomaly issue**: Bugs introduced by developers when enhancing existing source code, relating to exceptions, problems with return values and unexpected crashes

9. **Test code-related issue**: Bugs that appear in test code

A similar issue to that of Grottke and Trivedi comes with the categorization done by Catolino et al. Despite more categories, most faults introduced in this study would still fall under a single category, being program anomaly issues. This is because operators are related to the logic of the core functionality. Therefore, categories such as database-related issues and network issues cannot be applied.

As a result, in section 4.5.2, faults will be categorized in an alternate way that is more specific to the study.

### 2.3.3 Combinatorial Interaction Testing

Combinatorial interaction testing, or CIT, is a technique for generating test cases that cover the input space to a defined strength [9]. Consider a system that has a number of parameters, each of which can take several different values. A combinatorial test of strength 1 would generate a test suite that ensures all values, for all parameters, are tested. However, the interesting use of CIT is for higher strengths. In addition to testing all possible inputs, a higher strength combinatorial test will generate a test suite with different combinations of input values, meaning interactions between different inputs can be uncovered.

As explained by Anand et al. [1], CIT is aimed at detecting faults caused by "the interactions of the combinations of inputs or configuration options". This is of particular interest in the context of this work, as the SDK is meant to be combined and configured in different ways according to the needs of the customer. Ensuring the correctness of the SDK under various configurations or inputs is therefore a useful addition to their test suite.

### 2.3.4 Model-based Testing

Model-based testing, or MBT, is a testing approach with the use of models. By designing an abstract model for the SUT, it focuses on the core functionality, instead of the small intricate details [27]. This way, the main key features of the SUT can be prioritized which eventually results in higher quality tests. Once the model is created, a model-based testing tool can be used for automatic test case generation, with each test being related to the specified model. This high-level approach requires less resources since it does not test every single detail of the SUT, resulting in a test suite that targets the critical functionality. However, a downside is the amount of

time required to create a model. Compared with other test generation techniques, this is the main barrier to entry when it comes to model-based testing. In conclusion, model-based testing has a lot of initial overhead, but once the model is developed, the result is a testing system with long-term accuracy and efficiency.

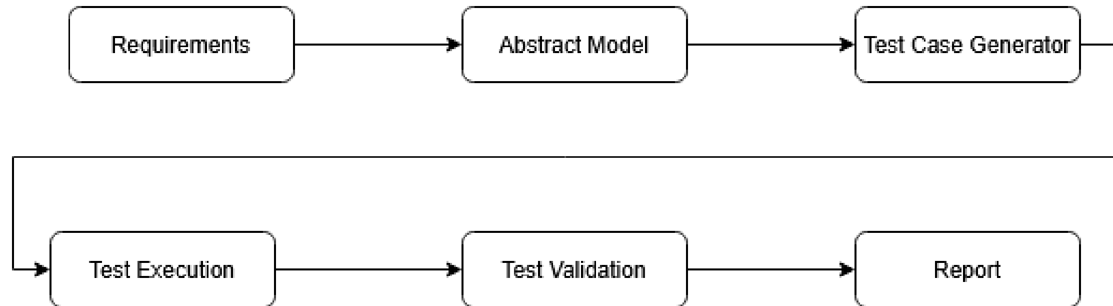



**Figure 2.8:** Model-based Testing flow-chart.

A flow chart depicting the steps to conduct model-based testing can be seen in figure 2.8. The first step of model-based testing is to create the abstract model based on the requirements or the test plan. From there a model-based testing tool can be used to generate and execute test cases. During the execution phase, checks are used to validate the results of the tool during runtime. Finally, a report is typically returned that contains the necessary information about the execution of the model-based testing tool.

### 2.3.5 Property-based Testing

The main idea of Property-based Testing, or PBT, is that each program has some specification for how it should behave, and those specifications can sometimes be formulated as properties of the program. An example, as explained by Goldstein et al. [11], is for a Binary Search Tree, BST, which is a data structure with an efficient search algorithm. A data point is stored as a node, where each node can have a left and right child. This creates a tree structure, with the data being stored in a semi-sorted order due to the rule that lesser values must be the left child of a parent node. All operations for modifying a BST should yield a new valid BST. This is a property of a BST, and can be tested using property-based testing. The example used by Goldstein et al. [11] is that given some integer to be stored in the tree, the insert operation of that integer should always return a valid BST. Using property-based testing, this can be specified. Then, using a generator that produces integers in this case, a PBT library such as QuickCheck[6] can test the validity of this property by generating lots of different inputs. Each case is evaluated to ensure that inserting the generated value yields a valid BST. A main advantage of the generators compared to manually writing the tests, is that lots of common edge cases can be tested automatically without having to choose the inputs on your own, potentially missing some input that would break the property. The study concludes that there is still much to learn about PBT and the challenges that arise when it is applied in an industry setting.

[6]https://github.com/nick8325/quickcheck

## 2.4 Technologies under Investigation

During the study, a number of tools will be used to implement each test generation technique. The two main tools are Hypothesis and PICT.

### 2.4.1 Hypothesis

Hypothesis [19] is an open source Python library that is primarily used for property-based testing. It generates arbitrary data according to the specification given to it by the user, and then checks if the property holds. An example of a property could be that the sum of two positive numbers should always be positive. In the case of it finding an example that breaks the guarantee, it will simplify said example to find the smallest example that causes the test to fail [18]. This simple technique is what allows testers to have more trust in their implementation than having a few basic test cases as they know the implementation has been tested against hundreds or even thousands of inputs. The number of generated inputs can be configured, allowing for varying degrees of confidence. This value can for instance be lowered for ordinary CI testing, but be increased during release testing. This demands less resources during CI runs but allows Hypothesis to validate more of the input space before a new release, at the cost of time and computational resources.

Additionally, Hypothesis supports model-based testing through its stateful testing capabilities. The library provides a `RuleBasedStateMachine` class, which allows the user to define a state machine model in which its operations modify the state. Then, the SUT is validated against the model to check for inconsistencies. State transitions are made using the `@rule` decorator that describe how the state is supposed to change when an action is performed. The user can also set `@invariant`s or `@precondition`s if needed. Invariants are properties that should always hold, and preconditions are conditions that must be true for the rule to be eligible for selection. If the state machine finds any issues, Hypothesis will minimize the failing test case to assist with debugging.

Hypothesis also includes a database that they call a cache, which stores failures from previous runs. During the next run, failures will be replayed to check if they still fail. This is useful for PBT and MBT tests, where it may take a while for failing inputs to be generated. By saving these failing inputs, the next run can be quicker. It also serves a purpose to check prior failures, to ensure that these have now been fixed since the last run.

### 2.4.2 PICT

PICT[7], an open source pairwise testing tool developed by Microsoft, is commonly used for combinatorial testing. It is a command line tool making it very flexible to use, with no reliance on a certain codebase or programming language. It takes a model file as input, and generates another file as output. The model file consists

[7]https://github.com/microsoft/pict

of a set of parameters, each with a number of choices. The tool will parse the input file and generate a compact list of parameter values that optimally cover the combinatorial space. The tool can be configured for different strengths. PICT is available on Windows, Mac and Linux and can be built on all three operating systems.

# 3

# Related Work

This chapter will supplement the theory discussed in previous chapters by connecting it to academic research within the field of the study.

## 3.1 Mutation Testing on C++ Software

Mutation testing is widely known within the research community, but has yet to be adopted into the industry on a larger scale [14, 28]. As a result, there is still a lack of robustness in the few available mutation testing tools in the open source community [8]. A few of the more known tools, such as Pitest[1] and mutmut [16] exist for Java and Python respectively. However, in languages such as C++ there is a lack of polished mutation testing tools [8, 28].

As a solution to this problem, this thesis will instead implement a custom fault injection tool as will be discussed in section 4.5.1. This not only solves the issue with the current lack of open source tools, but also differs the thesis from research by instead evaluating the test generation techniques in a new manner.

Örgård et al. [31] evaluated five C++ mutation testing tools that could be suitable for a continuous integration workflow. These tools include Dextool[2], Mull [8], MuCPP [5], Mutate++[3] and CCmutator [17]. The study evaluated the tools on six different C++ projects, comparing the results between them from different tests, such as execution time, mutation score, number of mutation operators and many more. In the end, Dextool and Mull were chosen out of the five as they could potentially be suitable for use in a CI workflow.

## 3.2 Test Generation Methods

As mentioned earlier, there are a number of test generation methods that can be applied within the study.

Zafar et al. [30] compared manual, combinatorial and model-based test generation

[1]https://pitest.org/
[2]https://github.com/joakim-brannstrom/dextool
[3]https://github.com/nlohmann/mutate_cpp

techniques in an industrial setting. Each technique was evaluated using MC/DC coverage, requirements coverage and efficiency based on a cost model. The manual test suite was created by developers at Alstom Transport AB, where the study was conducted, while the other test suites were generated by CAgen [29] and TIGER [20], two test generation tools. The result was that each technique reached a substantial level of MC/DC coverage. However, the MBT-generated test suite achieved the highest MC/DC coverage, as well as the highest number of unique test cases. Meanwhile, the combinatorial test suite was the most efficient test suite of the three.

A reappearing problem with test case generation is the lack of tools for languages such as C++ [13, 15, 23]. There exists polished test case generation tools, for example EvoSuite [10], Hypothesis and Randoop [21], but they are only compatible with languages such as Java and Python. This lack of support for C++ is a common pattern when it comes to software testing which could be attributed to C++ complex syntax and semantics [15, 23].

Herlim et al. [15] introduces CITRUS, an automated unit test tool for C++ programs as a solution to this issue. It functions by analysing the source files of the C++ program, and then generating test driver files. These files consist of function calls to methods in the original C++ source code. Finally, it mutates the test driver code to generate more test drivers. After experimenting on a number of real-world C++ programs, CITRUS achieved up to 95% statement and 79% branch coverage.

A notable difference between the thesis and research is in its industrial implementation and application. Test generation techniques are rarely compared in an industrial setting, as techniques are usually compared using synthetic benchmarks rather than evaluated on a industry product. This also includes other dimensions such as programming language support, implementation difficulty and scalability which is sometimes omitted from research.

## 3.3 SDK Testing

Testing an SDK is not the same as testing a product. Since the SDK is not a single product, a clear SUT does not exist. It is still unclear within research on the optimal technique to use to test SDKs.

Turilli et al. [25] developed the ExaWorks SDK which aims to assist users during development of scientific workflows on high-performance computing platforms. Before ExaWorks released, the tools and workflow technologies at the time were of varying robustness and had limited capabilities. They stated that "the SDK is a curated collection of workflow technologies engineered following current best practices and specifically designed to work on HPC platforms" [25]. To validate the functionality of the SDK, an infrastructure that enables testing was designed and developed. The main challenge when testing on HPC platforms is the varying hardware between HPC clusters. The infrastructure revolves around a test runner running on a GitLab continuous integration pipeline. It executes a set of tests to validate the

integration between two or more components, which is later displayed within the included dashboard.

# 4

# Methods

The methodology of the study consists of three major phases. The first phase that will be discussed in section 4.2, revolves around selecting the testing techniques that will be evaluated and compared. Once this phase is completed, the implementation stage will begin. The implementation phase itself has three sub-phases: the tool selection in section 4.3.1, tool integration in section 4.3.2 and the final implementation in section 4.3.3. Finally, the last phase in section 4.5 is the evaluation phase, where each technique will be compared with each other.

## 4.1 Evaluating the results

The selected testing techniques will be compared in two ways. Firstly, a qualitative analysis of the experience with each technique during the study. This will both be based on our own opinions as formed during all stages - highlighting advantages and disadvantages with each technique, as well as their differences - and an interview with employees at Carmenta sharing their thoughts on how well the techniques fit in their domain.

Secondly, a quantitative analysis will be performed using fault injection. This will provide data comparing how well the techniques catch introduced bugs, as well as describing potential trends or drawbacks for the different techniques. Both analyses will focus on the test techniques rather than the tools, as there are alternatives to the tools used in the study that may influence their evaluation. The different parts described in this section will be detailed in the following sections.

## 4.2 Technique Selection

Since there are many testing techniques available, a selection of which techniques to evaluate is necessary. The goal is to provide a useful comparison, including only relevant techniques. For each technique, a tool that implements that technique has to be chosen. To determine which techniques and tools should be evaluated, the methodology presented in figure 4.1 was used. In the figure, the goals and motivations of each step are presented in the accompanying clouds.

Selecting the testing techniques to investigate needs to be done with close ties to

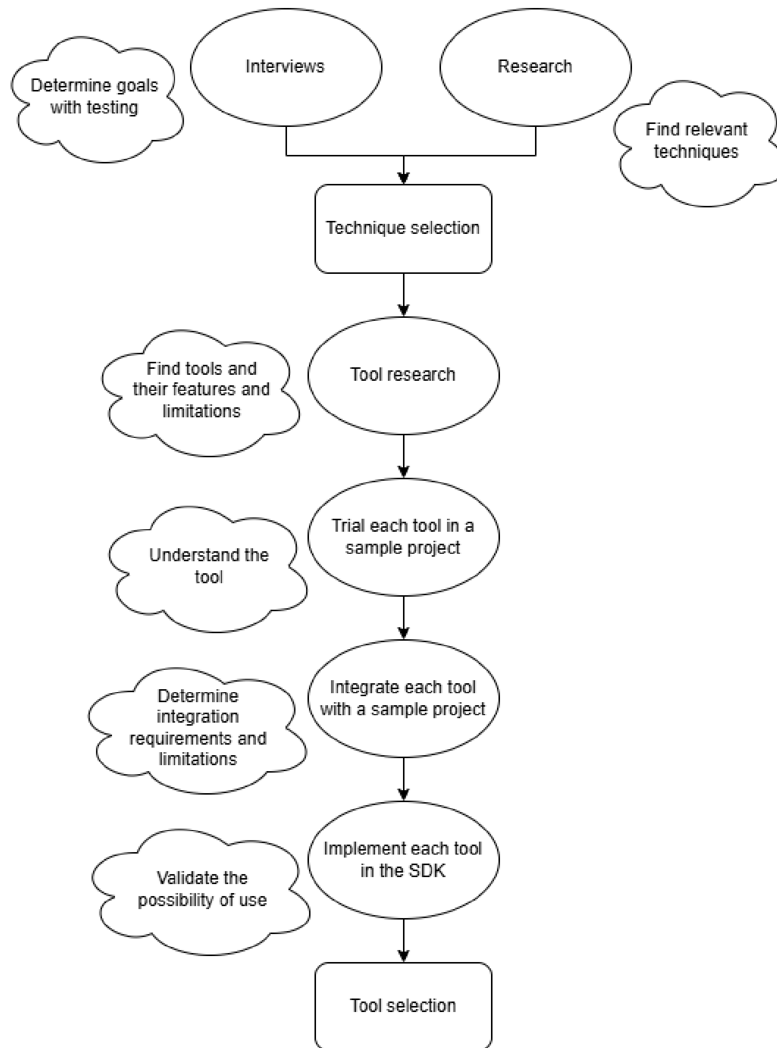



**Figure 4.1:** Technique and tool selection process

the domain and SUT in mind. Therefore, interviews will be held to understand the testing needs and focuses, as well as the shortcomings at Carmenta. The interviews should be open and semi-structured, so that each interviewee can speak their mind and share their wants and wishes. When analysing the resulting answers, conclusions can then be made for which software testing techniques would be beneficial and fill the testing needs. A variety of employees of different experience levels and roles should be interviewed, to gain insights from several perspectives. The results should also be analysed with this in mind, as not all interviewees will have experience in the testing field and may therefore not know what is possible or what types of testing are available.

### 4.2.1 Choice of Data Collection Method

To collect the data to aid the technique selection process, interviews were chosen. Interviews were picked over other forms of data collections based on a few criteria. To begin, the number of employees relevant to the domain of the study is low. At

Carmenta, there are around 25 - 30 developers, 2 software testers, and a few project managers. This limits the effectiveness of other data collections methods such as surveys, since they are typically used to capture a wide range of data [22].

Furthermore, since the goal of the data collection is to assist in making the choice of testing technique for Carmenta's product and use case, input from affected parties would be prioritized. Therefore, semi-structured interviews were chosen due to their characteristic of allowing the interviewer to probe for additional information [22]. By asking an initial question on a subject, the interviewer could ask relevant follow-up questions to further gather data on the subject.

### 4.2.2 Interviews

The questions asked during the interviews can be categorized into three sections: general information, software testing knowledge and SDK testing thoughts. The questions gradually gravitate towards more precise inquiries, meaning the interviewee is able to answer unassisted in the beginning, before being guided to asking directly about the SDK testing. No testing techniques are mentioned, or otherwise described, to ensure that the answers are not biased towards any technique.

The first part of the interview is directed at collecting general information, to be presented together with the results for putting answers in context. The second part is aimed at understanding the current testing situation and uncovering areas of improvement and importance. The Likert scale questions have the goal of assigning a numerical value of the interviewees opinions, while forcing them to motivate their answers. As an example, if an employee answers the trust question with a four, a follow up question will be asked to make them explain why they scored it as such. This has two goals, firstly to get data supported by motivations, and secondly to understand their frame of reference. If they score a four, we would ask what is missing from a ten and why it is better than zero. Then, we can get an understanding of how they classify the answers, as interviewees can have different frames of reference.

The interview results were compiled into a set of conclusions using content analysis. As we conducted the interviews, we picked up some patterns and ideas during the time they were held, which aided in analysing the notes from all interviews once they had all concluded. This enabled us to "start with a specific idea of what to look for" [26]. We then performed conceptual analysis, which is the practice of "identifying and quantifying (determining the number and frequency) of various concepts or ideas expressed in the media" [26]. By analysing trends and differences in the answers, conclusions can be drawn for what the interviewees agree on, and which ideas are present from different perspectives.

A total of 4 employees were selected and willing to be interviewed for the study. Each employee was asked the general information questions shown in table 4.1. The purpose of these questions is to collect the necessary information about each employee to provide context for their responses. The results of these questions are presented in table 4.2.

| General Information | |
|---|---|
| **1** | What is the name of your position? |
| **2** | What do you do at Carmenta? |
| **3** | How long have you worked in the industry? |
| **4** | How long have you worked at Carmenta? |
| **5** | What are your connections to software testing at Carmenta? |

**Table 4.1:** Interview questions regarding general information

| ID | Role | Time in industry | Time at Carmenta | Connection to testing |
|---|---|---|---|---|
| 1 | Developer | 3 years | 3 years | Using it at work |
| 2 | Developer | 6 years | 3 years | Using it at work |
| 3 | Project Manager | 18 year | 18 year | Using it at work |
| 4 | Test Lead | 7 years | 1 year | Planning tests, organization and test plan creation |

**Table 4.2:** General information of interviewees

As shown in table 4.3, the software testing knowledge section contains two quantitative questions where the interviewee will rank their knowledge of software testing and software testing techniques, ranging from one to ten.

Lastly, table 4.4 shows the final section which contains both quantitative and qualitative questions related to the testing conducted with the SDK. This section will be critical due to its importance for the project. These questions will guide the projects direction moving forward, deciding which techniques will best fit the Carmenta product.

### 4.2.3 Focus Group

Towards the end of the study, the same participants from the initial interviews will be invited to participate in a focus group to evaluate and discuss the chosen methods and their results. The focus group session consists of two parts. The first part is to present the necessary background information about each test generation method that was used, and some examples for each one. The second part of the session consists of a demonstration of the test suites developed for the study and a discussion regarding the participants' ideas of their use case within Carmenta.

These discussions are meant to allow the participants to have their own input on which technique might have the highest potential, and in what areas of their software. This assessment is important since it will confirm whether the test generation methods are useful for answering RQ2.

### 4.2.4 Anonymization

All interviewees, who were also later part of the focus group, consented to both sessions. They are kept anonymized as the exact individuals are irrelevant to this

| | Software Testing Knowledge |
|---|---|
| **1** | How experienced are you with software testing in general?<br>1: Not familiar<br>5: I know some terms<br>10: I am comfortable with several testing terms<br>and have written tests other than unit tests |
| **2** | How experienced are you with software testing techniques?<br>1: Not familiar<br>5: I have heard some names and have a vague understanding of them<br>10: I know at least 3 techniques and could explain them |

**Table 4.3:** Interview questions regarding software testing knowledge

| | SDK Testing |
|---|---|
| **1** | How much do you trust the testing of the SDK?<br>1: I do not trust it<br>5: I am comfortable with the testing and expect it to catch most major bugs<br>10: I trust it to find most bugs and cannot see any major or moderate holes or issues |
| **2** | What are the best aspects of the SDK testing? |
| **3** | What are the worst aspects of the SDK testing? |
| **4** | What are some opportunities in the SDK testing?<br>Where you know a solution or have an idea of how it could be solved/improved |
| **5** | What are some flaws in the SDK testing?<br>Where you can identify an issue but do not not exactly how it can be solved |
| **6** | Open comments |

**Table 4.4:** Interview questions regarding the current testing of the SDK

study. During the focus group, the participants also consented to an audio recording that was used to provide quotations.

## 4.3 Technique Implementation

Once the techniques are chosen, tools must be selected for each technique. The tool selection will be split into three main phases: tool trial, integration and finally the implementation phase. When investigating a new tool, the initial work will be done as an exploratory trial before potentially moving on to the integration stage if it is deemed successful.

### 4.3.1 Choosing the tools

During the tool trial phase, a simple SUT is to be used as a testbed for the language required by the tool or technique under investigation. These SUTs were initially ba-

sic projects under MIT licences[1], granting free use and modification of the software. As our Python testbed, we used the **PythonProjectTemplate**[2]. The project was extended with additional functionality and methods, for which tests were generated. The Python testbed was used for all exploratory research of Python testing techniques, allowing easy modification of the SUT and a simple and quick environment to work in.

Likewise, another SUT was used for C++ testing. This testbed was based on the **cpp-project-template**[3]. The aim in this phase is to be able to research the available tools to investigate their features and determine their shortcomings. Many tools lack proper documentation, and in some cases the documentation is aged and may therefore no longer reflect the latest version of the tool. By using the tool, issues and drawbacks can be discovered that will affect the suitability of the tool for the SDK.

### 4.3.2 Compatibility with the Environment

Once a tool or technique had been vetted in the prior phase, an attempt was made to integrate it properly with the testbeds mentioned in the prior section. This included developing the necessary scripts and surrounding structure to get it set up correctly. The main point of this phase was to gain a better understanding of the tool, and to polish the integration. By performing all the steps that would be required in the real environment, any issues or shortcomings related to environment integration would arise in the testbed, where investigating them and finding solutions is both faster and easier.

### 4.3.3 Implementation phase

After the tool had been thoroughly researched through the previous two phases, the final step was implementing it into the SDK. This leveraged the learnings and findings from prior work to apply it onto a more complex system. By this point, enough confidence in the tool and surrounding infrastructure would be present, allowing for a smoother implementation where issues are more likely to involve the SDK rather than the tool itself. This would also be beneficial during discussions with the SDK developers, as they would be able to see a working example to understand the differences with the SDK they are familiar with. Eliminating uncertainties, which is the tool itself from the developers' point of view, helps understanding where issues lie and where solutions may be.

As the SDK itself is written in C++, the native techniques can be implemented directly whereas the Python techniques will use the Python API which is part of the SDK. The API exposes a Python library that interacts with the C++ engine, allowing for Python tests to be executed against the C++ source code. Faults or issues in the API can therefore affect the Python test implementation. However,

[1]https://opensource.org/license/mit
[2]https://github.com/franneck94/PythonProjectTemplate
[3]https://github.com/ssciwr/cpp-project-template

the API is closely tied to the source code and the difference is expected to be negligible. Should there be any issues or unexpected test fails, developers will assist in determining whether it is related to the source code, or caused by an API deviation.

Each technique is to be allocated the same amount of time to write tests for the SDK. This attempts to make the comparison as fair as possible, and will be affected by drawbacks of the techniques, such as complexity or slow execution leading to fewer produced tests. This is a conscious decision made to make the comparison more realistic. A less powerful but quick and easy to use technique may be more beneficial and lead to better test suite quality than a complex one where each test case is of higher quality, but means less test cases can be written in the same time frame. Writing a single unit test for all possible inputs to a function would in theory allow you to exhaustively test that function, but is neither feasible nor a good idea in practice.

Although each technique is allocated the same amount of time, the comparison will not be entirely fair. That would for instance require a large number of independent testers, each writing tests with the techniques and then aggregating and comparing them. Due to the time constraints of the study, it is not feasible to perform a perfectly fair comparison.

## 4.4 Scenarios

As the SDK is not a single product, there does not exist a clear SUT to use during evaluation. This distinction is as mentioned in section 3.3, an aspect that differs this thesis to research, where test generation techniques are usually compared on a singular product. In order to test the SDK a total of five scenarios of various complexity will be constructed with the help of developers, which will be used as the SUT. Three of the scenarios will include basic configurations, such as a single operator performing a simple task. The other two scenarios will include more complex configurations with multiple operators that simulate a customer implementation. The range of scenarios is meant to capture different levels of software complexity, from simple parts of a solution to a more complete stand-alone solution, although they will in practice not be completely ready to use as software solutions.

The three simple scenarios include:

1. ***FilterOperator***: An operator that filters out features not fulfilling a condition.
2. ***ConnectLineOperator***: An operator that connects two line features with common endpoints within a certain radius of each other to create a longer line.
3. ***RectangleClipOperator***: An operator that clips features, such as rasters and vectors, with a rectangle.

Then there are two complex scenarios:

1. ***SplitFilterFunnelChain***: An operator chain consisting of a SplitOperator, splitting the input data into two identical copies, two FilterOperators filtering on a certain condition, and lastly a FunnelOperator, combining the two outputs. Figure 4.2 shows the operator chain.

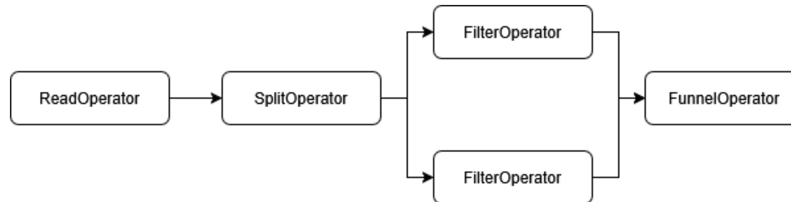



**Figure 4.2:** SplitFilterFunnel Operator Chain

2. ***ThinInsideBufferChain***: An operator chain that starts with a SmoothOperator that converts lines and polygons into smoother shapes using splines. The output is then split with a SplitOperator. One of the outputs is used as input for a LineThinningOperator that removes unnecessary points. The other output is used as input in a BufferZoneOperator that generates a 2D buffer zone around a vector. Finally, the outputs from the LineThinningOperator and BufferZoneOperator are used as input in a PolygonClipOperator that clips features with polygons. Figure 4.3 shows the operator chain.

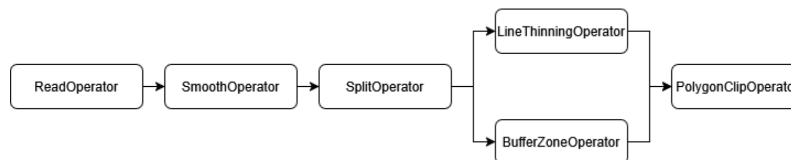



**Figure 4.3:** ThinInsideBuffer Operator Chain

## 4.5 Technique Evaluation

To evaluate and compare the techniques, the test lead and the developers will receive the findings and explanations, and their verdicts will be collected. This will form the basis of the qualitative comparison, which will incorporate context based biases depending on how useful the techniques are in the geospatial domain, on a highly configurable SUT. Simply basing the results on the effectiveness of the test suites does not paint a complete picture for this study. Instead, results will be collected from multiple perspectives. These will be combined into a final conclusion for each technology which tries to give a better understanding of the benefits and drawbacks in an applied setting. There, a technique that is hard to understand could prove impractical as the users will be developers and test engineers, not researchers in the software testing field. We therefore aim to provide useful insights for others who would like to implement these techniques, so that they can make their own decision based on which perspectives are most important in their situation.

### 4.5.1 Why Fault Injection was Chosen over Mutation Testing

To provide a quantitative result, a technique known as mutation analysis is used. The two main forms of mutation analysis are mutation testing and fault injection. The most efficient choice when comparing test suites is to use mutation testing, as it will automatically introduce mutations into the source code, run the test suite to record the number of failing test cases, and return a mutation score. As previously mentioned in section 2.3.1, the mutation score is used to rank the strength of a test suite, allowing for comparisons between them.

Therefore, during the tool selection, multiple mutation testing tools were chosen to be trialed to ensure they would function for our use case. Some tools that were explored are Mutate++, mutmut and Mull.

Mutmut is a Python mutation testing tool. In the earlier stages of the study it was thought that it would be possible to conduct the mutation testing directly onto the Python API. It was later learned that this was not possible since the Python API is built within the SDK build system, and does not exist as a direct copy of the C++ version.

Mutate++ was one of the first C++ mutation tools that were explored. It is a web app that runs locally with its own web UI. During the trial of the tool, two major issues were identified. First, the tool did not seem to have support for external test suites, and uses the CMake[4] ctest drivers to execute its tests. Secondly, the main issue which most C++ mutation tools face is that it requires the compiler to recompile the code for each mutation. When a mutation testing tool needs to inject possibly hundreds or thousands of mutants and recompile the code for each one, the time required quickly scales.

This exact scenario is what the mutation testing tool known as Mull [8] solves. Instead of recompiling the codebase for each mutant, Mull will instead create mutations of the program within memory. Mull will inject all possible mutations into the program's bitcode with each mutation hidden under a conditional flag that can be toggled to enable said mutation. The program is then compiled once and later run once per mutation, enabling mutation testing without the need for recompilation of the code [6].

During the tool trial stage of the study, a number of attempts to incorporate Mull into Carmenta's build system were made. There were a number of issues that occurred during the this phase. To begin, the first issue was a mismatch with Mull's compatible compilers. Mull requires programs to be compiled with either Clang or LLVM, while Carmenta mainly uses the Microsoft Visual C++ compiler, MSVC.

Furthermore, since Mull is a C++ mutation testing tool, its most common use case revolves around a C++ code base and test suite. For this study, the source code

[4]https://cmake.org/

is written in C++ and tested through a Python API using a test suite written in Python. This complex configuration causes additional issues. Mull has native support for non-standard test suites for the scenario of when a test suite is written in another language than the codebase [7]. However, this only functions if Mull can access the compiled executable file of the C++ code to apply the mutant on. This access to executables is not available through the Carmenta build system.

Due to these issues and the time constraints of the study itself, the decision was made to instead use fault injection over mutation testing. Due to the simplicity of fault injection, there is no need to use an already existing tool allowing for a more custom implementation compared to mutation testing. This approach also removes any dependency and compatibility issues that come with existing tools.

### 4.5.2 Fault Injection

A selection of manual faults will be introduced into the SDK, each tested with the generated test cases to also provide quantitative results. The manual faults will be inspired by mutation testing, as well as any prior faults found in the relevant parts of the code base to make the faults as realistic as possible.

The faults will be of various types, in order to evaluate the performance of the test suites across differing fault categories. Following, are the types of fault to be introduced.

- **Constant change**: A constant is replaced with an incorrect value
- **Logical**: Incorrect algorithm implementation, forgotten steps, or other faults related to logical developer mistakes
- **Variable reference**: A referenced variable is exchanged with another variable reference
- **Wrong boolean operator**: A boolean operator is replaced by another

These faults were designed and created during the study. They stem from the fault categories introduced in section 2.3.2.1 from Grottke and Trivedi, and Catolino et al.

Injection and testing runs will be completed a total of five times in order to form an average result. This is important to show differences between the techniques when it comes to consistency and their performance over time.

## 4.6 Implementation of the Fault Injection Tool

As the decision to use fault injection was made, the need for a custom tool was apparent. The process of injecting faults, rebuilding the code and running the test suites is quite straight forward. However, as it is a process that needs to be repeated

a lot during evaluation, the decision was made to automate it. This not only saves time, but also ensures that each iteration is the same, avoiding human mistakes such as missing a fault, injecting the incorrect fault, or forgetting to rebuild the code before a test.

The tool should be able to mutate specific lines of code, LoC, in the Carmenta C++ source code, recompile it, run the PBT, MBT and CIT test suites, and save the results for future comparisons. Developing a tool for this use case allows for quick re-runs and greater amount of tested mutants. Furthermore, automating it means that the evaluation can be automatically repeated, which may be required if some data is found missing or an issue in the process would be discovered after it has been completed. The automated tasks will be described in the following paragraphs. The fault injection tool's process is presented in figure 4.4.

The first step is a clean build of the code, followed by a validation run of the test suites. This both ensures that the build is fresh, so that the injected faults will be the only changes applied to the evaluation runs. Furthermore, by ensuring that all tests pass on this clean build makes sure that no tests fail without an injected fault. Otherwise, a test suite might contain a test that always fails, making it seem like it catches all mutations during the evaluation.

The second step is injecting a fault into the code. This is done by modifying the file contents in the source code. A validation is also performed to check that the code to be replaced is only found once in the file.

The third step is rebuilding the code. This is done with the existing build pipeline for the software, thus only requiring the invocation of a build script. The build system at Carmenta contains optimizations so that only changed parts are rebuilt, making this step much more efficient. However, it is still the step that takes the longest to execute, together with running the tests. There are possibilities for speeding this up for future reruns which will be discussed in section 6.7.2.

After rebuilding the code, the fourth step is running the test suites against the newly built code. The results are serialized to files, to be used by a separate analysation script that compiles the statistics. This means that the analysis can be rerun after being extended or modified, without needing to inject the faults all over again. If a test has failed, it is considered to have detected the fault. Since it has already been validated that the test succeeded on the reference code as part of step one, the fault must have caused it to fail, thus detecting it.

The fifth step is to restore the source code files, going back to the original code. The source code is tracked by git, and by running the `git status` command, it can be verified that the code is unchanged after the injected fault has been reverted. At this point, the tool is ready to inject and test the next fault, so it starts over at step two unless there are no more faults to inject, in which case it is done.

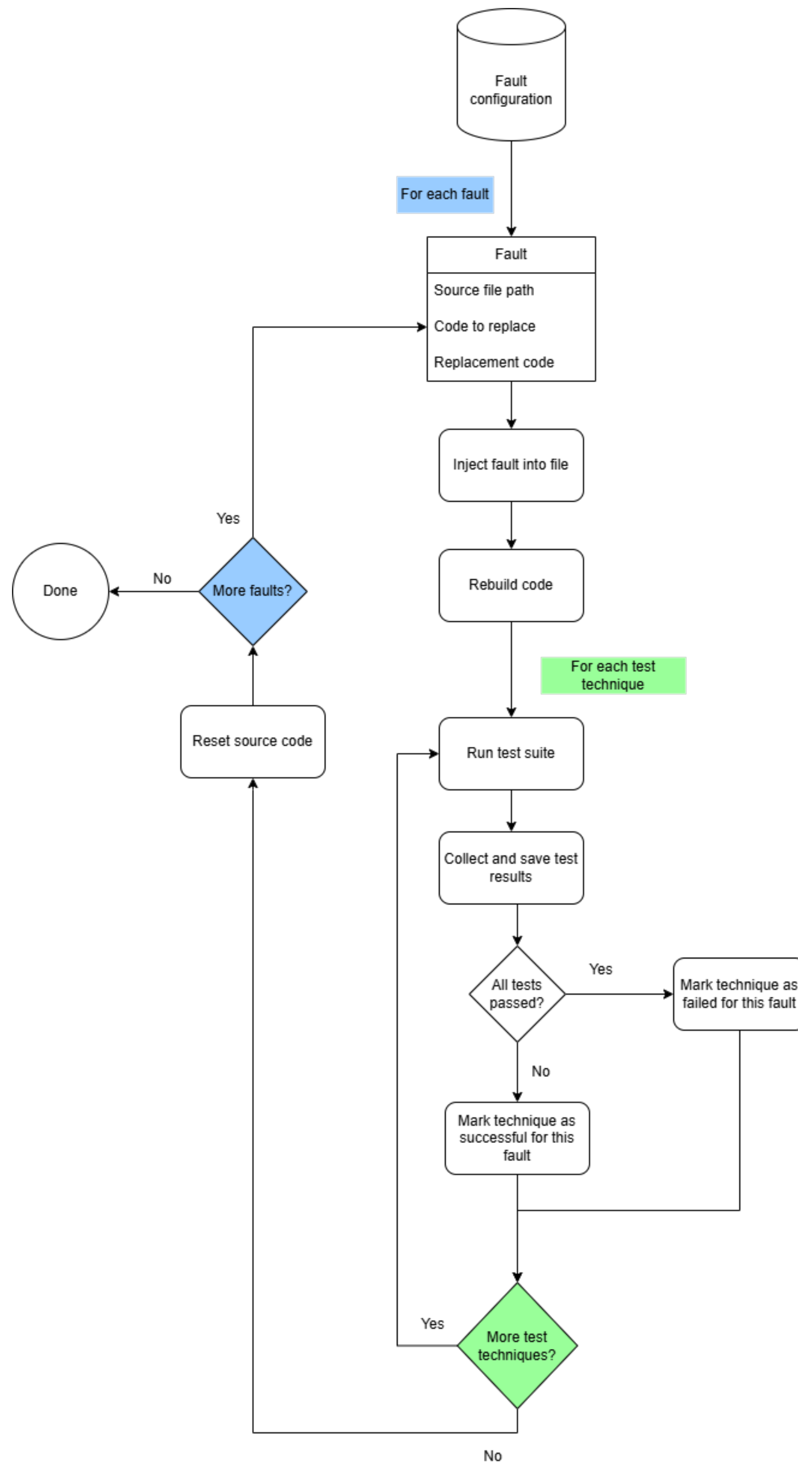



**Figure 4.4:** Fault injection tool process

# 5

# Results

In this chapter we present the results of the study. It consists of the results from the two interview sessions, the quantitative results from fault injection and our own reflections on the test generation techniques.

## 5.1 Results from the Initial Interview

As mentioned in section 4.2.2, we conducted semi-structured interviews during the technique selection stage of the project. Figures 5.1, 5.2, 5.3 showcase the results from the three quantitative questions that were asked during the interviews. During each interview, follow-up questions were asked to validate our interpretations of their answers, as well as to gather additional information.

## 5.2 Determining Relevant Testing Techniques

Based on the interviews, a few conclusions were drawn. These are presented below.

1. The team is confident in their software through their testing, which is validated by the few issues they receive from customers

2. The image tests are good, but it is hard to quickly tell what caused them to fail

3. There are many reasons why image tests can fail, some of them unrelated to an actual fault

4. The image tests are slow, and it would be good to complement or sometimes even replace them with other tests

5. Combinatorial testing is sparsely used in their current testing environment and would be beneficial

6. More unit tests would be good, especially for regression testing

7. It would be beneficial to create some distance between developers and testers, to avoid too much testing of own code

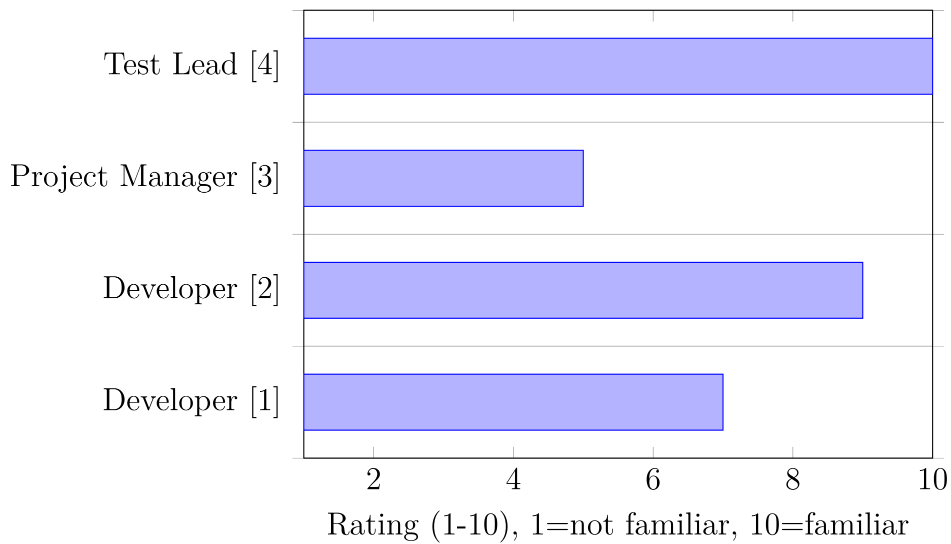



**Figure 5.1:** How experienced are you with software testing in general?

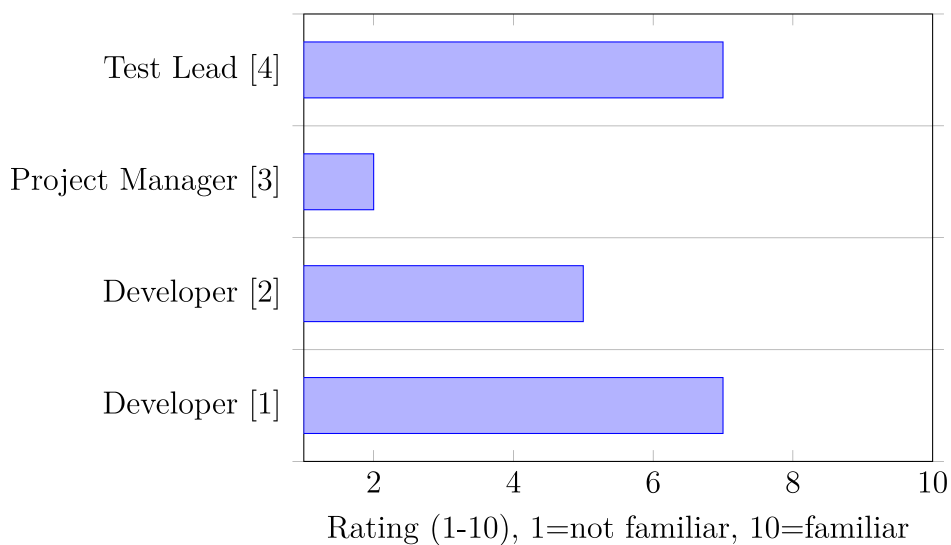



**Figure 5.2:** How experienced are you with software testing techniques?

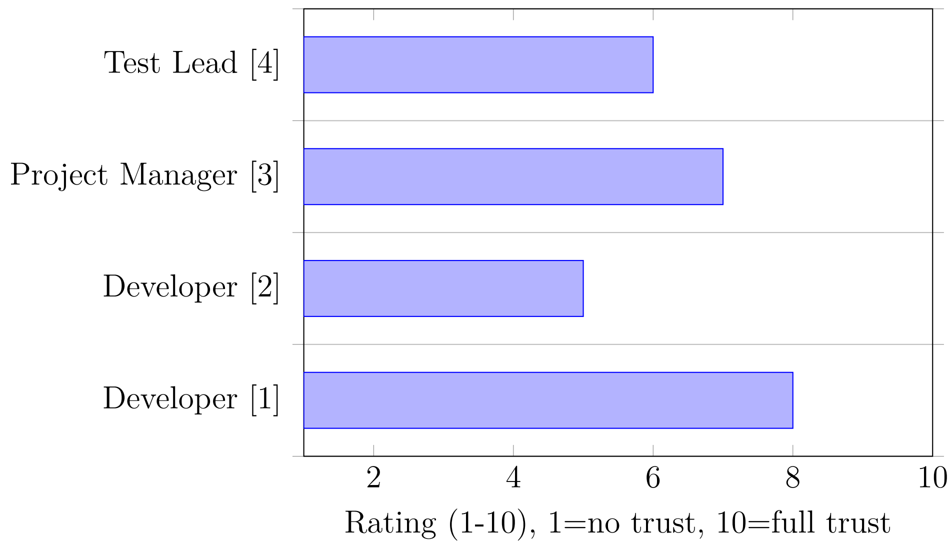



**Figure 5.3:** How much do you trust the testing of the SDK?

### 5.2.1 Justifications of Conclusions

Conclusion 1 was based on the responses from all interviewees. All developers and the project manager mentioned that the tests are used for regression testing, to protect the code base against new changes breaking old code. They all seemed to share the opinion that of everything that is tested, they have confidence in that it should catch major issues. A developer also mentioned that they do not receive many bug reports from customers, indicating that the software is stable and that their tests work well.

Conclusions 2 and 3 came from comments about the image tests, where a developer and the test lead described how image tests can fail for several reasons. The developer specified that the tests could fail due to incorrect input or issues in the logic itself, but that it could also be caused by the graphics pipeline. The test lead clarified this by stating that different renderers could produce slightly differing images as explained in section 2.2.1. These renderers could either be software renderers, or hardware renderers and exemplified this by mentioning that they sometimes see tests fail if it is run on a computer with a different graphics card. The test lead also expressed a wish to improve the comparison algorithms for image tests, so that perhaps these issues would be solved. For conclusion 3, a developer also specified that it is not obvious what the reason is behind a failing image test and that it requires knowledge about which components are involved in that particular image test.

Conclusion 4 was drawn from all interviewees. Developer 1 mentioned that there are a lot of image tests, but other forms of testing are sometimes missing. This is an indication that it may be possible to replace some image tests with automated tests. Developer 2 clarified that the image tests are slow and for the most part run manually. Another related comment was made that the quantity of image tests was too high, and that they require a lot of maintenance since there are still some that

need to be fixed to stop them from failing.

Conclusion 5 was drawn from a few sources. One of them was the project manager, who stated that a flaw in the SDK testing is the combinations of functionality, making it difficult to test the software in all scenarios. The project manager stated that different combinations are not always tested, and emphasized that there are many ways of configuring the product, some of which may hide issues. A developer also mentioned that for a lot of automated tests, only a single or a low number of configurations are currently tested.

Conclusion 6 was based on the responses from the developers, who are in charge of the unit tests. They all mentioned that they would like to see more unit tests. From follow-up questions, we understood that tests are added after issues are fixed, to protect against the issue appearing again in the future. To increase the number of unit tests would therefore improve the component testing, aiding their goal of increasing the protection against regression issues.

### 5.2.2 Decisions

From conclusion 5 it was decided to trial Combinatorial Interaction Testing. That combinatorial testing would be an improvement was also an understanding we got from speaking to the employees in various contexts, and was confirmed by the interview. It could be helpful for their test specification development, where decisions are made on which hardware, operating systems, operators et cetera to perform the tests. It is also of interest as it differs from other techniques in that it does not provide concrete test cases, but rather inputs and combinations that should be tested. This makes it very versatile and should make it applicable in multiple scenarios.

Conclusions 2, 3 and 6 led us to understand that techniques aiding with test case generation and execution is sought-after, which made us consider Property-based Testing and Model-based Testing. Property-based testing was confirmed to be included by its close ties to more traditional testing and the tests written at Carmenta today, meaning it can be used as a drop-in replacement or complement. Hypothesis allows individual test inputs to be defined, that will always be included, which allows the developers to specify explicit test cases for regression testing.

Model-based testing was also chosen because of conclusion 5, where actions with different operators can be defined. This means that they will be combined in different ways and tested in several configurations which was explicitly mentioned would be a good addition by developers. Its connection to finite state machines also means that it is useful for exploring the SDK, as it can combine different operators and actions in ways that customers may, emulating real use cases that are otherwise not explicitly tested.

All three techniques aid conclusion 4, as they will provide complements to the image tests. In some cases, it may even be possible to replace image tests with generated tests using these techniques, which would speed up the testing process and also

reduce the manual work caused by image tests.

All three techniques also aid conclusion 7 as it creates some distance between what is being tested and who is testing it. In the cases of PBT and CIT, the chosen inputs are not determined by the developer, but by another party. This avoids possible tunnel vision of developers that look over or ignore certain inputs due to their knowledge and context. It also moves a bit in the direction of random testing, trying to stress test the code with a lot of test cases and validating that it works, which was something mentioned during interviews with two candidates that would be an improvement. In the case of MBT, there is also the benefit of the test case itself being generated partly by the tool, in that you do not know exactly which combinations it will explore and therefore what the test cases will cover.

## 5.3 Comparison of Test Generation Techniques

To conduct an accurate comparison between the three testing techniques, the quantitative results from fault injection, the qualitative results from the final focus group session, and our experiences during the study were taken into consideration.

### 5.3.1 Fault Injection

After running the fault injection tool on the SUT, varying types of results were returned. In total, 23 faults were injected. The types include fault catch frequency, the number of faults caught and the time-based performance metrics of the tool itself. As mentioned in section 4.5.2, a total of 5 runs were conducted in order to validate the performance of each technique over time. Since the Hypothesis cache was disabled during the first 5 runs, another 5 runs were later conducted with the cache.

#### 5.3.1.1 Missing Test Suites

Some test suites proved flaky and would sometimes fail unexpectedly. This was never an issue with CIT tests as they were consistent across runs, but both MBT and PBT had cases where tests would be too unstable to use. These test suites were rerun many times, and would sometimes pass 10-20 times in a row, each with tens or hundreds of generated test cases, until the first failure. The flaky tests were excluded from the final fault injection run as them not passing could potentially be caused by flakiness, which would classify the test as having caught the fault, thus yielding incorrect results. MBT was found most prone to flakiness. Attempts were made to fix it during the test suite construction, but were unable to fully solve it in some cases. Efforts were made to reduce the input space in such cases, to try and guide the tests toward more ordinary inputs that would be more stable. However, this was evidently not enough in some cases.

Finding edge cases is a great advantage of using PBT or MBT, however for these flaky tests the failing inputs could not be validated as being edge cases even with

the help of the developers, and incorrect behaviour could not be identified. When reproducing failing test cases, neither us nor the developers were able to find evidence that pointed to the software misbehaving. This is the reason why the tests could not be included, as them failing was not found to be indicative of an actual fault in the code.

These missing test suites meant that some faults were not tested by certain test methods. The untested faults are presented in table 5.1. The faults missing coverage by the fault injection tool has an impact on the following results, as each method cannot be compared directly. A method with tests that do not detect a fault is different from a method without tests for the same fault.

| Fault ID | Missing in | Related to |
|---|---|---|
| VR-IAEC | PBT, MBT | *ConnectLineOperator* |
| VR-IECL | PBT, MBT | *ConnectLineOperator* |
| VR-WCR | PBT, MBT | *ConnectLineOperator* |
| WBO-NVL | PBT, MBT | *ConnectLineOperator* |
| WBO-ELNN | PBT, MBT | *ConnectLineOperator* |
| VR-ILC | PBT, MBT | *ConnectLineOperator* |
| WBO-SFAELOALT1D | MBT | *ThinInsideBufferChain* |
| CC-ICZU | MBT | *ThinInsideBufferChain* |
| L-FDTRC | MBT | *ThinInsideBufferChain* |
| L-ICOD | MBT | *ThinInsideBufferChain* |
| L-IOIR | MBT | *ThinInsideBufferChain* |
| WBO-IOC | MBT | *ThinInsideBufferChain* |
| WBO-RFOLOST | MBT | *ThinInsideBufferChain* |
| L-FTP | MBT | *ThinInsideBufferChain* |
| WBO-NPNA | MBT | *RectangleClipOperator* |
| L-PTP | MBT | *RectangleClipOperator* |

**Table 5.1:** Faults not tested by techniques

#### 5.3.1.2 Caught Faults

| Run ID | PBT | MBT | CIT |
|---|---|---|---|
| 1 | 14 | 5 | 12 |
| 2 | 7 | 5 | 12 |
| 3 | 8 | 5 | 12 |
| 4 | 7 | 5 | 12 |
| 5 | 7 | 5 | 12 |
| Average | 8,6 | 5 | 12 |
| Number of Faults Injected | 17 | 7 | 23 |

**Table 5.2:** The number of faults caught by each test generation technique

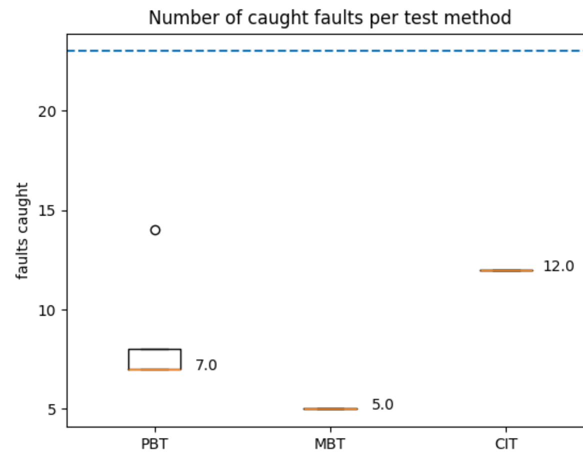



**Figure 5.4:** Number of caught faults per test method

Table 5.2 and figure 5.4 show the amount of faults caught within each run of the fault injection tool. These display the same data, as a table and a box plot respectively.

#### 5.3.1.3 Fault Detection by Category

| Fault Category | CIT | MBT | PBT |
|---|---|---|---|
| Constant change | 0 / 1 | 0 / 0 | 0 / 1 |
| Logical | 2 / 7 | 1 / 2 | 6 / 7 |
| Variable reference | 3 / 4 | 0 / 0 | 0 / 0 |
| Wrong boolean operator | 7 / 11 | 4 / 5 | 8 / 9 |

**Table 5.3:** Fault detection count by category

In table 5.3, the number of detected faults is presented out of the total number of faults for each category. A fault caught in any of the 5 different runs is counted, even if it was only caught during a single run.

#### 5.3.1.4 Fault Detection Frequency

Different runs may catch different faults in the case of MBT and PBT testing. The detection frequencies for all 23 faults are presented in table 5.4. The fault ID is a combination of the abbreviation of the fault category, VR for Variable Reference as an example, and a random ID. The IDs are not important for the results. The frequency is presented as the number of runs where the fault was detected, where the maximum is all 5 runs. Missing data is marked with the symbol -, in cases where the test method did not have any test suites for that fault.

| Fault ID | CIT | MBT | PBT |
|---|---|---|---|
| CC-ICZU | 0 | - | 0 |
| L-FDTRC | 0 | - | 1 |
| L-FTP | 5 | - | 5 |
| L-ICOD | 0 | - | 1 |
| L-IOIR | 0 | - | 1 |
| L-MP | 0 | 0 | 0 |
| L-NCHBG | 5 | 5 | 5 |
| L-PTP | 0 | - | 1 |
| VR-IAEC | 0 | - | - |
| VR-IECL | 5 | - | - |
| VR-ILC | 5 | - | - |
| VR-WCR | 5 | - | - |
| WBO-ELNN | 5 | - | - |
| WBO-FGI | 5 | 5 | 5 |
| WBO-IFG | 5 | 5 | 5 |
| WBO-ILIC | 5 | 5 | 5 |
| WBO-IOC | 5 | - | 5 |
| WBO-NPNA | 0 | - | 1 |
| WBO-NVL | 5 | - | - |
| WBO-RFOLOST | 0 | - | 1 |
| WBO-ROFG | 0 | 0 | 2 |
| WBO-ROICINO | 5 | 5 | 5 |
| WBO-SFAELOALT1D | 0 | - | 0 |

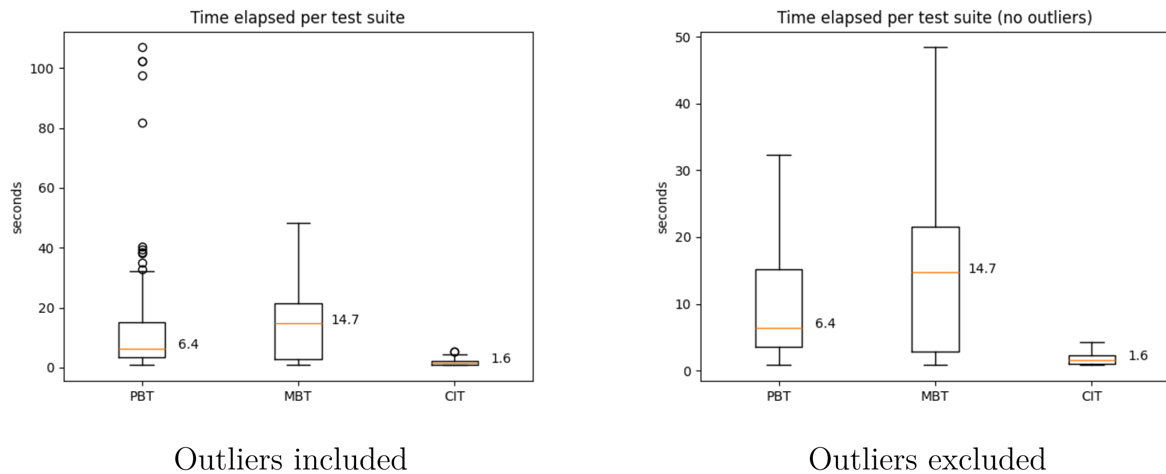
**Table 5.4:** Fault detection frequency from all five runs

#### 5.3.1.5 Performance Metrics

| Run ID | Runtime (min) |
|---|---|
| 1 | 31.76 |
| 2 | 32.23 |
| 3 | 32.12 |
| 4 | 32.10 |
| 5 | 31.25 |
| Average | 31.89 |

**Table 5.5:** The total time to run the custom fault injection tool

Table 5.5 shows the amount of time each run of the fault injection tool took. Even with a relatively low number of injected faults, the time to inject a fault, recompile the code and run the test suites quickly accumulates.




Outliers included

Outliers excluded

**Figure 5.5:** Elapsed time per test suite, with and without outliers. The medians, in orange, are presented

The two box plots shown in figure 5.5, depict the time it takes to run each type of test suite.

### 5.3.2 Fault Injection using Cache

A sequence of five runs was also performed to investigate the effects of a cache to guide test value generation. The first run was with a clear cache, and the subsequent four runs used and updated that cache progressively.

The cached and non-cached runs were performed after each other, without any changes to the code base between them. The same preconditions as for the non-cached runs were therefore true, and the data from the cached runs will be presented in this section.

The fault detection rate is presented in figure 5.6, the detections by category in table 5.6, the detection frequency in table 5.7 and the elapsed time in figure 5.7.

| Fault Category | CIT | MBT | PBT |
|---|---|---|---|
| Constant change | 0 / 1 | 0 / 0 | 0 / 1 |
| Logical | 2 / 7 | 1 / 2 | 3 / 7 |
| Variable reference | 3 / 4 | 0 / 0 | 0 / 0 |
| Wrong boolean operator | 7 / 11 | 4 / 5 | 7 / 9 |

**Table 5.6:** Fault detection count by category, using cache

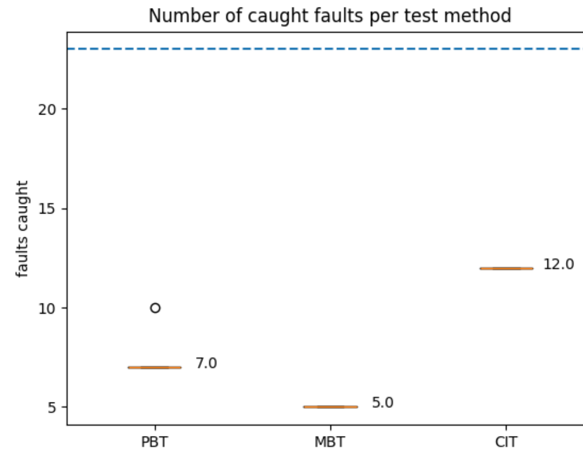
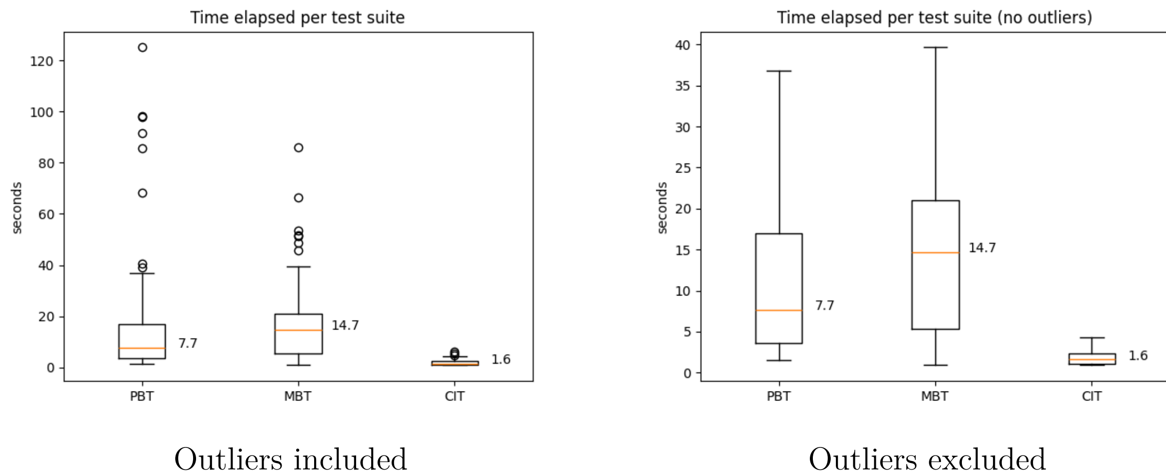



**Figure 5.6:** Number of caught faults per test method, using cache

| Fault ID | CIT | MBT | PBT |
|---|---|---|---|
| CC-ICZU | 0 | - | 0 |
| L-FDTRC | 0 | - | 0 |
| L-FTP | 5 | - | 5 |
| L-ICOD | 0 | - | 0 |
| L-IOIR | 0 | - | 0 |
| L-MP | 0 | 0 | 0 |
| L-NCHBG | 5 | 5 | 5 |
| L-PTP | 0 | - | 1 |
| VR-IAEC | 0 | - | - |
| VR-IECL | 5 | - | - |
| VR-ILC | 5 | - | - |
| VR-WCR | 5 | - | - |
| WBO-ELNN | 5 | - | - |
| WBO-FGI | 5 | 5 | 5 |
| WBO-IFG | 5 | 5 | 5 |
| WBO-ILIC | 5 | 5 | 5 |
| WBO-IOC | 5 | - | 5 |
| WBO-NPNA | 0 | - | 1 |
| WBO-NVL | 5 | - | - |
| WBO-RFOLOST | 0 | - | 0 |
| WBO-ROFG | 0 | 0 | 1 |
| WBO-ROICINO | 5 | 5 | 5 |
| WBO-SFAELOALT1D | 0 | - | 0 |

**Table 5.7:** Fault detection frequency for all five runs, using cache




Outliers included Outliers excluded

**Figure 5.7:** Elapsed time per test suite with cache, with and without outliers. The medians, in orange, are presented

### 5.3.3 Results from the Focus Group Session

The focus group session was held in Swedish, but we have translated the quotations for use in this study.

The session started with a short PowerPoint presentation of how each test generation technique functioned, along with an easy to understand example. The techniques had been previously presented during an earlier presentation at the company, which made this into more of a refresher.

Afterwards, the second part of the session began. As explained in section 4.2.3, a test from an implemented test suite was showcased, and then executed to show a real-world example of the technique implemented on the Carmenta SDK. This was done, one technique at a time, starting with PBT, then MBT and finally CIT. After a test was showcased and executed, a simple question was asked in order to start a discussion. The question was, "What is your opinion of this technique and do you believe it can be useful for testing here at Carmenta?". Once this question was asked, a discussion was started, followed up with relevant follow-up questions thought of during the discussion.

From these discussions, PBT seemed most relevant for more thorough testing of smaller components, with developer 2 explaining that "I think it [PBT] sounds the most interesting for testing smaller components in the kernel". The same developer also explained that "if you build smaller components that are clear input-output components, this [PBT] would be very interesting to use, because it is easy to write a test that has one, two or a maximum of three input variants". A question that was asked by developer 2 was if there are property-based testing tools available for C++. Continuing, the developer explained that there are parts in the kernel that they would hypothetically like to test that are not exposed by the Python API, and

that it would be good to skip the overhead of using the Python API for these kinds of tests. Another concern with PBT was mentioned by the project manager, who said "the difficulty maybe feels like describing the properties". The project manager then referenced a simple PBT example for testing an array sorting algorithm we had shown, and said that it "is a pretty trivial problem. We have pretty non-trivial solutions in many cases", hinting at the complex nature of geospatial computations and often intricate functions that make up the SDK.

The focus group seemed to agree that PBT and MBT were the most relevant techniques, with developer 1 stating that "I believe the two first [PBT and MBT] feel the most interesting" and that "I believe the most in the first two [PBT and MBT] if I had to prioritize".

A use case they discussed for MBT was to identify memory leaks, as they had a memory leak issue with an operator during a certain action. The project manager said that "I'm wondering if we would have had an MBT test on our [operator], that we would have found this memory leak". Later, developer 1 responded that "it is completely obvious that there are bugs we would have found with more explorative testing". However, developer 1 hinted that it can be difficult to use MBT since you need to ensure that the model is correct. Developer 1 drew a parallel to their image testing where "in image tests you compare with something prior", meaning test runs are compared against a reference image that you know is correct and can be determined visually looks correct. Continuing, the developer explained "if you model something and model an earlier version and compare those [...] they could both be wrong. You don't really know for certain as you do not have a reference image in that sense. [...] So there are some challenges". The reasoning was that these abstract models can be hard to decipher and validate their correctness, and drew a parallel to image tests where that is easier to determine visually, although with other issues as explained previously.

Table 5.8 presents a shortened version of the opinions from the focus group session.

| **Technique** | **Positive** | **Negative** |
|---|---|---|
| CIT | Applicable in manual testing. | Too similar to current testing, not as interesting. |
| MBT | Good with explorative testing. Can be useful for performance testing. | Difficult to validate correctness of models on complex software. |
| PBT | Good with explorative testing. Good in cases with clear properties. | Not as relevant for testing larger components. Harder to find good, testable properties for more complex code. |

**Table 5.8:** Positive and negative feedback for each technique, from the focus group session

### 5.3.4 Reflections during Implementation

Some other qualitative results are our own experiences and reflections after trialling, implementing, and using each testing technique during the study. Aspects such as the ease of use, applicability and relevance are all important factors to consider during the comparison.

#### 5.3.4.1 Ease of Use

All three techniques were relatively easy to use, however it is worth noting that we do have prior experience with all techniques. Property-based testing was found to be easier to understand as it can be thought of as an input generator for ordinary manual test cases, meaning that much of the same thought patterns and prior knowledge can be applied. However, in many cases it is required to find properties to test, as the output needs to be able to be computed based on the input. This proved to be a bit more difficult in some cases, as there were aspects we wanted to test but could not find related properties that would enable them. CIT was along the same lines, albeit even more similar to ordinary testing making it even closer for developers to adapt to. Especially given that the setup is very simple, non-reliant on any language or the source code, and trivial to run.

Model-based testing was the outlier in this case. It was deemed difficult to use for a few reasons. To begin with, it required a different mindset and pattern of thinking. This may not be easy for developers to get into without training and practice. Furthermore, creating an abstract model requires deep fundamental knowledge of the SUT, which every developer might not have. MBT is typically easier to implement and use, when it is developed in parallel to the SUT.

Property-based testing was difficult to get working, as many corner cases were uncovered that would fail tests. This is of course a positive thing, as you want the test suite to clear up behaviour for such input. However, it was sometimes hard to grasp why those tests would fail. Examples of such cases were when a generated offset was too large meaning it would wrap around the earth. This needed to be limited in the input generators, but just seeing a large number and then having to understand that the wraparound was the cause of the fail, meant the large value had to be actively compared with the circumference of the earth to understand the context of failure. This example, although a simple one, highlights one of the changes to the testing procedure caused by using PBT. Normally, it is the other way around, where the input is formed so that such cases are tested, meaning you are always aware of where the limit should lie as you are actively testing for it. When using PBT, the input is instead generated and these cases are the result of certain inputs. Many times, there are several such inputs that may trigger a fail. If you do not think of them all beforehand and correctly handle them, running PBT may expose some without you knowing which one was causing the fail.

#### 5.3.4.2 Applicability in Different Contexts

Our CIT implementation with PICT did not require the Python abstraction layer used to allow testing with Python. This means that the approach could be directly integrated and used with the existing C++ test suites that they have. Furthermore, it is a more abstract technique than the others, opening up different use cases. Since PICT essentially takes a parameter specification as input, and yields an input combination list to test as output, its use reaches beyond test case generation. It was also possible to use for planning testing efforts, and aiding in test plan construction for manual release testing.

Model-based testing was less transparent in some aspects. Although the run logs could be viewed to see which rules were applied, and in what order, the context also matters if you have a state being modified. Quickly understanding what caused a test to fail was therefore not as easy, and made debugging slightly more difficult. In contrast, both PBT and CIT show the input causing the failure, as well as the expected and actual output as its main artifacts. These can be analysed without further context needed.

#### 5.3.4.3 Relevance

CIT immediately proved relevant, especially in the context of the study. Several use cases could be quickly identified, hinting at even more possibilities in the future. Its ease of use made it a good fit for developers constructing tests, as well as its wide portability being available and buildable on Windows, Mac and Linux. The integration with pytest was simple, enabling parametrized test cases using the CIT combinations as input.

PBT was deemed relevant as it has close ties to traditional unit testing, however would be more relevant to use for unit testing rather than integration or system testing. With the added complexity and layering, PBT was in several cases quite complex to implement and resource intensive. As mentioned previously, it uncovered several corner cases at once when used for system testing. Testing isolated methods or components would be easier as the specification would be easier to determine, allowing for more precise testing. The resource use for system testing meant that the input generation took some time, leading to fewer test cases and a longer execution time. Since property-based testing differs from many other methods in that the number of tests is not specified, faster input generation would enable more test cases to run for same time frame, potentially uncovering additional bugs.

The relevance of MBT is related to that of PBT, with clear occasions where it would be very relevant to use. However, for the SUT used in this study, these were not frequent. Like PBT, it was more complex to implement than CIT or normal unit tests for that matter. The main drawback in this case was the flakiness. Should the SUT have clear specifications of its behaviour, MBT would be more relevant.

# 6

# Discussion

This chapter will discuss the results from chapter 5 and how they related to the research questions of the study. It will explain why flaky tests were an issue throughout the study. This will lead into some general drawbacks of each technique. Finally, it will conclude with some recommendations, threats to validity and potential future work.

## 6.1 Answering the Research Questions

This section aims to connect the results and findings to the original research questions that represent the purpose of the study.

### 6.1.1 Research Question 1

*What testing strategies and/or tools would be beneficial in the geospatial technology field with a highly configurable product?*

PICT was quick and easy to implement, because it did not need any integration with the source code. Since it functions by using a text file as input, and returning a text file, it avoids possible compatibility and reliance issues. This also allows it to be used in other areas than integration/system tests, such as during the release testing at Carmenta. In section 2.2.3, we described Carmenta's current release testing, which consists of testing the SDK on a combination of hardware and software setups. These combinations do not cover the entire combination space, which combinatorial interaction testing with PICT can solve.

| OS | CPU | GPU | Java Version |
|---|---|---|---|
| Windows | Intel | NVIDIA | 18 |
| Ubuntu | AMD | AMD | 19 |
| MacOS | | Intel | 20 |

**Table 6.1:** Example input for PICT

As an example, table 6.1 shows a possible input space for the release testing. There are a number of parameters of the test rig that can be configured.

| OS | CPU | GPU | Java Version |
|---|---|---|---|
| Ubuntu | Intel | Intel | 18 |
| MacOS | AMD | NVIDIA | 19 |
| MacOS | Intel | AMD | 18 |
| Ubuntu | AMD | Intel | 20 |
| Ubuntu | Intel | AMD | 19 |
| Ubuntu | AMD | Intel | 19 |
| Ubuntu | Intel | NVIDIA | 20 |
| Ubuntu | AMD | AMD | 20 |
| MacOS | AMD | Intel | 20 |
| Ubuntu | AMD | NVIDIA | 18 |

**Table 6.2:** Output generated by PICT from the input data in table 6.1

After running PICT, the results can be seen in table 6.2. If all these pairwise combinations are tested during the release testing stage, the developers can be confident that their test suite covers most of the configurations.

Hypothesis was used for property-based and model-based testing due to its support for both techniques. Its quick integration process, and ability to use custom property generation objects further proves how it can be beneficial to the industry. However, there are drawbacks to both techniques. The abstract model that is implemented for MBT should be an accurate reflection of the SUT. The easiest way to do this is to implement the model in parallel to the source code. In the case of Carmenta, the SUT was developed a long time ago. As a result, this would make the process of developing a model a large investment in both time and resources.

PBT does not share this issue with MBT, since each test is implemented individually and would work without any dependencies. As previously mentioned, the ability to generate custom objects allows it to perform well in a highly configurable environment.

### 6.1.2 Research Question 2

*Which test generation methods are useful for testing operators?*

In section 1.3, it was defined that usefulness is determined by the developers. It is how useful the developers consider each test generation technique in relation to the scenarios of operators in section 4.4. It is therefore not a single dimension, but a composite metric which takes into consideration how different its testing is to current efforts and how well it targets their testing goals. As mentioned in section 5.3.3, a focus group session was held with the participants of the initial interview. The goal was to gather the thoughts of the participants responsible for the development and testing of the Carmenta SDK.

PBT and MBT were clearly favoured by the participants over CIT. From their understanding of CIT, they believed it to be too similar to what they currently use.

CIT improving current testing was not considered as useful as the other methods which provide exploratory testing, which they have less of and is therefore of higher value to Carmenta. Even though they saw use cases, they were not directly related to writing tests but rather an assisting role for manual testing, or other forms of guiding input selection. A common type of bug report that they receive from their customers is related to a problem where the customer has encountered a specific bug caused by a combination of operators or a certain input. An efficient method to locate these bugs before customers is through exploratory testing, which is what PBT and MBT presents, and what Carmenta is currently missing.

### 6.1.3 Research Question 3

*How well do different test generation methods withstand manually introduced faults in operators?*

To begin with, CIT had the most consistent results because of its static nature. Another explanation is that its consistency comes from the limited input space rather than its algorithm. Since each combination that was tested is pre-calculated from the configuration file, the number of caught mutants did not vary between runs. As can be seen in table 5.2, it caught 12 of the 23 introduced mutants, with no variation between runs. In table 5.4 the detection frequency of faults is shown, where CIT either catches the faults 100% or 0% of the time. In that table, a value of 0 or 5 means that the fault was either never or always caught respectively. However, other values should raise more concern, as this points to inconsistencies in fault detection. For CIT, no such inconsistencies were noted.

PBT had both the best and worst performance when it came to the number of caught mutants. This was expected due to the generative aspect, where each run uses different inputs. This can be seen in table 5.2, where the first run peaked with 14 caught mutants out of 17, but in later runs only finding around 7 or 8 mutants. The run where 14 out of the 17 faults were caught is an outlier of the results, with the average number of caught faults being 8,6. In table 5.4, the inconsistencies of PBT are visible as it had multiple detections with values other than 0 or 5. These highlight the uncertain nature of PBT, where a single test result is not enough to reflect the overall quality.

The PBT test suite for the *ConnectLineOperator* scenario was disabled during the test run. As mentioned in section 6.3, there existed some extreme edge cases that generated failing tests found by PBT. The root cause of these failing tests was never located during the study. Neither the documentation available nor the developers at Carmenta were able to find the issue. This explains why the total number of injected faults is lowered from 23 to 17 for PBT.

Finally, MBT is difficult to judge due to its problematic issues regarding its implementation in certain scenarios. The test suite implementations for the *ConnectLineOperator* and *ThinInsideBufferChain* scenarios were disabled during the final test runs shown in table 5.2. A similar issue to that of the PBT *ConnectLineOpera-*

*tor* test suite previously mentioned, appeared with both the *ThinInsideBufferChain* MBT test suite but also the *ConnectLineOperator* MBT test suite, where the root issues were never found.

Therefore, since the implementation time for each test suite was specifically limited in order to preserve a fair comparison, the decision was made to disable the *ConnectLineOperator* and *ThinInsideBufferChain* test suites. These issues come as a direct result of one of the main drawbacks of MBT. The abstract model that model-based testing revolves around should be an accurate reflection of the SUT. Due to the set time limits of the study and the scale of the SDK, it is not feasible to develop a perfect model in such a short time. As a consequence, some MBT test suites were disabled due to time constraints on model accuracy.

However, when excluding the faults that affect the two disabled test suites, 7 injected faults remain. As can be seen in table 5.2, a total of 5 of them were caught, leaving 2 uncaught. This result has the second highest percentage of caught mutants after the peak run of PBT. The 5 caught mutants are caught in every single run shown in table 5.4 meaning that there was no variation in between each run. This is unexpected due to the unpredictable nature of MBT.

## 6.2 Flaky Tests

As explained in section 5.3.1.1, some test suites were too flaky to be able to use. An example is how difficult it was to define a "normal line" in the case of generating lines for the *ConnectLineOperator*. A line consists of a series of points, but it is hard to specify what characterises a line as more normal and thus less prone to flakiness. Lines would be generated that consisted of sequential points at the same location or that had points at extreme coordinates. As mentioned in section 5.3.1.1, it could not be determined that the software behaved incorrectly during the failing tests. Rather, it was common to find gray areas where there was no specification for how these cases should be handled, and the documentation did not fully explain it either. In the context of this study, the input generation sometimes proved to be a drawback, and we believe PBT and MBT therefore suffered some lost potential because the circumstances were not in favor of them. In other contexts, we could see MBT and PBT performing better as we believe clearer specifications and simpler logic could allow them to detect edge cases without causing flaky behavior.

It is not clear what the reasons behind these flaky tests were. It may be related to floating point precision errors or issues caused by the Python API, but these are just guesses. Our opinion is that MBT and PBT can be more effective in environments with clear software specifications, with ways to isolate parts of the software so that a specific property can be tested.

## 6.3 Drawbacks of the Techniques

The main drawback of PBT, which controversially is also one of its main advantages, is the close ties to the specifications of the SUT. In cases where it was not exactly known how the operators would behave, PBT tests would often fail since the property being tested did not always hold. As an example, connecting two lines required them to have the endpoints within the set tolerance. However, what happens if the distance between the endpoints are equal? Which points should then be connected, or should they all be? What happens if a line has its endpoints at the same coordinates? There was also an undocumented requirement regarding the angles between the lines, which was difficult to understand. All these questions meant that a lot of effort had to be put into carefully crafting the inputs, so the generated test cases would always fall into the category you were trying to validate.

Since PBT has no idea of what a line is, custom generators had to be written. In contrast, a developer would have this understanding and would instinctively know how to create relevant line inputs to use in the tests. It may seem trivial to generate a number of points, and letting those make up the line. However, this causes PBT to generate a lot of abnormal lines that are either not possible, or so unlikely that the effort and resources diverted to test these lines are not valuable for testing real life use cases.

To exemplify this, presented in figure 6.1 are four examples of failing test cases using PBT. These examples were randomly selected from one of the tests that was failure prone during development. Examples 1 and 2 are similar in that a middle point is placed in the same location as one of the endpoints of the other line. The other two examples are more reasonable, but probably fail due to the angles between the lines, meaning the operator does not connect them. This uncertainty is part of the issue with PBT in this case. The test generating these inputs was not designed to test edge cases, but rather that it works for normal inputs. Still, even after effort was put into customizing the input generation, these types of inputs were generated, and without a complete specification of the behaviour of the operator it is difficult to tell if it is behaving correctly for these inputs. The close ties to the specification is one of the main advantages with PBT but can sometimes be a drawback. Isolating the input space for testing certain properties is difficult with more complex input, such as lines, since limiting and controlling the generation of complex input types is not trivial. MBT showed many of the same issues as PBT, as it shares the same input generation strategies in Hypothesis leading to these odd cases.

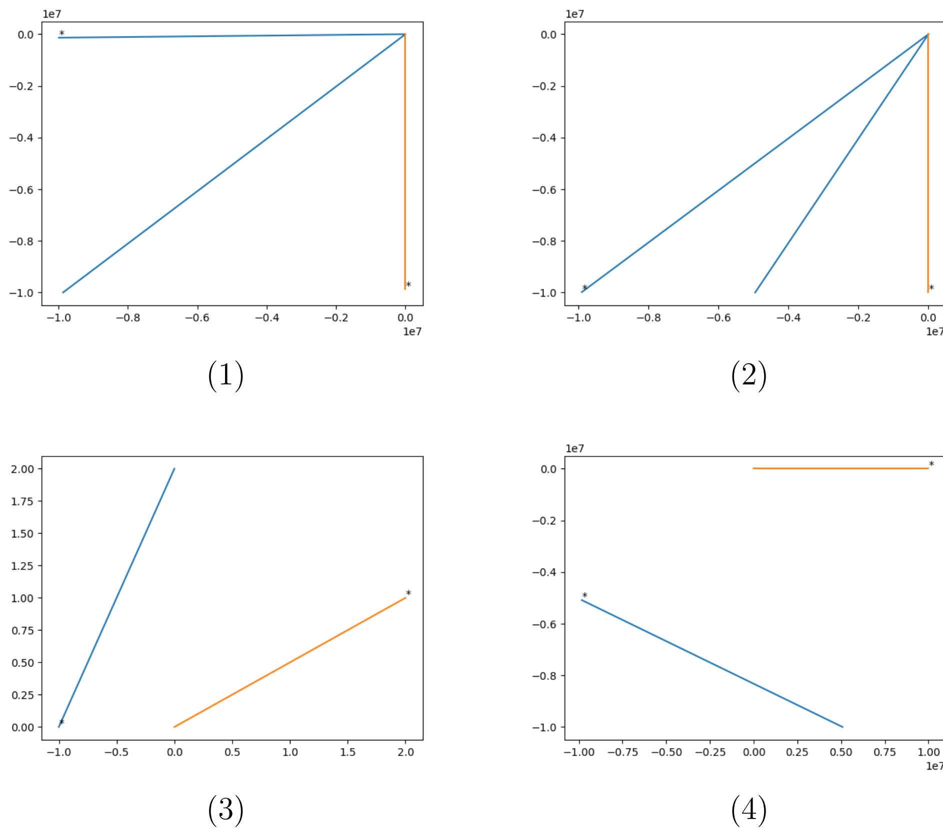



**Figure 6.1:** PBT failing connect line tests. The asterisk marks the start of the line.

One of the clear drawbacks of both PBT and MBT is the time required to run the tests. As seen in figure 5.5, MBT and PBT had significantly longer run times than CIT with the median run time for PBT of 6.4, MBT of 14.7 and CIT at a significantly lower 1.6 seconds. For applications where the tests need to be completed quickly, CIT is therefore the best choice. PBT is slower, and as can be seen by the percentiles in the box plot, is also less consistent, but is still faster than MBT by a good margin.

## 6.4 Impact of Test Generation Cache

As presented in the data, there was little difference between using the cache or not. The caught faults by category were almost identical, as seen when comparing table 5.3 and table 5.6. The only difference was for PBT, which caught an additional wrong boolean operator fault, and three logical faults during the non-cached runs. However, the first non-cached run was an outlier and was the largest contributing factor to this difference as that first run caught significantly more faults than seen in any other singular run. This is also reflected in the figures depicting the faults caught per method, as PBT was the only difference.

For the fault detection frequency presented in table 5.4 and table 5.7, the only

difference was again for PBT, and caused by the outlier run. This is even more clear in this comparison, as the difference between the tables are that four faults were detected during one non-cached run, the outlier run, but never with the cache. The final difference was for the fault WBO-ROFG, which was detected once during the cached runs, but twice during the non-cached runs, again caused by the outlier. We are unable to explain why this outlier run happened, but it emphasizes the inconsistency argument we have presented when using generated inputs, as you cannot be sure exactly what will happen during any singular run. The benefits with MBT and PBT, as mentioned, are that they can effectively explore the input space, but any singular result cannot be completely trusted to be representative.

The run times were slightly longer during the non-cached runs as seen in the differences between figure 5.5 and 5.7. This was expected as the cache is useful for failing tests, as Hypothesis will reuse the last failing inputs in the following run.

## 6.5 Recommendations

From the results of the testing conducted during the study, there are a few recommendations that can be defined. These recommendations are closely linked to Carmenta, but can be applied in any similar code base.

### 6.5.1 Regression Testing

All techniques can be utilized for regression testing. Since both PBT and MBT are closely tied to the test specification, the tests themselves should be developed in parallel with the new code. Once the tests are written and tested, they can be used for regression testing. By running these tests less often, such as over weekends or with limited amounts of generated inputs per test, the reward outweighs the resources required.

For integrations with a CI system, CIT would be a better fit as it is so much faster, which is evident in figure 5.5. PBT and MBT would be more beneficial to use during development, as mentioned, and findings during development could be used for CIT regression tests. For example, if during development, PBT or MBT finds a failing corner case that is then fixed, this input could be added to a CIT regression test. This would leverage the edge case finding ability of PBT and MBT, and use it for a more performant test that needs to run often.

### 6.5.2 Nightly Testing

An improvement to the nightly testing that takes place at Carmenta, would be to use model-based testing. By developing an abstract model of the SDK, an execution cycle could run during the night. If the model closely reflects the SDK, it could keep running different actions during the night, aiming to test as many combinations as possible. Running the execution cycle during the night gives the tool ample time to test, and in the case of a failure, it can be reported and reviewed in the morning

once developers are available.

Furthermore, the execution tool using the abstract model reflects what a user of the SDK could potentially use the tool for. The drawback of using MBT is that bugs can go unnoticed over a period of time due to the randomness of MBT. This can result in a bug remaining hidden for a period of time, which can make the fault tracing process difficult. For example, if a commit with a bug is pushed into the repository, other commits can be pushed before MBT is able to locate the issue. Once MBT finds this bug, developers might be tricked into thinking the bug was caused by these later commits. Therefore, a combination of MBT and other consistent techniques such as CIT should be considered.

### 6.5.3 Release Testing

We would recommend the use of a combinatorial interaction testing tool such as PICT to aid in the construction of release testing plans, generating system specifications for use during release testing. This would eliminate the guessing from selecting the software and hardware specifications to use during final testing, minimizing the risks of oversights. It could also help in planning which tests should be run on which machine, and we believe it would be a useful tool for helping design the release testing. This would reduce the risk of a customer encountering a configuration issue, since good coverage of the configurations during testing would be guaranteed.

## 6.6 Threats to Validity

During the course of the study, a number of potential threats to validity have been identified. These are presented in the following section.

### 6.6.1 Internal Validity

As mentioned previously in the report, we allotted a short amount of time for the development of each test suite. This was done as an attempt to make sure that each technique was given the same amount of time, to allow for an equal comparison between each test technique. Even though this was successfully executed in the study, there could be potential threats to the internal validity due to the previous experience we as developers have with each test technique. Specifically, we had some prior experience with writing PBT tests, but not MBT or CIT.

To increase the credibility of the results, the study would have to be performed by multiple independent researchers. There is a risk that we are better at writing tests using some test generation techniques, which could yield better results for these techniques. However, a technique that is difficult to use leading to worse results is not considered incorrect in this study, as it is one of the aspects being evaluated. That said, the study being repeated by several independent researchers would still improve the quality of the results.

Additionally, the answers given during the interviews relating to the current testing conducted at Carmenta could be inaccurate, or information being left out due to the participants bias towards their own testing.

The choice of using interviews and focus groups as the data collection methods of this study could in itself introduce potential bias. Answers might have differed if other more anonymized methods were used, where the participants could still remain anonymous.

### 6.6.2 External Validity

A threat to the external validity is that the entire study was conducted at a singular company. The software solution and testing methodology in place may not reflect the domain as a whole. The results are therefore tied to the context of the study, and may not be applicable in some external cases.

Another potential threat is the low number of participants in the interviews and the focus group session. To minimize the impact of this, we attempted to choose participants with different roles within the company to get a diverse mix of inputs. However, this does introduce potential selection bias to the study.

A special opportunity for this software was the availability of the Python API. Effort was put into finding suitable tools for C++ at the start of the study, for direct implementation. However, it was significantly harder and the tools were often not actively maintained or well documented. This study therefore leveraged the available Python API to enable the use of tools developed for Python, to keep the main focus on evaluating the test generation techniques rather than comparing specific tool implementations of the techniques. However, this poses a validity threat as not all software will have the possibility of leveraging tools developed for other programming languages. Some of the applicability of this study may therefore be lost, since the results are in some aspects tied to the programming languages used, and the languages available dictate which tools - and therefore techniques - can be used.

## 6.7 Future Work

The results of this study will help navigate and assist future research and implementations of automated test generation in the industry. However, to further improve the results of the study, and the use case for Carmenta, a few different aspects needs future work.

### 6.7.1 Mutation Testing

During the course of the study, mutation testing was explored, trialled and tested on Carmenta's software but to no success. A number of different mutation testing tools were tested, but none could be used in the end. The main reasons why the

tools failed was either due to compiler compatibility issues or the lack of support for an external test suite in a language other than the one used by the source code. Furthermore, the lack of C++ mutation testing tools in general was a major problem. Another possibility would be to use a mutation testing library that recompiles for each mutant, however this was not an option in this study due to the time required to run such a tool.

In order for mutation testing to be successfully implemented at Carmenta, a tool needs to be developed that supports the MSVC compiler, allows for a test suite written in a different language, and does not need to recompile the source code for each mutation, similarly to that of the mutation testing tool Mull. If this tool is implemented, it would remove the need for manually picking and injecting faults into the SUT. However, it would also introduce the flaws of mutation testing, the main one being its varying quality of mutants, whereas manual fault injection can guarantee high quality mutants.

### 6.7.2 Improvements to the Fault Injection Tool

There are a few improvements that can be made to our custom fault injection tool. First, at its current stage it recompiles the code after each fault like most of the mutation testing tools that were tested. During the end of the study we experimented with an optimization that would remove the requirement of recompiling the code between each fault. Instead, by saving the mutants they could be reused in future runs without needing to recompile the code. When running the fault injection tool, instead of injecting and recompiling the code, the test suites could instead be run against the saved mutants. A drawback of this method is that if the code base changes, the mutants would be outdated and need to be recompiled again. However, during the study the code base was static and was therefore not an issue.

### 6.7.3 Combinations of Techniques

An avenue that was not explored by this study is combining techniques for better results. As an example, one possibility would be to leverage the input generation in Hypothesis that is used by PBT and MBT to identify edge cases, and use these edge case inputs as parameter values for CIT. This could improve the data quality for inputs used by CIT to hopefully detect more faults.

### 6.7.4 Implementation at Another Company

The three testing techniques should also be tested in a different geospatial software company other than Carmenta to ensure that the results of the study are not limited to Carmenta. At other companies, integration times and difficulties would be different, and it would be interesting to compare the results with those of this study. It would also help reduce external validity concerns. Additionally, it would elucidate how transferable the software- and programming language-specific issues found during this study are.

# 7

# Conclusion

This thesis explored the feasibility of applying test case generation techniques usually found in research on an industrial SDK in the geospatial domain. Three research questions were formed in order to evaluate the usefulness of each test case generation technique in an industrial setting.

**RQ 1:** *What testing strategies and/or tools would be beneficial in the geospatial technology field with a highly configurable product?*

**RQ 2:** *Which test generation methods are useful for testing operators?*

**RQ 3:** *How well do the different test generation methods withstand manually introduced faults in operators?*

To answer these questions, a plan was developed which consisted of three major phases: the technique selection phase, the implementation phase and the evaluation phase. To select the techniques, a literature review on research within the test case generation domain was conducted, as well as a series of interviews at Carmenta. Property-based testing, model-based testing and combinatorial interaction testing were chosen, after which a number of open source tools were trialled and tested on a simple SUT testbed. This was done to get a better understanding of each tool, and to find their individual features and limitations. Furthermore, both integration requirements and compatibility issues were found during this stage. After the tools Hypothesis and PICT had been selected, the implementation phase begun. With the help of developers at Carmenta, five different scenarios were designed to be tested. A test suite using each of the three techniques were developed for each scenario during a set time limit in order to create a fair comparison. Finally, a custom fault injection tool was developed in order to evaluate the test case generation techniques and answer RQ2 and RQ3. In addition, a focus group session was held to assist in answering RQ1, but also RQ2 and RQ3.

In chapter 5, the results from the interviews, fault injection and focus group sessions were presented. Using the custom fault injection tool that was developed during the study, faults were injected into the SUT for the test suites to be tested on. A few test suites were disabled due to their flaky nature, originating from the techniques' complex implementation stage. PBT had both the best and worst performance out

of the three techniques due to its random and generative properties. CIT had the most consistent results with no variation between each run. This was expected since CIT inputs are calculated beforehand, and the same between runs. Lastly, MBT was the technique with the most difficulties, but when it worked, it caught 5/7 faults in stable scenarios. From the focus group session, it was clear that PBT and MBT were the preferred techniques because of their exploratory properties. This type of testing was highly prioritized since most of their customer feedback regarding SDK issues involves specific configurations that currently are not being tested. As mentioned in chapter 6, each technique has its own strengths and weaknesses, differentiating each of their optimal use cases.

In summary, our thesis contributes to the research conducted regarding test case generation. It explores its usefulness in a setting with no prior exposure. Its results are not only beneficial to the awareness and development at Carmenta, but also to industrial applications of research within software testing and SDK testing. In practice, we recommend using CIT for guiding manual testing efforts and for regression tests, but using PBT and MBT where applicable during development of new code where resource use is not critical to leverage exploratory testing.

# Bibliography

[1] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, Antonia Bertolino, J. Jenny Li, and Hong Zhu. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[2] Carmenta. Carmenta engine overview. `https://docs.carmenta.com/pages/carmentaengineoverview.html`.

[3] Carmenta. Operator class. `https://docs.carmenta.com/?page=operator`.

[4] Gemma Catolino, Fabio Palomba, Andy Zaidman, and Filomena Ferrucci. Not all bugs are the same: Understanding, characterizing, and classifying bug types. *Journal of Systems and Software*, 152:165–181, 2019.

[5] Pedro Delgado-Pérez, Inmaculada Medina-Bulo, Francisco Palomo-Lozano, Antonio García-Domínguez, and Juan José Domínguez-Jiménez. Assessment of class mutation operators for c++ with the mucpp mutation system. *Information and Software Technology*, 81:169–184, 2017.

[6] Alex Denisov and Stanislav Pankevich. How mull works. `https://mull.readthedocs.io/en/latest/HowMullWorks.html`.

[7] Alex Denisov and Stanislav Pankevich. Non-standard test suites. `https://mull.readthedocs.io/en/latest/tutorials/NonStandardTestSuite.html`.

[8] Alex Denisov and Stanislav Pankevich. Mull it over: mutation testing based on llvm. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2018.

[9] Simon Diemert, Adam Casey, and Jeremiah Robertson. Challenging autonomy with combinatorial testing. In *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2023.

[10] Gordon Fraser and Andrea Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Sym-*

*posium on Foundations of Software Engineering,2011*. ACM, 2011.

[11] Harrison Goldstein, Joseph W. Cutler, Daniel Dickstein, Benjamin C. Pierce, and Andrew Head. Property-based testing in practice. In *2024 International Conference on Software Engineering*. ICSE, 2024.

[12] Michael Grottke and Kishor Trivedi. A classification of software faults. *Supplemental Proc. Sixteenth International IEEE Symposium on Software Reliability Engineering*, 2005.

[13] Peter Hamberger, Claus Klammer, Thomas Luger, Michael Moser, Michael Pfeiffer, and Christina Piereder. Specification-based test case generation for c++ engineering software. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2023.

[14] Farah Hariri and Shi August. Srciror: A toolset for mutation testing of c source code and llvm intermediate representation. In *ASE '18: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018.

[15] Robert Sebastian Herlim, Yunho Kim, and Moonzoo Kim. Citrus: Automated unit testing tool for real-world c++ programs. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2022.

[16] Anders Hovmöller. mutmut - python mutation tester. `https://mutmut.readthedocs.io/en/latest/`.

[17] Markus Kusano and Chao Wang. Ccmutator: A mutation generator for concurrency constructs in multithreaded c/c++ applications. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013 - Proceedings*, pages 722–725, 11 2013.

[18] David R. MacIver, Zac Hatfield-Dodds, and many other contributors. Hypothesis. `https://hypothesis.readthedocs.io/en/latest/`.

[19] David R. MacIver, Zac Hatfield-Dodds, and many other contributors. Hypothesis: A new approach to property-based testing. *The Journal of Open Source Software*, 2019.

[20] Muhammad Nouman Zafar, Wasif Afzal, Eduard Paul Enoiu, Anthanasios Stratis, and Ola Sellin. A model-based test script generation framework for embedded software. In *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2021.

[21] Carlos Pacheco and Michael D. Ernst. Randoop: feedback-directed random testing for Java. In *OOPSLA 2007 Companion, Montreal, Canada*. ACM, October 2007.

[22] Phillips Patricia Pulliam and Cathy Stawarski. *Data Collection : Planning for and Collecting All Types of Data*. ProQuest Ebook Central, 2008.

[23] Sanghoon Rho, Philipp Martens, Seungcheol Shin, and Yeoneo Kim. Taming the beast: Fully automated unit testing with coyote c++, 2024.

[24] Yogesh Singh. *Software Testing*. Cambridge University Press, 2011.

[25] Matteo Turilli, Mihael Hategan-Marandiuc, Mikhail Titov, Ketan Maheshwari, Aymen Alsaadi, Andre Merzky, Ramon Arambula, Mikhail Zakharchanka, Matt Cowan, Justin M. Wazniak, Andreas Wilke, Ozgur Ozan Kilic, Kyle Chard, Rafael Ferreira da Silvia, Shantenu Jha, and Daniel Laney. Exaworks software development kit: A robust and scalable collection of interoperable workflow technologies. *Frontiers in High Performance Computing*, 07 2024.

[26] Janine Ungvarsky. Content analysis, 2024.

[27] Mark Utting and Bruno Legeard. *Practical Model-Based Testing : A Tools Approach*. Elsevier Science & Technology, 2007.

[28] Stefan Alexander Van Heijningen, Theo Wiik, Fransisco Gomes de Oliveira Neto, Gregory Gay, Kim Viggedal, and David Friberg. Integrating mutation testing into developer workflow: An industrial case study. In *2024 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 2024.

[29] Michael Wagner, Kristoffer Kleine, Dimitris E. Simos, Rick Kuhn, and Raghu Kacker. Cagen: A fast combinatorial test generation tool with support for constraints and higher-index arrays. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2020.

[30] Muhammad Nouman Zafar, Wasif Afzal, and Eduard Enoiu. Evaluating system-level test generation for industrial software: A comparison between manual, combinatorial and model-based testing. In *2022 IEEE/ACM International Conference on Automation of Software Test (AST)*. ACM, 2022.

[31] Jonathan Örgård, Gregory Gay, Fransisco Gomes de Oliveira Neto, and Kim Viggedal. Mutation testing in continuous integration: An exploratory industrial case study. In *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2023.