UNIVERSITY OF TWENTE.

Master Thesis
University of Twente
Formal Methods and Tools

Industrial validation of test coverage quality

Martijn Adolfsen
March 21, 2011

Committee:
Mark Timmer, MSc (UT/FMT)
Dr. Marielle Stoelinga (UT/FMT)
Prof.dr. Jaco van de Pol (UT/FMT)
Drs. Marco Pil (Info Support)
William Maas (Info Support)

Acknowledgements

This thesis marks the end of the master 'Computer Science', which I followed since February, 2009. The last two years have been very interesting and much fun. I would like to take this opportunity to thank the people that in one way or another have meant a lot for me the last two years.

I would like to thank my supervisors from the University of Twente: Mark Timmer, for inspiring and motivating me from the beginning of the project, but also for the many suggestions that helped me to improve my work. Marielle Stoelinga, for reviewing my work, even while she was on maternity leave at the beginning of the project. Finally, Jaco van de Pol, for reviewing my work despite the late involvement in the project.

Also, I want to thank my supervisors from Info Support: Marco Pil, William Maas, Marieke Keurntjes and Pascalle Hijl. Marco and William helped by reviewing my work, and providing me with data needed to conduct the experiment. Marieke and Pascalle helped by monitoring my progress and making sure I enjoyed my stay at Info Support.

I would like to thank my colleague graduate Michael Meijer and my other roommates. For providing useful suggestions on my work, but also for the lots of fun we had.

During my master program, I followed of lot of courses together with Mark Oude Veldhuis en Tim Harleman. I want to thank them for their support during these courses.

I want to thank my parents, as they supported me a lot in completing my study.

Last but certainly not least I would like to thank my girlfriend Marischa, for motivating me and helping me relax at times I felt frustrated.

Martijn Adolfsen
March, 2011

Abstract

This thesis shows the results of an empirical experiment that used three industrial case studies to investigate the effect of coverage testing on software quality. Moreover, the experiment investigated the relative costs of coverage testing.

Several test suites were created that achieved 5 different coverage percentages for the metrics statement, branch and basis path coverage. Test suites consisted mostly of test cases that were created based on the test-driven development methodology.

We show that a positive correlation is likely to exist for each metric between the coverage percentage that is achieved by a test suite and the number of faults that are found. In most cases, increasing the coverage percentage for any of the tested metrics results in a higher number of faults found. We also show that despite using coverage metrics, even with high coverage percentages, a significant portion of faults is left undetected.

Moreover, we show that increasing the coverage percentage influences the costs by a linear increase. The costs were calculated using the number of test cases, number of tested functions and the total tested cyclomatic complexity.

Contents

1 Introduction **11**

1.1 Motivation 12

1.2 Software quality 12

1.3 Research questions 14

1.4 Approach 14

1.4.1 Info Support 15

1.5 Results 15

1.6 Overview 16

2 Preliminaries **17**

2.1 Software testing 17

2.2 Test methods 18

2.2.1 Black-box versus white-box 18

2.2.2 Mutation testing 18

2.3 Flow graphs 19

3 Coverage Testing **21**

3.1 Coverage testing 21

3.2 Example code 22

3.3 Metrics 23

3.3.1 Statement coverage 23

3.3.2 Branch coverage 24

3.3.3 Path coverage 24

3.3.4 Basis path coverage 25

4 Related work 27

4.1 Garg, 1994 27

4.1.1 Experiment design 28

4.1.2 Results 28

4.2 Frankl and Iakounenko, 1998 28

4.2.1 Experiment design 28

4.2.2 Results 30

4.3 Lyu, Huang, Sze and Cai, 2003 30

4.3.1 Experiment design 30

4.3.2 Results 31

4.4 Cai and Lyu, 2005 32

4.4.1 Experiment design 32

4.4.2 Results 32

4.5 Conclusion 33

5 Approach 35

5.1 Introduction 35

5.2 Design 36

5.2.1 More on test suite creation 38

5.3 Execution 39

5.4 Analysis 39

5.5 Case studies 40

5.5.1 Integrity Service 41

5.5.2 Insurance Administration 41

5.5.3 Hosted Business Suite 41

6 Results and discussion 43

6.1 Case study 1 43

6.1.1 Statement coverage 43

6.1.2 Branch coverage 45

6.1.3 Basis path coverage 46

6.2 Case study 2 48

6.2.1 Statement coverage 48

6.2.2 Branch coverage 48

6.3 Case study 3 49

6.3.1 Statement coverage 49

6.3.2 Branch coverage 49
6.3.3 Basis path coverage 50
6.4 Costs versus fault percentage 52
6.5 Evaluation 54
6.5.1 Overview correlations 54
6.5.2 Merged results 54
6.5.3 Fault detection capability 55
6.5.4 Threats to validity 55

7 Conclusion and Future Work **59**
7.1 Answer to research questions 59
7.1.1 Subquestions 59
7.1.2 Main question 60
7.2 Future work 61

References **63**

A Test suite design costs **67**
A.1 Case study 1 67
A.2 Case study 2 69
A.3 Case study 3 70

CHAPTER 1

Introduction

One of the techniques most used to improve software quality is testing [26]. The purpose of testing is broad, and includes (1) showing that software (does not) work(s), (2) finding software faults, (3) supporting debugging and (4) investigating software quality. Due to the reliance of nowadays' society on software, testing has become a critical task. Software faults can have undesirable (or even devastating) effects. Take for example the recall on the Toyota Prius and Lexus HS 250h [28]. After several people complained about experiencing an inconsistent brake feel, a software update had to reconfigure the response time of the anti-lock brake system (ABS), and its sensitivity to tire slippage. Also, software faults are a nightmare for the economy. The National Institute of Standards and Technology assessed that they cost the U.S. industry 59.5 billion dollars annually [27].

Several testing techniques were developed to reduce the number of software faults, from formal methods [31] to ad-hoc techniques [7]. One of the concepts to improve software quality is using one or more coverage metrics, e.g., statement coverage.

A coverage metric specifies which parts of program code should be executed, e.g., all statements. Each coverage metric has different criteria and therefore has specific requirements that a test suite should meet. When a test suite meets the requirements of a coverage metric, testers assume it is able to find more faults than a random test suite that does not meet the requirements. Based on this assumption, coverage metrics are used to quantify the quality of a test suite, i.e., the higher the quality of a test suite, the greater the percentage of faults it finds.

We conduct an experiment to investigate if this assumption is indeed valid. The metrics investigated are statement, branch and basis path coverage. By executing several test suites that meet the requirements of these metrics, and by using coverage tooling, we reveal the quality of the test suites. As we know to which metric each test suite belongs, we may conclude to what extent the use of coverage metrics improve test suite quality.

Organization of this chapter Section 1.1 provides the motivation for the experiment. Then, Section 1.2 describes how we interpreter software quality. In Section 1.3 we introduce the research questions. Followed by an introduction to the approach of the experiment in Section 1.4. Then, the results are evaluated in Section 1.5. Finally, we provide an overview of the structure of this thesis in Section 1.6.

1.1 Motivation

As testing has become more and more important, testers should aim at testing a system as thoroughly as possible. Currently, 50 percent of total project costs are typically made in the test process [26]. Nonetheless, software faults still remain.

However, investing even more in the test process is simply unrealistic and not desirable by the industry. Therefore, the industry needs a method that limits the testing costs but (at least) maintains the quality of the test process.

The concept of coverage testing is one the test methods that serves this demand. Several coverage metrics were introduced during the last decades, that all aim at satisfying this demand.

Each metric has specific requirements that must be met by a test suite. Once these requirements are met, no additional test cases need to be created and/or added to the test suite. This reduces the number of test cases needed, which is very necessary, as the number of possible test cases are infinite, but the available resources (both economically and in the sense of available hardware) for testing are not.

As mentioned in the previous section, testers assume that meeting the requirements of one or more metrics increase the quality of the test suite. When this assumption is confirmed or contradicted, the industry knows whether coverage testing should be used, and to what extent. The exact relation can tell us which metric is the best to use in a particular situation, and to what extent it is financially attractive to apply coverage testing.

We investigate this relation on the following metrics: (1) statement, (2) branch and (3) basis path coverage. For each of these metrics, different coverage percentages are tested. We analyze the number of faults found, resulting in an efficiency prediction for each metric. This result can be used by the software industry as a guide for selecting coverage criteria to meet their desired quality requirements. Moreover, the result can be used by the industry to evaluate to what extent coverage testing is a profitable technique. A more detailed description of the experiment is provided in Chapter 5.

1.2 Software quality

Software quality is a concept that is interpreted in many different ways. Garvin studied how quality is perceived in different domains [20], and concluded that 'quality is a complex and multifaceted concept' which can be described from five different perspectives: transcendental, user, product, manufacturing, value-based.

The *transcendental view* sees quality as a product property that cannot be formally defined. Philosophers also refer to this type of property as "logically primitive". Most people recognize this type of quality through experience, but have trouble describing the quality properties. An example is the beauty of an object, on which different people agree. However, when asked what makes the object beautiful, people cannot really tell.

The quality in the *user view* can be defined as the degree to which a specific product satisfies the desires of a specific consumer, i.e., fitness for purpose. While this is easier to formally define than the transcendental view of quality, it is obviously very subjective. Experience shows that different users have different expectations of a product, e.g., one user may find software with a nice look and feel to be of good quality, while another user may prefer software to be simple and fast. As a direct consequence, the level of quality in the user view is not universally agreed upon.

The *product view* assumes that measuring and controlling product properties will result in improved external product behavior. Placed in the context of software quality, we can map these product properties to software quality characteristics as defined by Steven McConnell [24] and ISO 9126 [22]. This would imply that if software conforms to the ISO 9126 standard, its external behavior improves and therefore its quality.

The *manufacturing view* of quality suggest 'defect counts' and 'rework costs' as characteristics to measure. Defect counts are the number of defects found in a product during development and usage by the end-user. Rework costs are costs that are made to repair software when it is already in production. The manufacturing view states that making a product 'right' the first time is the best option. Repairing faults in the development process is considered to cost less than when the product is already in use, i.e., rework costs should be kept to a minimum.

With the *value-based view*, quality is determined by the performance that can be delivered at a certain cost. For example, a running shoe of good quality for $500 would not be interesting for a customer to buy. No matter how good the quality of the shoe is, $500 is too much to find customers that want to buy the shoe. Therefore, the value-based view considers this to be poor quality. In software engineering, the risk-based testing approach [9] can be used to improve quality in the perspective of the value-based view. Risk-based testing computes the risk of terminating the test process, by determining the probability that faults occurs, and the impact these faults have on the system. This process can be used to find a balance between the price of a product (as more testing increases production costs) and its quality.

The transcendental, user and value-based views do not allow for an objective quality measurement, as they are mostly determined by the view of the beholder. As we want to measure software quality in an objective manner, the product and manufacturing view are possible perspectives from which we could interpret software quality. However, as we will use coverage testing to find faults, we interpret software quality from the manufacturing view perspective, i.e., measuring the number of faults. The more faults a software program has, the worse we consider its quality.

1.3 Research questions

The following research questions aim at answering whether the assumption discussed in Section 1.1 is indeed valid. The term software quality should be interpreted as discussed in Section 1.2.

RQ: How should coverage testing be used by the software industry to improve software quality in a profitable way?

SQ1: What is the correlation between different coverage metrics and software quality?

SQ2: At what coverage percentages will the effect of increasing coverage diminish?

SQ3: How does an increase or decrease of the coverage percentage influence the testing costs?

The first subquestion (SQ1) asks for an overview of the quality of test suites that are created following the requirements of several coverage metrics. Moreover, the Pearson correlation coefficient is asked, so we can determine if a positive linear relation exists between metrics and software quality. The second subquestion (SQ2) asks to investigate if the positive effect of coverage metrics (assuming it exists) stops to increase above a certain coverage percentage. The third subquestion (SQ3) asks for the test effort that was required to achieve a certain coverage percentage. The main research question (RQ) asks how the industry should use coverage testing, based on the required quality of a product, and the available resources for the product. The goal is to provide the best solution (satisfying at least the minimum quality requirements), using minimum resources.

1.4 Approach

To provide answers for the research questions defined in Section 1.3, we used industrial case studies (see Chapter 5). These case studies originate from the software company Info Support, and are handpicked based on their mutual diversity. For each case study, synthetic faults have been introduced by one or more developers that actively worked on the case. By using their experience in the project(s), the faults are as true-to-nature as possible.

To find these faults, we used test cases that were created by a group of testers using the test-driven development methodology. Additional test cases were created by a second tester, based on his experience. These additional test cases were necessary to achieve (and test) higher percentages of coverage. The entire set of test cases served as a ‘pool’, from which we created test suites.

To test several coverage percentages of each metric, we created multiple test suites. Each test suite had a unique set of test cases, and achieved a certain coverage percentage for the following metrics: statement, branch and basis path coverage. After executing a test suite, we obtained the number of faults it found.

When combining the coverage percentages for each metric with the number of faults found, we obtained a result on the efficiency of the metrics under the tested percentages. For instance, a test suite that achieved 80 percent statement, 60 percent branch and 30 percent basis path coverage, was able to find 5 out of a total of 10 faults.

For more details on our approach and specific requirements for the experiment we refer to Chapter 5.

1.4.1 Info Support

The industrial case studies were provided by Info Support. Info Support is a company developing custom-made administration software for different branches, e.g., government, health care and financial. Most customers of Info Support are indexed in the Top-500 companies in the Netherlands and Belgium, and include UWV, CZ and Microsoft.

Using a software factory called *Endeavour*, Info Support aims at creating solutions that innovate the way organizations do their business. Endeavour offers different standardized assembly processes for software projects, reducing the amount of resources needed and increasing the overall process quality.

More interesting, Endeavour prescribes how the test process should be executed, e.g., test suites should have at least 80 percent statement coverage. Varying per project, different test methods are used, including test-driven development.

1.5 Results

In this thesis we show:

- Formal definitions of the statement, branch and basis path coverage metrics, based on flow graphs (Chapter 3).
- A positive correlation is likely to exist between coverage metrics (more specifically: statement, branch and basis path) and the number of faults found (Chapter 6).
- Coverage testing is likely to find a rather low number of faults. Depending on the coverage percentage it ranges from 0 to 55 percent (Chapter 6).
- The costs to increase the coverage percentage grow linearly. Even with higher coverage percentages (Chapter 6).

As the number of faults found and the testing costs both seem to grow linearly, there is no obvious minimum or maximum coverage percentage that can be recommended. While increasing the coverage percentage is likely to increase the number of faults that are found, still a lot of faults are left undetected. Therefore, coverage testing should definitely not be used as the only test method during a test process.

1.6 Overview

This thesis consists of seven chapters. After the introduction in Chapter 1, we provide preliminaries that are used throughout the thesis in Chapter 2. Then, Chapter 3 provides an introduction to the concept of coverage testing and explains several used metrics. Chapter 4 discusses related work that has been done on the relation between coverage testing and software quality. Chapter 5 describes the approach that was used. Then, in Chapter 6 we evaluate the results of the experiment. Finally, Chapter 7 provides the conclusion and introduces directions for future work.

CHAPTER 2

Preliminaries

This chapter discusses preliminaries that the user should be familiar with to fully understand the methods and techniques used in this thesis. We first provide an introduction to software testing in general, followed by discussing several test methods. Finally, we introduce flow graphs.

Organization of this chapter Section 2.1 gives a brief introduction to software testing. Then, Section 2.2 discusses several testing methods like black-box and white-box testing. Finally, Section 2.3 provides a description and formal definition of flow graphs, which is used in this thesis to describe several coverage metrics.

2.1 Software testing

As discussed in Chapter 1, one of the purposes of testing is finding faults in a software system. By providing the system under test with inputs and observing outputs, the tester can assess whether the system conforms to its specification. Providing input and observing output is done by executing test cases. A *test case* is a single test that is executed on a system under test. A set of test cases is called a *test suite*.

The life cycle of every test process roughly follows the same four stages: (1) formalizing the requirements, (2) deriving test cases based on the specification, (3) executing test cases, and (4) interpreting, evaluating and documenting test results.

As the system under test is always compared to its specification, the quality of the test process partially depends on the quality of the provided specification. Still, testing can only show the presence of errors, not their absence [14]. We can test a system very thoroughly, but still faults may remain. However, thoroughly testing a system increases our confidence in the system, i.e., we expect less faults to remain.

2.2 Test methods

This section introduces basic test terminology. Section 2.2.1 discusses black-box and white-box testing. With black-box testing, we only test to assess software functionality, i.e., functional testing. White-box testing is also used to assess software functionality, but tests a system more thoroughly (in the context of unit testing) than black-box testing, by also assessing the system's internals. This enhances the ability of the tester to find faults.

Finally, in Section 2.2.2 we discuss an interesting method called mutation testing. It is used to assess whether a test suite is correct. This is necessary, as tests can also contain faults.

Note that each section discusses a different method. However, this does not imply that a test process can be categorized in only one category (the categories are orthogonal). For example: a test process may execute functional tests, but use a model to do so. Such a process would be a combination of white-box testing and model-based testing.

2.2.1 Black-box versus white-box

Black-box testing is a test method that only deals with a system's external behavior. There are no assumptions needed whatsoever about the internal structure of the system under test. Correct behavior is observed by executing test cases, that are derived from the specification. Several techniques exist that help the tester to create an efficient test suite, including equivalence partitioning and boundary-value analysis.

While black-box testing is limited to dealing with a system's external behavior, white-box testing focuses specifically on the internal structure. By knowing the internal structure, we can design test cases that intentionally reach specific parts of the code. This allows us to test a system more thoroughly. Like black-box testing, the tests are based on the system's specification.

Gray-box testing Sometimes a combination of black-box and white-box testing is used: *gray-box testing*. This may happen if the tester is aware of the internals of a system to some extent, and uses this information to create more efficient test cases. However, the tester will not specifically assess code parts, e.g., branches or conditions, but remains to rely on the combination of inputs and expected outputs.

2.2.2 Mutation testing

Unlike the earlier introduced testing methods, mutation testing [11], also called mutation analysis, is used to test the quality of a test suite. This is very necessary, as testers can obviously make faults in constructing their tests, just like programmers do in writing code. Mutation testing is performed by testing mutants. A mutant is a modified version of the original program, that contains one or more added faults. The aim is to create multiple mutants and keep the number of faults per mutant restricted to a few. Now that a fault is injected in the program, we can validate whether our test suite finds that fault. If it is found, we say that the mutant is killed.

2.3 Flow graphs

A flow graph is a formal model used to represent the possible control flows of a process, and consists of different types of nodes and directed edges. Although Allen [8] provides a formal definition of flow graphs, we create our own definition, as Allen's definition does not allow the assignment of process descriptions and boolean expressions. Moreover, we subdivide nodes in two specific nodes: process nodes and decision nodes. This allows for a better formalization of the various coverage metrics.

In the following definition, we assume a universe of boolean expressions `Bool` and a universe of process descriptions `Proc` (represented by pseudo-code statements). We assume that flow graphs have one begin node and one end node.

Definition 2.3.1. A *flow graph* is a tuple $G = \langle P, D, L_P, L_D, E_P, E_D \rangle$, where

- P is a finite set of process nodes;
- D is a finite set of decision nodes;
- $L_P : P \rightarrow \texttt{Proc}$ is an assignment of process descriptions to process nodes;
- $L_D : D \rightarrow \texttt{Bool}$ is an assignment of boolean expressions to decision nodes;
- $E_P : (P \cup \{\text{begin}\}) \rightarrow (P \cup D \cup \{\text{end}\})$ is a function that connects process nodes to their successors;
- $E_D : D \rightarrow (P \cup D \cup \{\text{end}\})^2$ is a function that connects decision nodes to their successors.

Flow graphs always start in the begin node and terminate at the end node. Between these nodes, a combination of zero or more process and/or decision nodes exists, which are connected by edges. These edges represent the different flows of control that can be traversed. Process nodes always have one or more incoming edges and one outgoing edge. Decision nodes have one or more incoming edges and two outgoing edges. If the decision predicate evaluates to true, the first outgoing edge is traversed. Otherwise, the second outgoing edge is traversed. More formally, if $E_D(d) = (d_1, d_2)$ and $L_D(d)$ evaluates to true, control flows to d_1, otherwise it flows to d_2. The graphical notation that is used in this thesis is listed in Table 2.1.

Having formalized the notation for flow graphs, we can derive a flow graph from program code as follows: (1) program statements are represented as process nodes, (2) program decisions (e.g., if and while statements) are represented as decision nodes, (3) different control flows and execution orders are represented by arrows. Note that the derivation is language dependent, and therefore may vary per language. More information on deriving flow graphs from program code can be found in [16].

Example 2.3.2. Consider the flow graph depicted in Figure 2.1. The flow graph starts with the rounded begin node and has one edge to traverse to the decision node. If the decision is evaluated to true, the task is executed and the program terminates, otherwise the program terminates immediately. This is captured in our definition as follows:

Type	Represented by
Begin/End node	Circle
Process	Rounded rectangle
Condition/Decision	Diamond
Path	Line (direction by arrowhead).

Table 2.1: Flow graph notation

- $P = \{p_1\}$;
- $D = \{d_1\}$;
- $L_P = \{p_1 \mapsto \text{Task}\}$;
- $L_D = \{d_1 \mapsto \text{Execute task?}\}$;
- $E_P = \{\text{begin} \mapsto d1, p1 \mapsto \text{end})\}$;
- $E_D = \{d_1 \mapsto (p1, \text{end})\}$.

The execution of a program corresponds to a path through its flow graph, that formally consists of a sequence of nodes.

Definition 2.3.3. Let $G = \langle P, D, L_P, L_D, E_P, E_D \rangle$ be a flow graph. A *path* is a sequence $\pi = \langle n_1, n_2, \ldots, n_k \rangle$ such that $n_i \in (P \cup D)$ for every $1 \leq i \leq k$ and $E_P(n_i) = n_{i+1}$, otherwise there exists some m such that either $E_D(n_i) = (n_{i+1}, m)$ or $E_D(n_i) = (m, n_{i+1})$. A path always starts at the begin node, and terminates at the end node, i.e., $n_1 = \text{begin}$, $n_k = \text{end}$.

We write $n \in \pi$ if there exists an $1 \leq i \leq k$ such that $n_i = n$.

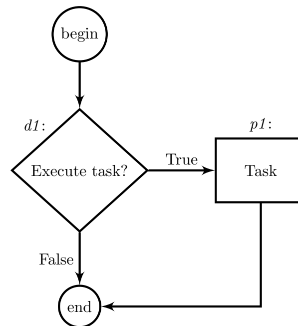


Figure 2.1: Flow graph

CHAPTER 3

Coverage Testing

This chapter explains the coverage metrics that are used in our experiment: statement, branch and basis path coverage. Each metric is formally defined in terms of flow graphs. Moreover, we provide a textual explanation, followed by an example.

For each metric, we explain how the coverage percentage can be calculated. Moreover, to improve understandability, we provide an example calculation for each metric individually.

Organization of this chapter Section 3.1 explains how coverage testing works, and introduces terminology we use throughout the rest of this thesis. Section 3.2 explains the example code and its corresponding flow graph that is used during this chapter. Finally, in Section 3.3 we explain specifically how statement, branch and basis path coverage work.

3.1 Coverage testing

When testing a software system, a tester creates one or more test suites that test specific parts or the whole system. However, as a system's size and complexity grows, the tester wants to create test suites that are as efficient as possible. An *efficient test suite* tests a system as thoroughly as possible (in terms of a larger portion of the code or more important/complicated code), with a minimum number of test steps. To compute how thoroughly a test suite tests a system, several *metrics* exist. A metric defines a way to compute the fraction of the system that is touched by a test suite. A tester may decide for himself if he wants his test suite to touch the entire fraction of the code prescribed by a metric. In other words, it is left to the tester to decide to what extent his test suite complies with a metric. This is expressed as a percentage, and is referred to as the *coverage percentage*. A test suite may be checked on multiple metrics. Therefore, a tester could decide that his test suite must touch all the code parts

prescribed by two different metrics. Moreover, he could decide to partially touch the code parts prescribed by one metric (e.g. a coverage percentage of 70 percent), and touch all the code parts as prescribed by another metric.

Each metric focuses on different aspects of the code to be touched by a test suite, for instance on covering each statement or every branch. As a consequence, one metric may require the tester to put more effort in a test suite than another metric. When a metric m_1 requires a tester to put more effort in the test suite than another metric m_2, we say that metric m_1 is *stronger* than metric m_2. Note that a test suite that touches all parts of the code as prescribed by strong metric m_1, does not imply that all parts of the code prescribed by a weaker metric m_2 are also touched. In general we assume that a test suite that meets the requirements of a stronger metric is more efficient (in terms of finding more faults) than a test suite that meets the requirements of a weaker metric. This is assumed for every possible system and every pair of test suites.

Example 3.1.1. To test a system A, a tester created the test suites T_1 and T_2, each testing a different part of the system. The tester decided that each test suite must achieve a minimum coverage percentage of 70 percent for metric m_1, and 80 percent for the weaker metric m_2. The tester finds that test suite T_1 achieved a coverage percentage of 80 percent for metric m_1, and 90 percent for metric m_2. Therefore, test suite T_1 meets the minimal requirements of the tester.

Test suite T_2 achieved a coverage percentage of 80 percent for metric m_1, and 60 percent for metric m_2. In this situation, the test suite meets the minimum coverage percentage for metric m_1, but lacks to meet the minimum coverage percentage for the weaker metric m_2. Therefore, test suite T_2 does not meet the requirements.

3.2 Example code

To show the specific properties of each metric, we introduce the code example depicted in Listing 3.1. The method *getSum* takes two parameters: *num1* and *num2*. If parameters *num1* and/or *num2* are smaller than zero, the method exits. Otherwise, it executes a for-loop that executes 10 repetitions. Each repetition calculates the sum of *num1* and *num2* and adds the result to the *result* variable. After the for-loop is exited, the *result* variable is printed to the screen.

The corresponding flow graph for this code example is depicted in Figure 3.1.

Listing 3.1: Code example

```
1 void getSum(int num1, int num2) {
2         if(num1 >= 0 || num2 >= 0) {
3                 int result = 0;
4                 for(int i = 0; i < 10; i++) {
5                         result += num1 + num2;
6                 }
7                 print result;
8         }
9 }
```

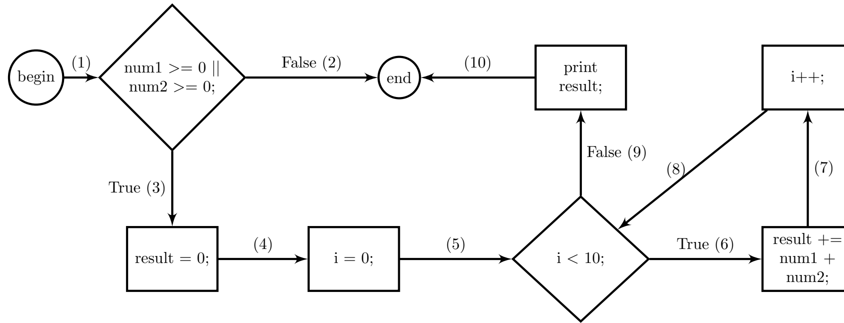


Figure 3.1: Flow graph of code example from Listing 3.1

The edges are numbered, so that they later can be referred to. As the flow graph is quite straightforward, we will not further explain the derivation of the flow graph from the code. A flow graph derivation method was introduced earlier in Chapter 2.

Each example in the following sections refers to the test cases depicted in Table 3.1. These test cases are designed for the code example from Listing 3.1. The first column shows the names of the test cases, the second column the parameters that are passed to the *getSum* method. The third column shows the edges that are traversed in the flow graph from Figure 3.1 by each test case.

3.3 Metrics

This section explains the statement, branch, path and basis path coverage metrics, which belong to the category of white-box testing. To the best of our knowledge, no formal definitions of these metrics yet exist. Therefore, we created formal definitions ourselves based on the theoretical explanation and examples shown in [26] and [29].

3.3.1 Statement coverage

Statement coverage requires that all statements in a program are executed.

Definition 3.3.1. Let $G = \langle P, D, L_P, L_D, E_P, E_D \rangle$ be a flow graph. The statement coverage of a set Π of paths through G is the percentage of process nodes $p \in P$ that are executed by Π:

$$\frac{|\{p \in P \mid \exists \pi \in \Pi \,.\, p \in \pi\}|}{|P|}. \tag{3.1}$$

A shortcoming of statement coverage is that it does not take into account the control flows in the code. To illustrate this we introduce test suite $T_1 = \langle t_2 \rangle$.

Test case	Parameters	Traversed edges
t_1	(num1 = -1, num2 = 1)	$\langle 1, 2 \rangle$
t_2	(num1 = 1, num2 = 1)	$\langle 1, 3, 4, 5, 6, 7, 8, 9, 10 \rangle$

Table 3.1: Available test cases

Note that t_2 is depicted in Table 3.1. While every statement of the program is executed by T_1 (therefore achieving a 100 percent statement coverage), edge 2 is never traversed.

In complex software it is often infeasible to reach a 100 percent statement coverage. A cause of this may be the presence of unreachable code, e.g., code that is only executed on certain operating systems or hardware. The tester should be aware of such cases, as it prevents testing certain code blocks which in fact may be executed once the system under test goes in production. This may happen for instance if the hardware used in production is not available during testing.

3.3.2 Branch coverage

Branch coverage, also referred to as decision coverage, takes into account the control flows of a program. Every decision of the program must be evaluated at least once to true and once to false, e.g., edges $\langle 2, 3, 6, 9\rangle$ in Figure 3.1 should all be traversed. Note that this in fact implies that *all* edges, and therefore also all statements, are visited at least once.

Definition 3.3.2. Let $G = \langle P, D, L_P, L_D, E_P, E_D\rangle$ be a flow graph.

- A *branch* in G is a tuple $\langle d, o\rangle$, consisting of a decision node $d \in D$, and a possible outcome $o \in \{\text{true, false}\}$.
- A branch $b = \langle d, o\rangle$ is *matched* by a path $\pi \in \Pi$ that traverses d, such that $L_D(d) = o$ at the moment that π reaches d. If so, we say that matched(b, π)=true.
- Let B be the set of all branches in G. The *branch coverage* of a set Π of paths through G is the percentage of branches $b \in B$ that is matched:

$$\frac{|\{\text{b} \in B \mid \text{matched(b,}\pi\text{)}\}|}{|B|} \tag{3.2}$$

Test suite $T_2 = \langle t_1, t_2\rangle$ achieves a 100 percent branch coverage for the code example from Listing 3.1. If we would remove test case t_2 from T_2, we will no longer achieve a 100 percent branch coverage, but only 25 percent. Test case t_1 only traverses edge 2, therefore edges 3, 6 and 9 are left untested. If we would remove test case t_1, and stick with t_2, we reach 75 percent (as edge 2 will be the only one that is not traversed).

When a program contains a 'while(true)' loop, it can never evaluate to false. Therefore, it is likely that a 100 percent branch coverage cannot be achieved for an entire system. The tester should take notice of such events, as it decreases the coverage percentage and may give the tester an unfair idea of its test suite quality.

3.3.3 Path coverage

Path coverage is obtained by executing every possible path (Chapter 2, Definition 2.3.3) in a program.

Definition 3.3.3. Let $G = \langle P, D, L_P, L_D, E_P, E_D\rangle$ be a flow graph.

- When a path π is executed we say that executed(π) = true. Otherwise, executed(π) = false.
- The set of all paths through G is denoted by P_G. The path coverage of a set Π of paths through G is the percentage of all paths $\pi \in P_G$ that are executed:

$$\frac{|\{\pi \in P_G \mid \text{executed}(\pi)\}|}{|P_G|} \tag{3.3}$$

Let $T_3 = \langle t_1, t_2 \rangle$. Table 3.1 shows the edges traversed by each test case. At first sight one may argue that T_3 achieves a 100 percent path coverage. Obviously, when looking at the code, we know that the for-loop has 10 repetitions. However, the number of repetitions in the for-loop cannot be determined during execution. Therefore, the number of paths is infinite. As a result, the actual path coverage of T_3 is 0 percent.

3.3.4 Basis path coverage

Basis path coverage is a hybrid between path coverage and branch coverage. However, it partially ignores the loops in a software program by executing only one iteration. Moreover, instead of simply traversing every path, it only traverses independent paths. An *independent path* is any path that tests a decision independently of other decisions, i.e., it shows the influence of changing one particular decision at a time. This implies that basis path coverage subsumes statement and decision coverage. To find the independent paths in a program, McCabe developed a procedure called the baseline method [30]. The method starts at the begin node of a flow graph, and follows the leftmost path until the end node is reached. This is the first independent path (also referred to as the baseline). The second step is to repeat this procedure, but flip the first decision, resulting in a different path. The second step is then repeated until all subsequent decisions have been flipped. Note that with every step, only one decision is flipped. All other decisions stay in their original state (the state in step 1).

Definition 3.3.4. Let $G = \langle P, D, L_P, L_D, E_P, E_D \rangle$ be a flow graph, and P_G the set of all paths in G. The set of all decision nodes in a path $\pi \in P_G$ is denoted as D_π. The baseline of G is found as follows:

$$\text{Baseline}(P_G) = \{\pi \in P_G \mid \forall d \in D_\pi \,.\, L_D(d) = \text{true}\}$$

The basis path coverage of a set of paths Π is the percentage of all independent paths executed including the baseline:

$$\frac{|\{\pi \in \Pi \mid \exists! d \in D_\pi \,.\, L_D(d) = \text{false}\}| + |\text{Baseline}(\Pi)|}{|\{\pi \in P_G \mid \exists! d \in D_\pi \,.\, L_D(d) = \text{false}\}| + 1} \tag{3.4}$$

The number of test cases required to conform to basis path coverage can be computed with the use of McCabe's cyclomatic complexity metric [30].

In the following example we show how to use the baseline method:

Example 3.3.5. This example concentrates on the code example shown in Listing 3.2. The *returnParameter* function should return the original value that is assigned to the *num* parameter. Obviously, the boolean parameters are

Listing 3.2: Basis path coverage

```
1 int returnParameter(int num, boolean d1, boolean d2, boolean d3) {
2         if(d1)
3                 num++;
4         if(d2)
5                 num--;
6         if(d3)
7                 num=num;
8         return num;
9 }
```

used to manipulate the method's control flow. Under certain circumstances, the method returns a different value than expected. For example, if *d1* = true and *d2* = false (*d3* can be ignored), the result would increase the parameter *num* by one.

When using the baseline method, the first step results in *d1*, *d2* and *d3* to evaluate to true, which we abbreviate to TTT. The descendant steps flip *d1*, *d2* and *d3* each independently, resulting in the following set of independent paths: TTT, FTT, TFT, TTF. To achieve full basis path coverage, all four paths should be traversed.

CHAPTER 4

Related work

This chapter introduces several related studies on the relation between test coverage and software quality. Each study is empirical, therefore the exact relation still has to be confirmed. A detailed description of each experiment approach is given, followed by the results.

Organization of this chapter Section 4.1 describes one of the earliest investigations on the relationship between various coverage metrics and software reliability. Then, Section 4.2 describes an experiment that investigates the ability of the decision/all-uses metrics to find faults. Section 4.3 describes an experiment that applied mutant testing techniques to investigate the ability of block coverage, branch coverage, c-use and p-use coverage to find software faults. Section 4.4 describes an experiment that measures the efficiency of several metrics under different testing profiles. We end this chapter with an conclusion on the related work in Section 4.5.

4.1 Garg, 1994

Garg investigated the coverage-reliability relationship and the sensitivity of reliability to errors in the operational profile [19]. The operational profile can be defined as the context of a program in production, thereby restricting the input domain. The experiment had two major objectives:

- Finding a relation between various notions of coverage and the reliability of software.
- Investigating the sensitivity of reliability to errors in the operational profile.

The coverage-reliability relationship determines to what extent coverage testing improves the reliability of software. Software is considered to be reliable if

the input-output sets correspond to the specification. The exact computation can be found in the original paper.

Knowledge about the operational profile makes it easier to determine the behavior of the end-users. Therefore, more specific test cases can be constructed. When testing is then terminated because of time constraints (or other reasons), the most likely usage scenarios of the system can be tested first [25], as the created test-cases are bound to the input domain.

4.1.1 Experiment design

Notions of coverage that were used are: statement/block, dataflow, p-use [21] and c-use [21] coverage. For each of these notions, its relation to software reliability was measured. For the exact definition of reliability we refer to the original paper [19].

The measurements were conducted on the UNIX utility 'sort', that consists of approximately a thousand lines of code. The operational profile was determined by the following properties: (1) probabilities of usage of each of the command-line options provided by sort, and (2) probability distributions for the size of the input file.

Ten faults were injected in various parts of the program. The injected faults were provided by various graduate students, and none of the faults disrupted the syntactical correctness of the program. The faults that students chose were based on their experience and judgment.

4.1.2 Results

Figure 4.1 shows that the reliability of the 'sort' program shows an approximately linear growth for all notions of coverage. Note that p-uses coverage only needs a coverage percentage of less than 50 percent to achieve an average reliability of 0.9, where block coverage requires 70 percent to achieve the same result.

As the outcome of sensitivity analysis of different operational profiles on reliability is not directly related to our study, these results are omitted.

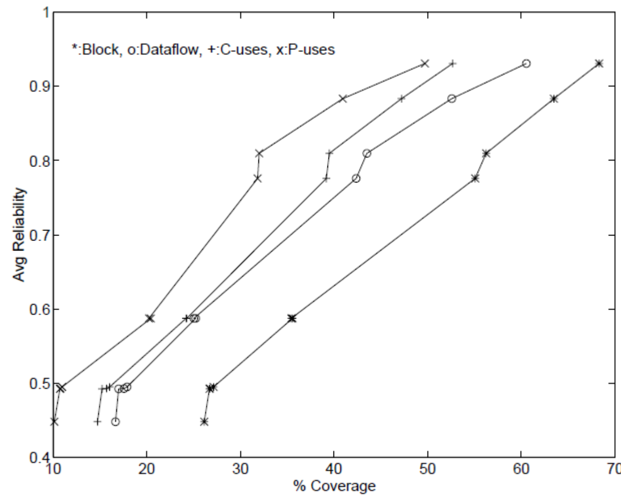
4.2 Frankl and Iakounenko, 1998

Frankl and Iakounenko published the paper 'Further empirical studies of test effectiveness' [18]. They conduced an empirical experiment to investigate the ability of the decision and all-uses [21] coverage criteria to detect software faults. Using a large set of test suites (which were randomly generated), each subject program was thoroughly tested. The relation between coverage and fault-detecting ability was determined by measuring the number of faults found for different percentages of coverage.

4.2.1 Experiment design

The effectiveness of a coverage criterion cannot simply be measured by using one test suite. Test suites that have slightly different test cases can all satisfy some criterion, but have different results. For example: when two test suites both have

Figure 4.1: Garg, 1994, results (taken from [19])



50 percent decision coverage, they may execute completely different decisions in the program. Thus, in defining the effectiveness of a coverage criterion, different test suites are required. To restrict the total number of different test sets that all satisfy the same criterion, the test suite sizes are required to be of equal order.

The programs used for this experiment are eight different versions of an antenna configuration program written for the European Space Agency, consisting of over ten thousand lines of C code. For more specific information about the number of decisions and definition-use (every variable that is defined, should be used at least once [21]) associations in each version, we refer to Figure 4.2.

Faults were found by integration testing and operational use. These faults were corrected, but faulty versions were preserved. Therefore, original faults could be used for this experiment.

Figure 4.2: Frankl and Iakounenko, 1998, Version properties (taken from [18])

subject	decis	unexec decis	duas	unexec duas	failure rate	test set size	number test sets
Version1	1175	353	5255	1437	0.0001	200	10^5
Version3	1175	353	5255	1437	0.0001	200	10^5
Version7	1171	353	5235	1437	0.0150	20	10^6
Version8	1171	353	5235	1437	0.0094	20	10^6
Version12	1175	353	5256	1437	0.0185	100	10^6
Version18	1175	353	5256	1437	0.0014	50	10^6
Version22	1169	352	5198	1411	0.0036	50	10^6
Version32	1175	353	5256	1437	0.0001	100	10^6

4.2.2 Results

For all of the programs that were tested, test sets that attained a high level of decision coverage were significantly more likely to detect the faults than randomly generated test sets of the same size. However, in most subjects, even the high coverage level test sets were not terribly likely to detect the faults. Therefore, the results leave the question open whether the benefits of using such coverage criteria outweigh the testing costs.

Figure 4.3 shows the result of testing version 18 of the program. Most of the other versions showed a somehow similar result. The effectiveness increases drastically around a coverage percentage of 90 percent. However, this percentage is very hard to reach in large and complex programs. For the results of other versions that were tested we refer to the original paper [18].

4.3 Lyu, Huang, Sze and Cai, 2003

Lyu et al conducted an experiment to test the effectiveness of different test methods, i.e., the effectiveness of test methods to find faults [23].

4.3.1 Experiment design

The experiment used a real-world project and engaged multiple programming teams to independently develop program versions based on an industry-scale avionics application. For each program version, the nature, source, type, detectability and effect of faults uncovered were studied.

The study applied mutant testing techniques to reproduce mutants with *real* faults, and investigated the effectiveness of data flow coverage, mutation

Figure 4.3: Frankl and Iakounenko, 1998, Result version 18 (taken from [18])

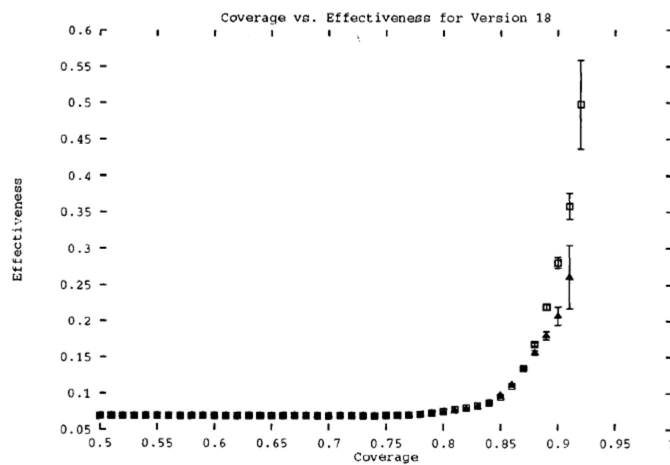


Figure 6: Coverage vs. effectiveness for Version 18. Size 50.

Figure 4.4: Lyu, Huang, Sze and Cai, 2003, Results (taken from [23])

Version ID	Blocks	Decisions	C-Use	P-Use	Any
1	6/11	6/11	6/11	7/11	7/11(63.6%)
2	9/14	9/14	9/14	10/14	10/14(71.4%)
3	4/8	4/8	3/8	4/8	4/8(50.0%)
4	7/13	8/13	8/13	8/13	8/13(61.5%)
5	7/12	7/12	5/12	7/12	7/12(58.3%)
7	5/11	5/11	5/11	5/11	5/11(45.5%)
8	1/9	2/9	2/9	2/9	2/9(22.2%)
9	7/12	7/12	7/12	7/12	7/12(58.3%)
12	10/19	17/19	11/19	17/19	18/19(94.7%)
15	6/18	6/18	6/18	6/18	6/18(33.3%)
17	5/11	5/11	5/11	5/11	5/11(45.5%)
18	5/6	5/6	5/6	5/6	5/6(83.3%)
20	9/11	10/11	8/11	10/11	10/11(90.9%)
22	12/14	12/14	12/14	12/14	12/14(85.7%)
24	5/6	5/6	5/6	5/6	5/6(83.3%)
26	2/11	4/11	4/11	4/11	4/11(36.4%)
27	4/9	5/9	4/9	5/9	5/9(55.6%)
29	10/15	10/15	11/15	10/15	12/15(80.0%)
31	7/15	7/15	7/15	7/15	8/15(53.3%)
32	3/16	4/16	5/16	5/16	5/16(31.3%)
33	7/11	7/11	9/11	10/11	10/11(90.9%)
Overal l	131/252 (60.0%)	145/252 (57.5%)	137/252 (53.4%)	152/252 (60.3%)	155/252 (61.5%)

coverage, and design diversity for fault coverage. In total, 21 program versions and 426 mutants were available. To take a more conservative view in evaluating test coverage, only program versions that at least passed the first test case were considered. This tailored the total number of mutants to 252.

4.3.2 Results

The effectiveness of test coverage in revealing faults is shown in Figure 4.4. Notions of coverage that were used are: block coverage, decision coverage, C-use coverage and P-use coverage.

Columns two to five identify the number of faults in relation to changes of blocks, decisions, c-uses and p-uses. For example, “6/11” for version 1 under the “Blocks” column means that during the evaluation test stage, six out of eleven faults showed the property that when these faults were detected by a test case, block coverage of the code increased. Column six (Any) counts the total number of mutants whose coverage increased in any of the four measures when the mutants were killed.

The results show that out of 252 mutants, 155 show some kind of coverage increase when they were killed. The range for the ratio is between 22.2 percent (found in rightmost column, version ID 8) and 94.7 percent (version ID 12), with an average of 61.5 percent. This is a high ratio, implying that coverage plays an important role in finding faults. However, as the range of the ratio is very wide, it heavily depends on the programmer who ‘created’ the faults.

4.4 Cai and Lyu, 2005

Cai and Lyu published a paper on the effect of code coverage on fault detection under different testing profiles [12]. Previous studies showed that the relationship between code coverage and fault detection is complicated. Cai and Lyu in their paper hypothesized that (1) the effect of code coverage on fault detection varies if different testing profiles are examined and (2) different code coverage metrics may have influence on such relation.

Testing profiles are used to partition test cases by type, e.g., random test cases or functional test cases. This study concentrates on the following testing profiles:

- *Whole test set*: The entire set of test cases.
- *Functional tests*: Test cases that specifically test the functional requirements.
- *Random tests*: Random test cases.
- *Normal tests*: Test cases that test the most common operations.
- *Exceptional tests*: Test cases that test exceptional operations, i.e., not commonly executed operations.

4.4.1 Experiment design

The experiment uses the RSDIMU avionics application [23]. The application was part of the navigation system in an aircraft or spacecraft, and was first engaged in [15] for a NASA-sponsored 4-university multi-version software project. The faults were introduced using mutation techniques.

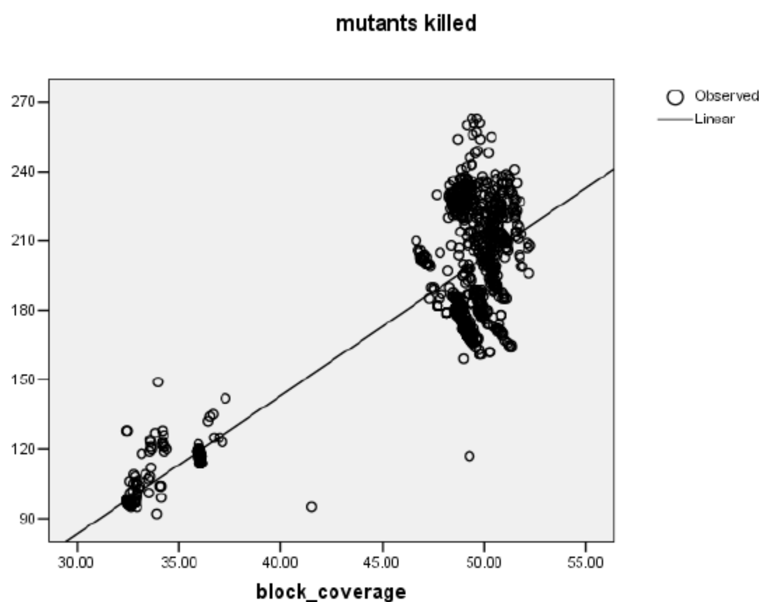
Based on the detected software faults, 21 program versions were selected and 426 software mutants were used [23], which were then exposed to coverage testing and mutation testing. Notions of coverage that were used are block, decision, C-use and P-use coverage. However, as every notion of coverage yielded almost exactly the same results, we depict the results of block coverage and omit the results of other notions.

4.4.2 Results

The experiment measured the block coverage of each test case on the whole set of mutants. The same was done for mutant coverage. The combination of these results shows that a test suite with a higher coverage percentage is likely to kill more mutants, i.e., find more faults. However, a different phenomenon can be observed when viewing the result of combining the two clusters. Figure 4.5 shows this phenomenon. Around 35 percent block coverage and 90-150 mutant coverage we see a lot of mutants were killed. At 50 percent block coverage and 150-270 mutant coverage we can observe the same massive kill of mutants. Exact details of the analysis are omitted, but can be found in the original paper [12].

The reason behind this phenomenon can be interpreted as the result of different testing profiles. After combining the results with the earlier defined testing profiles, we obtain the result shown in Table 4.1. From this result we can observe the effectiveness of coverage on each profile, e.g., the whole test set has

Figure 4.5: Cai and Lyu, 2005, Test case contribution of block coverage (taken from [12])



an effectiveness rate (R^2) of 0.781. More interesting is the result of test cases that fall into the exceptional test profile, as the rate is extremely high (0.944). Test cases that fall into the normal test profile seem to find the least amount of faults, with a rate of 0.045. Therefore, the testing profile plays an important role in the selection of test cases.

4.5 Conclusion

Although several empirical studies have been conducted to find a relation between several notions of coverage and software quality [12, 18, 19, 23], the results seem incoherent. This can be partially justified by different experiment designs, but still a consistent relation seems to be lacking.

An important difference among the experiment design of related studies, is the method used to introduce faults to the software. In [19], several graduate

Testing profile(size)	R^2
Whole test set(1200)	0.781
Functional test(800)	0.837
Random test(400)	0.558
Normal test(827)	0.045
Exceptional test(373)	0.944

Table 4.1: Profile results

students introduced faults to the ‘sort’ application, based on their own discretion. In an experiment of Cai et al [23], mutants were created that individually contained a small number of faults, introduced according to mutant testing techniques. The same technique was later used in another experiment of Cai et al [12]. An experiment conducted by Frankl et al [18] seems to be unique, as it used the original faults that were discovered during development and real-life usage of a software application.

CHAPTER 5

Approach

This chapter explains the approach of the experiment and how we intent to find answers for the research questions stated in Chapter 1. Moreover, it shows how we ensured that the results are reliable, by discussing important details in the experiment design and introducing the real life case studies that were used.

Organization of this chapter Section 5.1 gives an introduction to the experiment design, laying out the basic steps that were followed. Section 5.2 explains the process in more detail using a graphical representation. Then, Section 5.3 describes how we used the tooling. Section 5.4 shows how the results were analyzed. Finally, in Section 5.5 we describe the case studies that were used.

5.1 Introduction

To investigate the effects that statement, branch and basis path coverage have on software quality, our first step is to create test suites. These test suites serve as an input for one or more coverage tools. The second step is to use these coverage tools to measure the coverage percentage for the statement, branch and basis path metrics (see Chapter 3). The third step is observing the number of faults found by each test suite, measured by a unit-testing framework. In the last step the results are analyzed by combining the coverage measures with their corresponding *fault percentage*. The fault percentage is the number of faults found divided by the total number of injected faults, multiplied by a hundred. This provides us an overview of the effectiveness of several different coverage percentages for each of the metrics. Finally, we provide a statistical analyses by calculating the Pearson correlation coefficient on the list of coverage percentages and their corresponding number of faults found.

These steps allow investigating to what extent the requirements of a metric improve the quality of a test suite (also referred to as a metrics efficiency). This

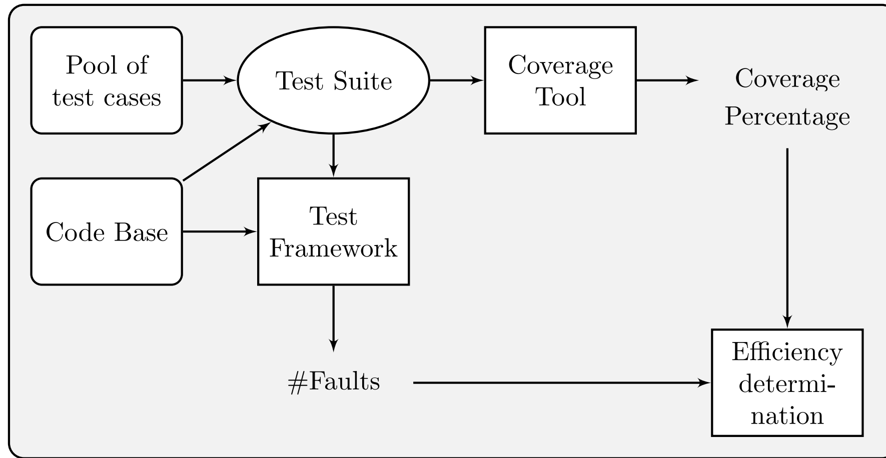


Figure 5.1: Experiment design

is investigated for several coverage percentages, e.g., is 60 percent statement coverage as efficient as 70 percent statement coverage? Also, we can investigate if one metric is more efficient than another. And finally, we can investigate the effort needed to achieve a certain coverage percentage (more on effort estimation can be found in Section 5.4). The outcome of these investigations helps us answering the research questions defined in Chapter 1.

5.2 Design

To improve the clarity of the experiment design, we introduce a graphical representation of the process in Figure 5.1. During our experiment, we used three different projects (see Section 5.5), each having its own unique set of test cases. The process graphically presented here was repeated for each individual test suite that we created, thus several times per project.

We now explain each block in an individual paragraph, providing details on their composition and role in the overall process.

Code base The code base represents the entire collection of source code of a system under test. As mentioned earlier, we have three projects. Consequently, we have three individual code bases.

Each code base is injected with synthetic faults. To inject faults that are as true-to-nature as possible, a group of developers was selected that had extensive experience with the code base. This resulted in a total of three different groups, one for each code base. Note that the developers of these groups have limited knowledge of the test cases designed for the projects, as this may influence their objectivity when inserting faults.

Pool of test cases The pool of test cases is the entire set of test cases available to test a code base. This pool partially consists of test cases created by a team of testers from Info Support (see Section 1.4.1), that used the test-driven development methodology [10]. The pool of test cases was complemented by

additional test cases that were created by another tester. These additional test cases were necessary to achieve higher coverage percentages.

Code bases written in the Java language, were tested by test cases that use the JUnit Framework [4]. Code bases written in the C# language by the NUnit Framework [6].

Test suite The test suite is a finite set of test cases that is formed by taking a subset of the pool of test cases. This selection is done by handpicking test cases from the test case pool, using the following guidelines: (1) Complex test cases are interspersed with simpler test cases. (2) Complex parts of the system are generally tested by a greater number of test cases than simpler parts of the system. (3) Test cases created by the Info Support test team are preferred above the additional added test cases. The coverage percentage is a calculated average for the entire project. However, each test suite must be designed to equally distribute the coverage percentage across each part of the system as much as possible. Obviously, due the nature of software systems, deviations do occur.

Each test suite is created to achieve a certain coverage percentage for a specific metric. The chosen metrics and coverage percentages (which can be found in Table 5.2) vary to investigate the efficiency on different coverage percentages (helping us answering SQ2, found in Chapter 1). As the desired coverage percentages may not be reached exactly due to the size and complexity of the case studies used, we allow a margin of 2.5 percent.

Coverage tool A coverage tool is used to measure the coverage percentages for each metric that is achieved by a test suite. In Table 5.1 we provide a list of coverage tooling that we used in our experiment. It shows that for the Java framework, the Codign Coview [1] tool supports the branch and basis path coverage metrics and Eclemma [2] supports statement coverage. NUnit [6] is combined with NCover [5], supporting both the statement and branch coverage metrics. By conducting small tests we validated that these tools use the metrics as we formally defined them in Chapter 3.

Coverage Percentage The coverage percentage is the result of a coverage tool. It presents, in the form a percentage, to what extent the test suite meets the requirements of a certain coverage metric. As can be found in Table 5.1, some tools support multiple metrics. In such a case the tool reports the coverage percentage for each metric. For example, Codign Coview may report that a test suite T_1 has a branch coverage percentage of 75 percent, but also has a statement coverage percentage of 70 percent.

Test Framework The test framework executes the test suites and reports the number of failed test cases. It allows us to observe the number of faults found by each test suite individually. We use the JUnit test framework for projects developed in the Java framework, and the NUnit test framework for projects developed in the .NET framework.

 # Faults The number of faults (written as #Faults) is the number of injected synthetic faults that are exposed by a test suite. Note that a test framework

Tool name	Framework	Statement	Branch	Basis Path
Codign Coview	Java	No	Yes	Yes
Eclemma	Java	Yes	No	No
NUnit/NCover	.NET	Yes	Yes	No

Table 5.1: Used tooling

returns the total number of test cases that fail. This may be different than the number of injected synthetic faults it exposes, as multiple test cases may fail due to only one fault. We manually analyzed the failed test cases, and documented the number of synthetic faults that were found by these test cases.

Efficiency determination Merging the coverage percentage and the number of faults found, we can determine the efficiency of the test suite. For example, 70 percent statement coverage found 6 out of a total of 10 faults. Therefore, the efficiency of 70 percent statement coverage is 60 percent. Repeating this step for the other metrics (and coverage percentages), we can determine the relation between the coverage metrics and software quality (helping us answering SQ1, found in Chapter 1).

5.2.1 More on test suite creation

When taking a closer look at Table 5.2, one may observe that for each metric, the coverage percentages vary from the other metrics. This is due the fact that one metric is stronger than another, and as a consequence it may be relatively easy to reach a certain coverage percentage with one metric, but impossible with another metric. In general, we picked these percentages by evaluating their diversity and practical feasibility.

To increase the quality of our experiment result, we have the following set of additional requirements during test suite creation:

- To increase the reliability of the result, we require that each coverage percentage is tested by at least two test suites per case study. We require the test suites to have a different composition. The degree of difference in composition is determined per coverage percentage, and depends on the available test cases. We aim at maximizing the difference in composition (by minimizing the size of the intersection set of the test suites), provided that the test suites do not loose plausibility and conform to the requirements from Section 5.2. The creation of two individual test suites reduces the chance that test suites (especially those with a low coverage percentage) expose or miss an exceptional number of faults by coincidence. For example, a test suite that achieves 50 percent branch coverage may

	Percentage levels				
Statement	50%	60%	75%	85%	95%
Branch	30%	50%	60%	75%	85%
Basis path	10%	15%	20%	25%	30%

Table 5.2: Coverage metrics and corresponding percentages being tested

expose zero faults if all faults coincidentally occur in the other 50 percent of branches.

- The test suite size should be of equal order for each case study. This is due to the fact that test suite size may influence its efficiency [32]. Moreover, once a coverage percentage is reached by a test suite, we stop adding test cases. As in practice, people aim at achieving a certain coverage percentage, and then immediately cease creating tests. Note that adding additional test cases is possible without increasing the coverage percentage, by testing same parts of the system with different input variables, e.g., boundary-value analysis.

5.3 Execution

The execution of the experiment required some development tools. For the Java case studies, the Eclipse framework (version 3.5) was used, including the EclEmma and Codign Coview plugins. These plugins both work out of the box after installing them using the Eclipse software update function. Information and documentation for installing plugins can be found at the Eclipse documentation site [3]. Additional documentation on using these plugins can be found at their websites [1, 2].

The .NET case study was developed using Visual Studio 2008. Additional tooling used is NUnit and NCover. NUnit is used as a library within the Visual Studio project to create test cases. After compiling the test suite containing the test cases, a DLL file is created. This file can be imported by the NCover tool, which executes the test suite and determines the coverage percentage reached by the test suite. More information on using NUnit and NCover can be found at [6, 5].

5.4 Analysis

Our analysis starts by documenting the number of faults that are found by each coverage percentage, resulting in an *efficiency prediction table*. The efficiency prediction table presents the relation between the coverage metrics and software quality. In other words, it shows the efficiency of each coverage metric under different coverage percentages .To prove in a statistically robust manner that indeed a relation between coverage metrics and software quality exists, we use the Pearson correlation coefficient [13]. The Pearson correlation coefficient determines if, and to what extent, a linear relation exists between two variables.

Definition 5.4.1. Let X be the set of coverage percentages tested, and Y be the corresponding percentages of faults found (faults found divided by the total number of faults). The correlation coefficient r_{XY} is given by:

$$r_{XY} = \frac{\sum X_i Y_i - n\bar{X}\bar{Y}}{\sqrt{(\sum X_i^2 - n\bar{X}^2)(\sum Y_i^2 - n\bar{Y}^2)}} \tag{5.1}$$

Where X_i is the value of the x-axis at measurement i, Y_i is the value of the y-axis at measurement i, n is the number of measurements, $\bar{X}$ is the average of all X values and $\bar{Y}$ is the average of all Y values.

The result of equation 5.1 is a number between -1.00 and +1.00, representing the degree of correlation. A positive value implies a positive correlation, i.e., large values of X tend to be associated with large values of Y. A negative value implies an inverse correlation, i.e., large values of X tend to be associated with small values of Y.

An important factor that should be kept in mind when interpreting these correlation coefficients, is the fact that the selection process of test cases was not completely random, and therefore is less reliable. The test case pools we used in the industrial case studies, do not reflect random samples of the total set of test cases used in the software industry. However, we do consider it to reflect a random sample of test case pools used at Info Support. As Info Support uses the test-driven development methodology to create their test cases, we also consider the test case pool to reflect (although in lesser extent) a random sample for every test case pool in the software industry, as long as it is based on the test-driven development methodology.

In addition to calculating the correlation coefficients, we calculate the confidence interval for each coefficient using the Fisher transformation [17]. This widely known calculation provides details on the certainty that the calculated correlation coefficient is not based on coincidence. We use this calculation to obtain an interval for which we can state with 95 percent certainty that the actual correlation coefficient will fall in.

Finally, the research questions (see Chapter 1) require us to estimate the effort that was spent to create each test suite. To provide a clear and consistent calculation for this effort, we introduce the test effort metric.

Definition 5.4.2. Let N be the total number of test cases, F_i the number of functions tested by test case i, and C_i the maximum cyclomatic complexity [30] of the function(s) tested by test case i. The *test effort* E_T to create test suite T is:

$$E_T = N + \sum_{i=1}^{N} F_i + 3 \sum_{i=1}^{N} C_i \tag{5.2}$$

Note that the test effort metric does not calculate the man hours spent to create the test suite. It can only be used as a reference model to calculate relative differences in effort among test suites. The formula was derived by discussing with experienced testers from Info Support. Therefore, the equation represents only the effort required specifically by their testers. It is not intended as a general metric for the software industry.

5.5 Case studies

As stated earlier, we used three industrial cases studies in our experiment. These case studies were selected by their mutual variance, increasing the diversity of the experiment. This diversity helps us in determining the reliability of the result, as the results of each case study may confirm or contradict the results from another case study.

This section provides an introduction to each case study, by describing its most important goals, identifying its size and the platform in which it was developed.

5.5.1 Integrity Service

This case study involves an integrity service for seven different payment requests, e.g. paying a bill with internet banking. Whenever a payment request is created, the integrity service creates a unique digital signature for the payment based on its internal information. Once the payment is signed, it transfers through several complex systems, mostly ending at the Society for Worldwide Interbank Financial Telecommunication (SWIFT). At this last station before the payment is executed, the integrity service is called to test whether the message has been altered.

Project size: 10.000 LOC
Average cyclomatic complexity: ≈ 1.32
Used technology: Java 1.5, Spring Core 2.5, Spring Web Services 1.5, JiBX 1.2, Spring JDBC.

5.5.2 Insurance Administration

This case study involves an application for a major dutch insurance company. The application is a portal that enables the employees to sell policies, view the financial state of customers, add and view the history of the employees, create tenders etc.

Also, a Service-Oriented Architecture exists that provides services which realize the communication to external parties, integration with standard packages, and persistence of customer related data. Therefore, the application is considered a business-critical system.

The system consists of approximately hundred web services and front-end components.

Project size: 25.000 LOC
Average cyclomatic complexity: ≈ 1.41
Used technology: .NET 3.5, Web Services.

5.5.3 Hosted Business Suite

Besides developing custom software, Info Support offers a hosted business suite for customers. A hosted business suite is a package of online software that can be used by the customer, which is hosted by Info Support. For example: Hosted Exchange Online Server, Windows Sharepoint Services and CRM.

Customers who use the Hosted Business Suite may act as resellers of this software. This results in a tree structure of customers. Each customer can order specific demands for the software, i.e., the software can be customized.

The Billing project partially automates the process of calculating the costs (that should be billed to the customer), and integrates this data with the information available about the customer in the CRM system of Info Support. The end result is a complete bill, specifying the customer, used software and costs.

Project size: 30.000 LOC
Average cyclomatic complexity: ≈ 1.9
Used technology: Java 1.6, JSF, Cobertura, Easymock.

CHAPTER 6

Results and discussion

This chapter provides and discusses the results of the experiment, by providing scatter charts and statistic calculations for each case study. The first case study is discussed in greater detail than case studies two and three, avoiding discussing almost identical results and/or showing simple calculations repetitiously. The chapter ends with an overview that provides the correlations and costs for each case study. Also, we provide charts in which the costs and fault percentages are combined. Finally, we provide charts showing the combined results of all case studies and provide a general evaluation of the results.

Organization of this chapter Section 6.1 provides the results of the first case study, including calculations of the correlation coefficients and the test effort metric. In Section 6.2, we provide the results of the second case study, followed by the results of the third case study in Section 6.3. Then, Section 6.4 shows the costs of achieving the coverage percentages of each metric. Finally, Section 6.5 provides an evaluation of the results.

6.1 Case study 1

6.1.1 Statement coverage

Figure 6.1 shows the results for the statement coverage metric. The x-axis represents the coverage percentage, and the y-axis represents the fault percentage. According to the experiment design, we used two different test suites for each tested coverage percentage, resulting in a total of 6 series (2 for each case study).

The scatter chart shows us that the test suite from series 1 that achieved 60 percent statement coverage, found 15 percent of all faults. A test suite from series 2 that achieved the same percentage of statement coverage, found 25 percent of all faults. The difference in fault percentages is likely caused by the greater difference in composition that test suites with a low coverage

percentage have (see Chapter 5, Section 5.2). The difference among the two series diminishes when the coverage percentage grows.

Correlation The Pearson correlation coefficient is calculated using the results provided in Figure 6.1. An explanation of the variables used in this formula can be found in Section 5.4.

$$
\begin{aligned}
r_{XY} &= \frac{\sum X_i Y_i - n\bar{X}\bar{Y}}{\sqrt{(\sum X_i^2 - n\bar{X}^2)(\sum Y_i^2 - n\bar{Y}^2)}} \\
\sum X_i Y_i - n\bar{X}\bar{Y} &= ((50 \cdot 10) + (50 \cdot 10) + (60 \cdot 15) \\
&+ (60 \cdot 25) + (70 \cdot 20) + (70 \cdot 25) \\
&+ (80 \cdot 30) + (80 \cdot 30) + (90 \cdot 40) \\
&+ (90 \cdot 40)) - (10 \cdot 70 \cdot 24.5) \\
&= 18550 - 17150 = 1400. \\
(\sum X_i^2 - n\bar{X}^2)(\sum Y_i^2 - n\bar{Y}^2) &= (2(50^2 + 60^2 + 70^2 + 80^2 + 90^2) \\
&- (10 \cdot 70^2)) \cdot ((10^2 + 10^2 + 15^2 + 25^2 \\
&+ 25^2 + 20^2 + 30^2 + 30^2 + 40^2 + 40^2) \\
&- (10 \cdot 24.5^2)) \\
&= (51000 - 49000) \cdot (7075 - 6002.50) \\
&= 2145000. \\
\sqrt{(\sum X_i^2 - n\bar{X}^2)(\sum Y_i^2 - n\bar{Y}^2)} &= \sqrt{2145000} \approx 1464.58. \\
r_{XY} = \frac{\sum X_i Y_i - n\bar{X}\bar{Y}}{\sqrt{(\sum X_i^2 - n\bar{X}^2)(\sum Y_i^2 - n\bar{Y}^2)}} &= \frac{1400}{1464.58} \approx +0.96.
\end{aligned}
$$

The correlation coefficient of approximately 0.96 indicates a strong positive correlation between the coverage percentage and the fault percentage. Therefore, it is likely that higher coverage percentages find more faults compared to lower coverage percentages. Moreover, the fault percentage linearly increases as the coverage percentage grows.

We now calculate the confidence interval for the above calculated correlation coefficient:

$$
\begin{aligned}
\text{Confidence interval } &= z' \pm z \cdot \sigma_{z'} \\
z &= 1.96 \text{ (95\% certainty)}. \\
z' &= .5[\ln(1+r) - \ln(1-r)] = .5[\ln(1+0.96) - \ln(1-0.96)] = 1.9459. \\
\sigma_{z'} &= \frac{1}{\sqrt{N-3}} = \frac{1}{\sqrt{10-3}} = 0.38.
\end{aligned}
$$

$$
\begin{aligned}
z' \pm z \cdot \sigma_{z'} &= [1.9459 - (1.96 \cdot 0.38), 1.9459 + (1.96 \cdot 0.38)] \\
&= [1.2011, 2.6907].
\end{aligned}
$$

Converting [1.2011, 2.6907] back to r: [0.83, 0,99]

Thus, there is a 95 percent chance that the actual correlation coefficient is in the interval [0.83, 0.99].

Test suite design effort We now show the calculation of the effort needed to achieve 50 percent statement coverage.

$$
\begin{aligned}
E_T &= N + \sum_{i=1}^{N} F_i + 3\sum_{i=1}^{N} C_i \\
N &= 90 \\
\text{Average } F_i &= 1.00 \\
\sum_{i=1}^{N} F_i &= 90 \cdot 1.00 = 90. \\
\text{Average } C_i &\approx 2.205 \\
3\sum_{i=1}^{N} C_i &\approx 3 \cdot 90 \cdot 2.205 \approx 595.49. \\
E_T &= N + \sum_{i=1}^{N} F_i + 3\sum_{i=1}^{N} C_i \\
&\approx 90 + 90 + 595.49 \\
&\approx 775.49.
\end{aligned}
$$

Calculating the costs for all measured coverage percentages provides the following overview:

Coverage percentage	50%	60%	70%	80%	90%
Costs (E_T)	775	933	1070	1221	1416

A graphical representation of the costs for each case study and corresponding metrics can be found in Appendix A.

6.1.2 Branch coverage

Figure 6.2 shows the results for the branch coverage metric. Evaluating the result, we observe almost identical behavior as seen earlier with statement coverage. Lower coverage percentages are likely to find a very limited number of faults, e.g., 40 percent of branch coverage found 0 or 10 percent of the total number of founds. Test suites with the highest coverage percentage measured, 85 percent, found 35 percent and 40 percent of all faults.

Interesting is the fact that the test suite from Series 1 (see Figure 6.2) that achieved 50 percent of branch coverage, was able to expose more faults than the test suite from the same series that achieved 60 percent of branch coverage. This could be the effect of luck being involved, which was discussed earlier in Section 5.2.1, Chapter 5.

Correlation Approximately +0.93. Confidence interval: $0.72 < r < 0.98$.
Test suite design effort

Coverage percentage	40%	50%	60%	75%	85%
Costs (E_T)	646	916	1018	1181	1416

6.1.3 Basis path coverage

The efficiency of basis path coverage is shown in Figure 6.3. Basis path coverage is considered to be a stronger metric than statement and branch coverage. The results show us that lower percentages of basis path coverage, found the same amount of faults as higher percentages of coverage for the statement and branch metrics, e.g., 25 percent of basis path coverage found the same amount of faults as 90 percent statement coverage. This confirms the assumption that stronger metrics are likely to find more faults than weaker metrics (See Chapter 3, Section 3.1). In both case studies, a coverage percentage of 30 percent for the basis path metric, achieved a fault percentage between 45 and 55 percent. These are the highest fault percentages measured during the experiment.

Correlation Approximately +0.95. Confidence interval: $0.80 < r < 0.98$.
Test suite design effort

Coverage percentage	10%	15%	20%	25%	30%
Costs (E_T)	897	1027	1356	1440	1472

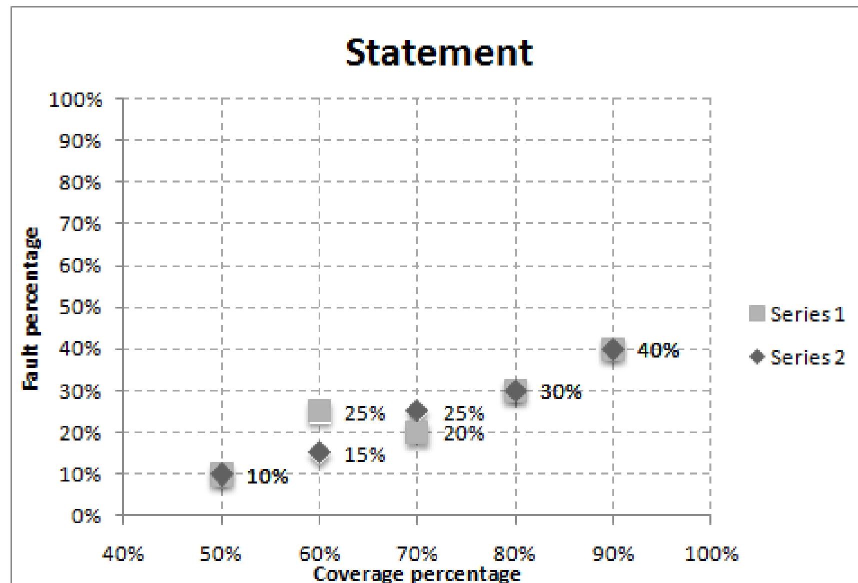


Figure 6.1: Case study 1 - Statement coverage

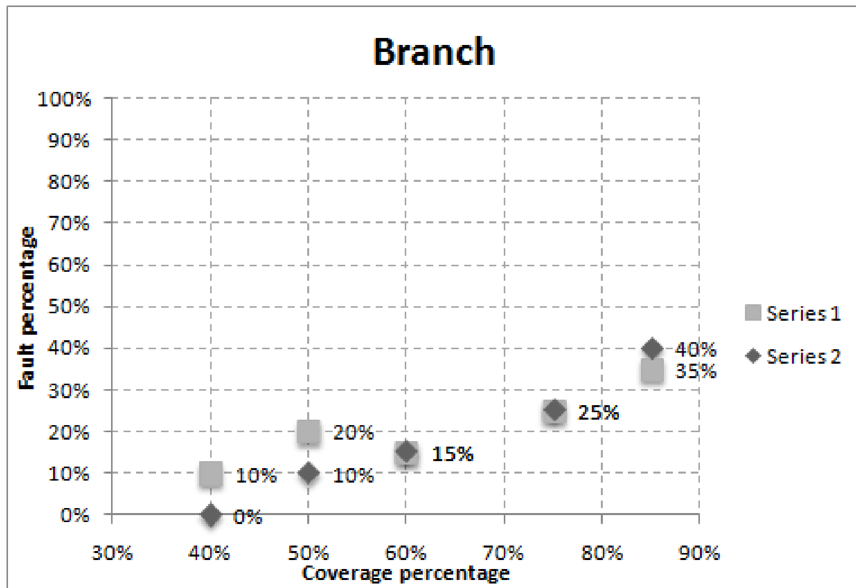


Figure 6.2: Case study 1 - Branch coverage

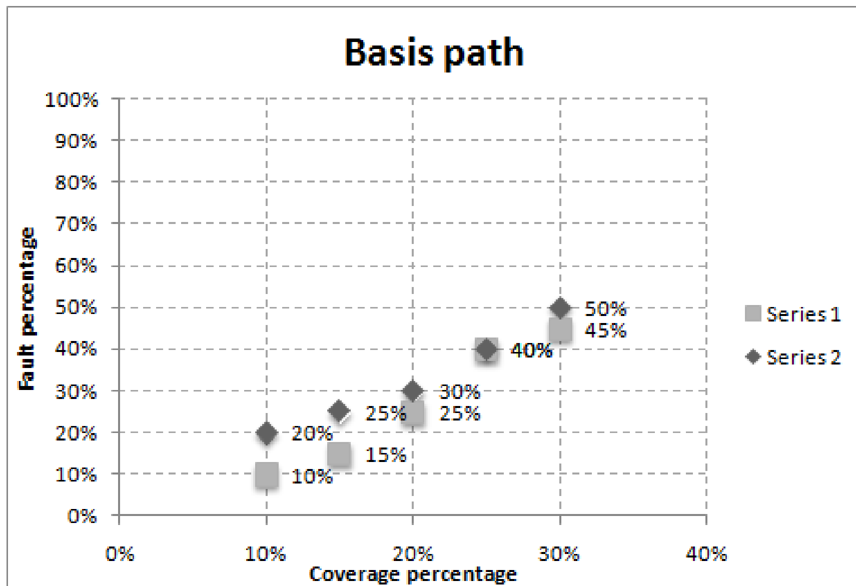


Figure 6.3: Case study 1 - Basis path coverage

6.2 Case study 2

6.2.1 Statement coverage

Figure 6.4 shows the results for the statement coverage metric. Note that the fault percentage does not seem to consistently grow from the moment that a coverage percentage of 70 percent is reached. With the exception of the test suite from Series 1 with 80 percent coverage, all other test suites do not manage to expose more than 33 percent of the injected faults.

Correlation Approximately +0.80. Confidence interval: 0.34 < r < 0.95.
Test suite design effort

Coverage percentage	50%	60%	70%	80%	90%
Costs (E_T)	3323	4104	5069	6084	6749

6.2.2 Branch coverage

The results for the branch coverage metric are depicted in Figure 6.5.

Correlation Approximately +0.96. Confidence interval: 0.83 < r < 0.99.

Test suite design effort

Coverage percentage	40%	50%	60%	75%	85%
Costs (E_T)	2628	3323	4104	5171	6749

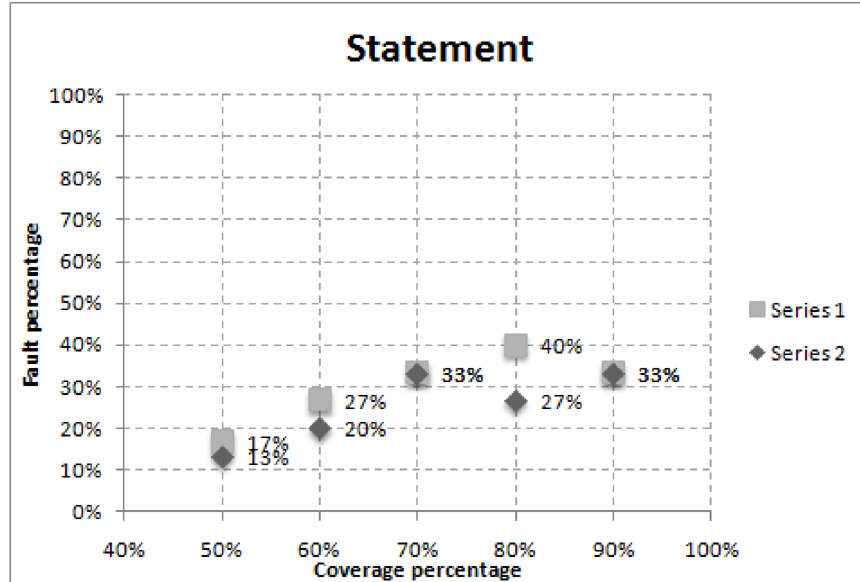


Figure 6.4: Case study 2 - Statement coverage

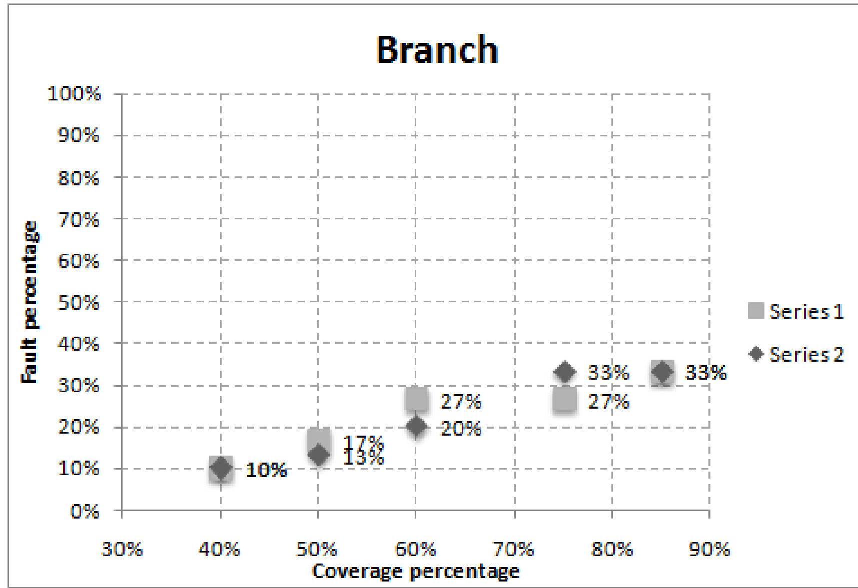


Figure 6.5: Case study 2 - Branch coverage

6.3 Case study 3

6.3.1 Statement coverage

The results of the statement coverage metric are depicted in Figure 6.6. A coverage percentage of at least 80 percent clearly finds more faults than lower coverage percentages. Test suites that achieved the highest measured coverage percentage of 90 percent, were able to consistently find 55 percent of all faults. Compared to case studies 1 and 2, the fault percentage is significantly higher. This is likely caused by the poor quality of the synthetically injected faults, resulting in the faults to be found more easily.

Correlation Approximately +0.84. Confidence interval: $0.44 < r < 0.96$.

Test suite design effort

Coverage percentage	50%	60%	70%	80%	90%
Costs (E_T)	1020	1139	1316	1496	1723

6.3.2 Branch coverage

Figure 6.7 depicts the results from the branch coverage metric. Compared to case studies 1 and 2, we observe the following differences (1) the results are more scattered and (2) the fault percentages are significantly higher. Both (1) and (2) are caused by the simplicity of the injected faults.

Correlation Approximately +0.58. Confidence interval: -0.08 < r < 0.89.

Test suite design effort

Coverage percentage	40%	50%	60%	75%	85%
Costs (E_T)	755	1020	1139	1320	1717

6.3.3 Basis path coverage

Figure 6.8 depicts the results from the basis path coverage metric. The results seem to have a stronger correlation compared to statement and branch coverage. However, a significant deviation occurs at a coverage percentage of 10 percent, resulting in a lower correlation coefficient.

Correlation Approximately +0.56. Confidence interval: -0.11 < r < 0.88.
Test suite design effort

Coverage percentage	10%	15%	20%	25%	30%
Costs (E_T)	1053	1274	1476	1709	1600

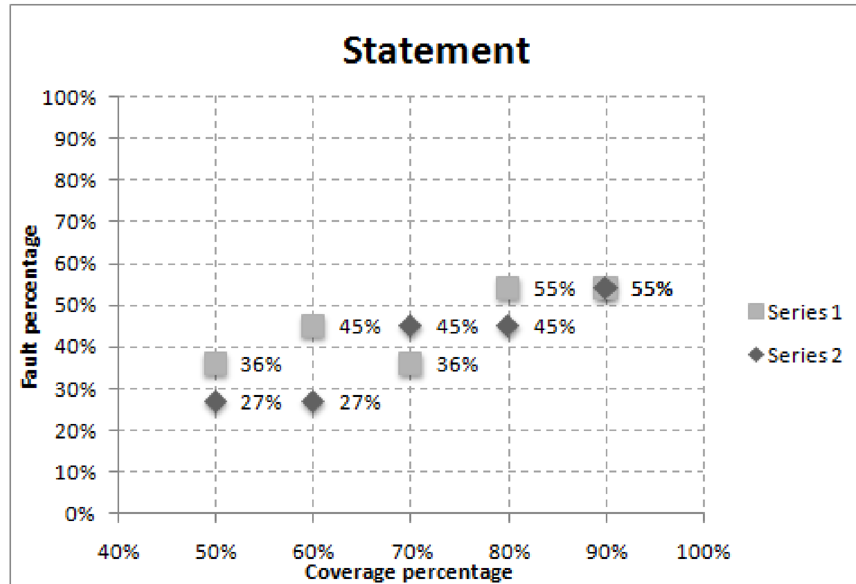


Figure 6.6: Case study 3 - Statement coverage

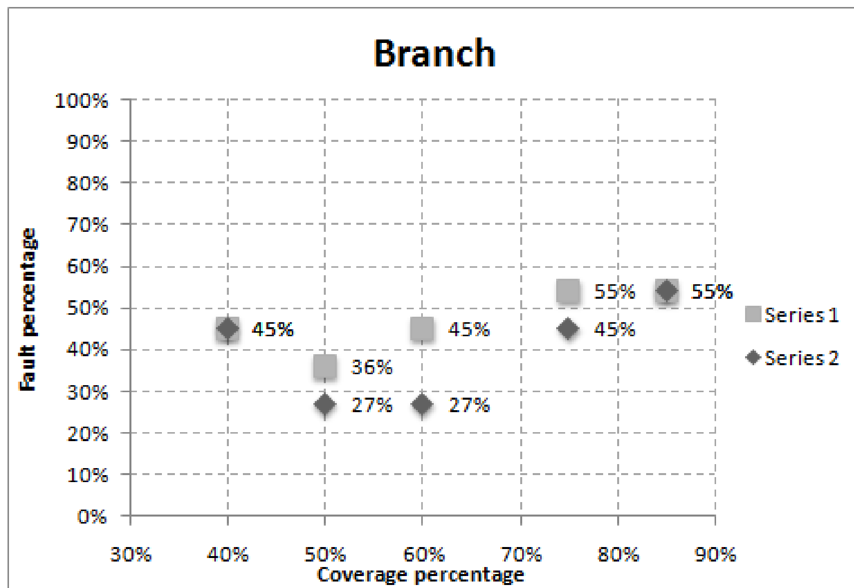


Figure 6.7: Case study 3 - Branch coverage

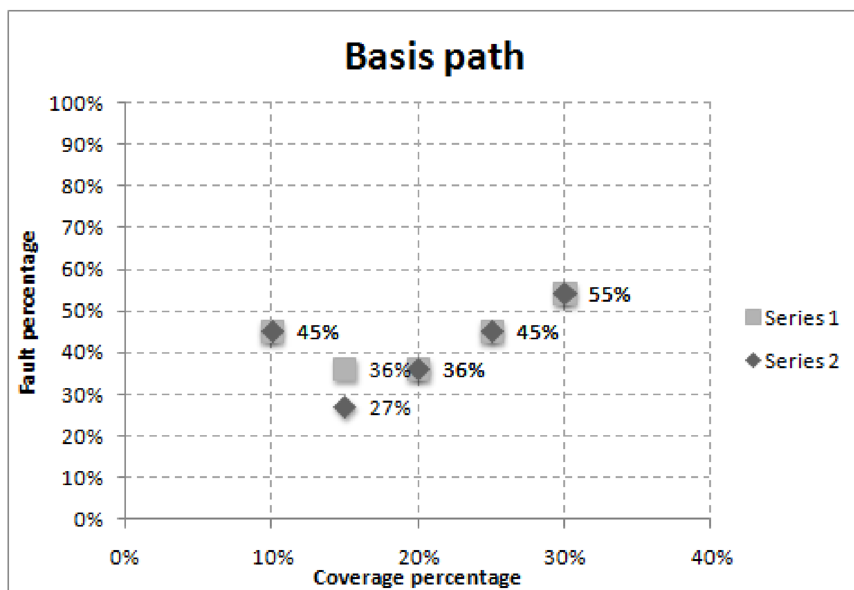


Figure 6.8: Case study 3 - Basis path coverage

6.4 Costs versus fault percentage

Figures 6.9, 6.10 and 6.11 show the test suite design effort (also referred to as *costs*) needed to achieve the different coverage percentages. The depicted costs and fault percentages are averages over three case studies (with the exception of basis path coverage, which was measured in only two case studies). From the linear increase, we can conclude that there is no steepening in costs whatsoever at any percentage. Therefore, the costs of increasing the coverage percentage is approximately the same independent of the current (i.e., already reached) coverage percentage.

The linear growth in costs is likely caused by the low cyclomatic complexity of the case studies, as this reduces the number of different paths to be traversed enormously. As a consequence, reaching higher coverage percentage takes less effort, as we do not have to create a lot of test cases that are specifically designed to flip a certain decision node in the code.

Appendix A provides the results for each case study and metric individually.

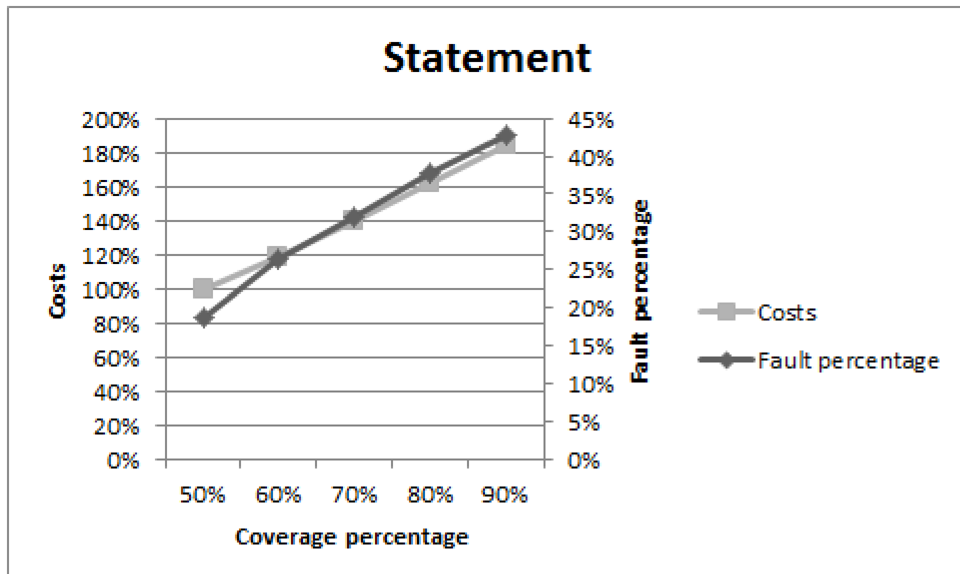


Figure 6.9: Overall costs - Statement coverage

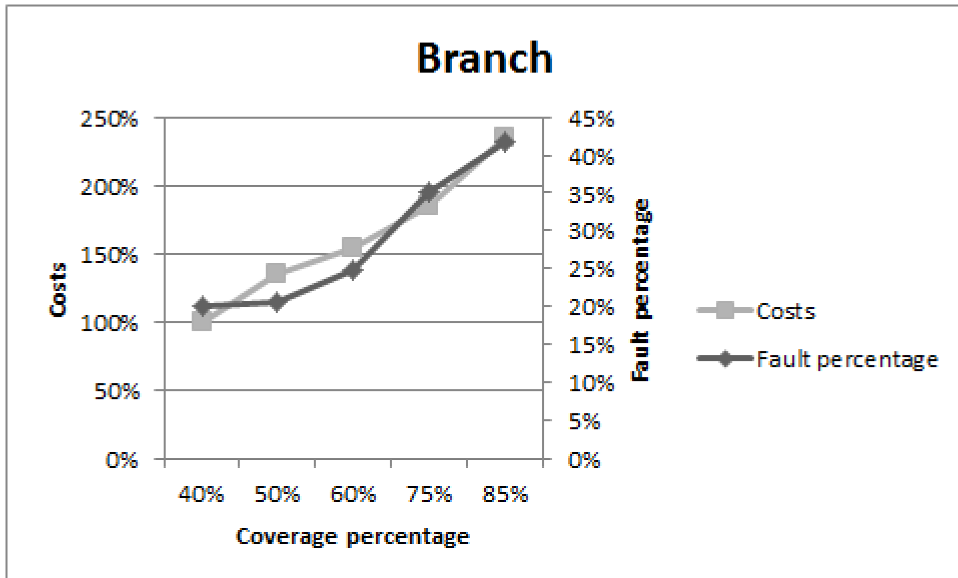


Figure 6.10: Overall costs - Branch coverage

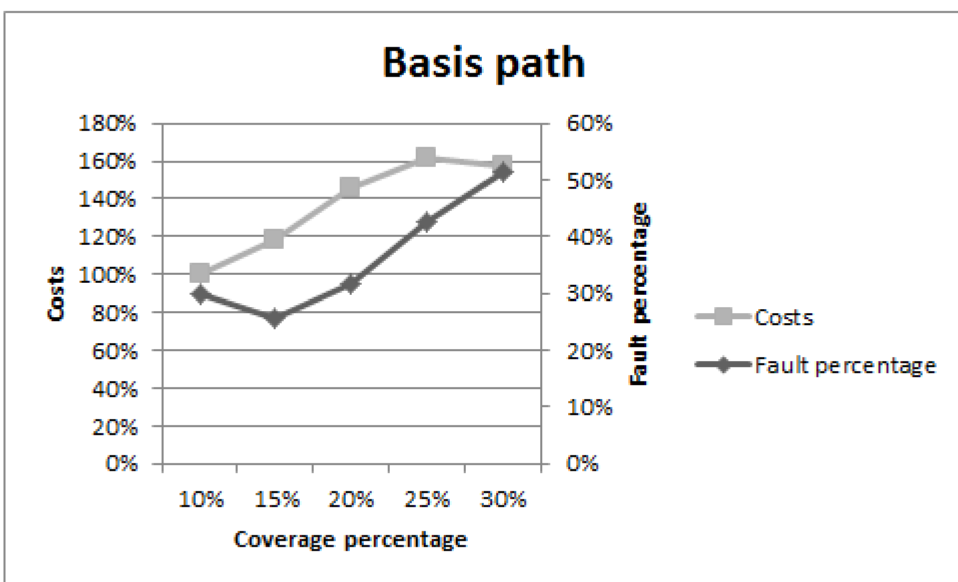


Figure 6.11: Overall costs - Basis path coverage

Case study	Coverage metric	Correlation	Confidence interval (95%)
1	Statement	0.96	0.83 < r < 0.99
	Branch	0.93	0.72 < r < 0.98
	Basis path	0.95	0.80 < r < 0.98
2	Statement	0.80	0.34 < r < 0.95
	Branch	0.96	0.83 < r < 0.99
3	Statement	0.84	0.44 < r < 0.96
	Branch	0.58	-0.08 < r < 0.89
	Basis path	0.56	-0.11 < r < 0.88

Table 6.1: Overview correlations

6.5 Evaluation

6.5.1 Overview correlations

Table 6.1 shows the Pearson correlation coefficient of each case study and corresponding metrics. Case studies one and two have the strongest positive correlation, with a minimum correlation coefficient of 0.80. The correlation of the third case study is exceptionally low for the branch and basis path coverage metric. As discussed in Section 6.3, the smaller coverage percentages of these metrics have likely by luck exposed more faults than some of the higher coverage percentages. The lower correlation coefficient is a direct consequence of this phenomenon. To give an idea of the influence of these deviations on the correlation coefficients and corresponding confidence intervals, table 6.2 provides a recalculation, excluding the deviations. We can observe that when these deviations are excluded, a strong correlation exists.

The confidence interval row shows the interval for the correlation coefficient for which 95 percent certainty is given. For instance, we can state with 95 percent certainty, that the actual correlation coefficient for the statement coverage metric of case study 1 lays in the interval [0.83, 0.99]. For 6 out of 8 measured correlation coefficients, we may conclude that a positive correlation is highly likely to exist (exceptions are branch and basis path coverage from case study 3). However, the degree of correlation remains uncertain, caused by the low number of measurements.

6.5.2 Merged results

Figures 6.12, 6.13 and 6.14 show the overall results for the statement, branch and basis path coverage metrics. Case studies 1 and 2 seem to achieve almost identical results for the statement and branch coverage metric. Test suites from case study 3 achieved a higher fault percentage, but this is caused by the poor quality of injected faults, resulting in these faults to be found more easily. Also,

Case study	Coverage metric	Correlation	Confidence interval (95%)
3	Branch	0.85	0.37 < r < 0.98
	Basis path	0.96	0.79 < r < 0.99

Table 6.2: Recalculated correlations

case studies 1 and 3 have almost identical results for the basis path coverage metric.

Due to the higher fault percentages achieved by case study 3, the range of faults that were found is very wide. Therefore, the results do not give certainty on the exact amount of faults that the metrics and corresponding coverage percentages will find.

6.5.3 Fault detection capability

The results provided in this chapter show that coverage testing was by far not able to find all faults. This raises the questions why some of the faults were found, and others were left undetected . An afterwards inspection of the injected faults, shows that faults that had a direct influence on the normal flow of a program were highly likely to be detected. This is probably a direct consequence of using the test-driven development methodology, as this methodology requires test cases to be written that specifically test the functionality of methods.

Faults which did not directly effect the normal program flow, were left undetected most of the time. An incomplete list of some of these faults is (1) incorrect boundary value checking, (2) not handling null values, (3) not checking for undesired duplicates in lists, (4) forgotten break-statements and (5) forgotten concurrency prevention (e.g. forgetting synchronize keyword). Test methods that would be able to find most of the undetected faults are boundary-value analysis and equivalence partitioning [26].

6.5.4 Threats to validity

One threat to this experiment is the validity of the calculations performed by the coverage tooling. Although we did test if the tools followed the rules of our formal definition by testing small examples, we cannot validate if the calculations are performed correctly on larger projects. Nevertheless, we assume the results to be correct.

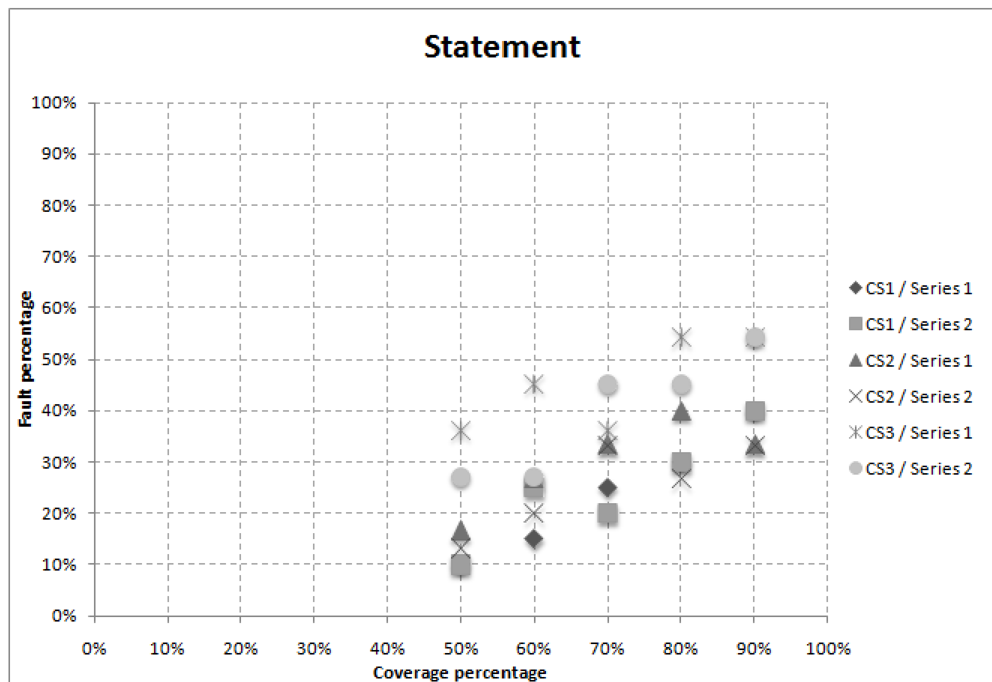


Figure 6.12: Overall result - Statement coverage

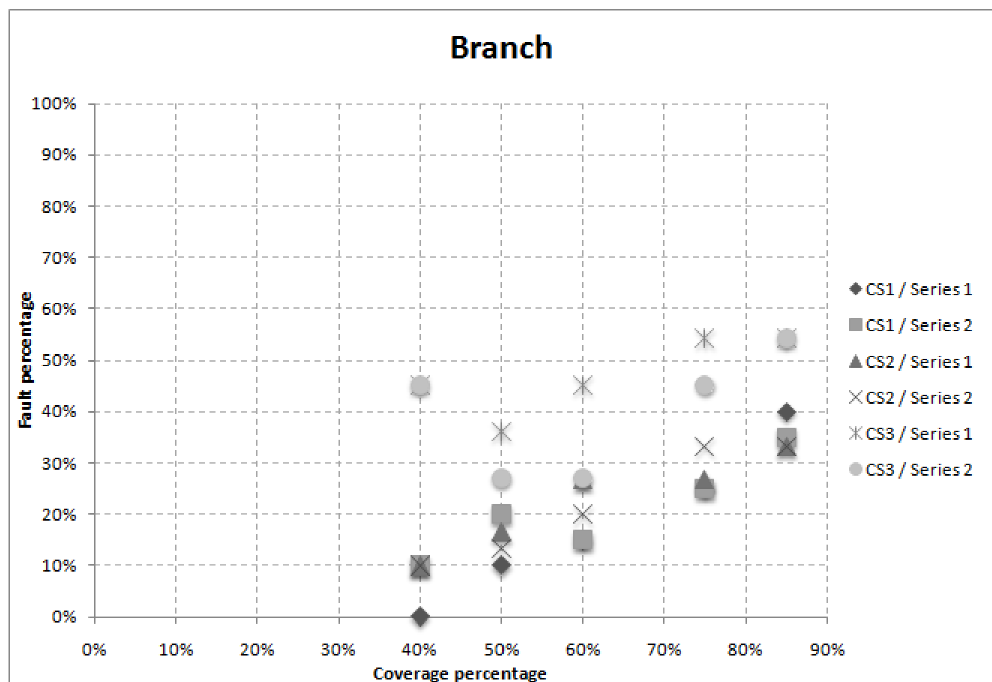


Figure 6.13: Overall result - Branch coverage

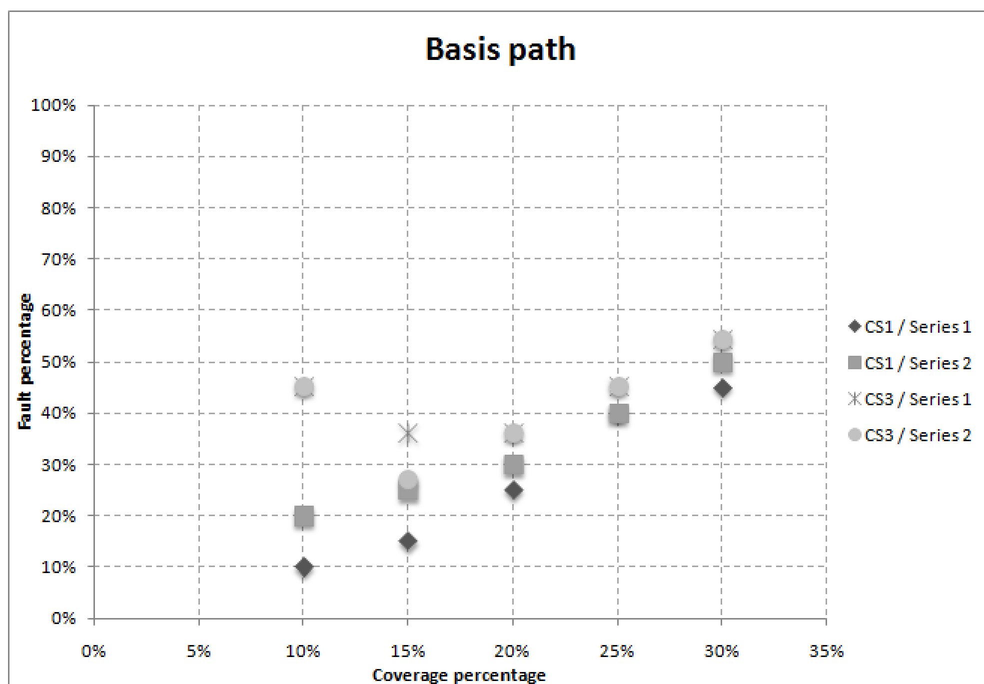


Figure 6.14: Overall result - Basis path coverage

CHAPTER 7

Conclusion and Future Work

This chapter provides an overall conclusion based on the results of the experiment. We provide answers to the research questions we formed at the beginning of the experiment, and propose directions for future work.

Organization of this chapter Section 7.1 provides answers to the research questions. Then, in Section 7.2 we propose directions for future work.

7.1 Answer to research questions

This section provides answers to the research questions we defined at the beginning of the experiment. First, we answer the subquestions in Section 7.1.1. Then, we answer our main research question in Section 7.1.2.

7.1.1 Subquestions

SQ1: What is the correlation between different coverage metrics and software quality? All three measured metrics showed a positive and almost similar correlation between the coverage percentage and fault percentage. As shown in Section 6.5.1, 6 out of 8 measurements show a correlation of +0.80 or higher.

The other 2 measurements, both from case study 3, showed a correlation of +0.58 (branch coverage) and +0.56 (basis path coverage). This is caused by deviations that occurred at the lower coverage percentages for these metrics. Test suites with 40 percent branch coverage found an exceptional high number of faults (45 percent of all faults). The same holds for test suites that achieved 10 percent basis path coverage (also found 45 percent of all faults). Further investigation on these deviations showed that these test suites were relatively lucky in their composition, as by coincidence they contained the test cases that exposed the faults. The explanation for the fact that this phenomenon occurred

with both metrics, is that the test suites used here achieved both 40 percent branch coverage and 10 percent basis path coverage at the same time. Therefore, the test suites were actually the same. To show the impact of these deviations on the correlation coefficient, we re-calculated them excluding these deviation. This resulted in a correlation coefficient of +0.85 for branch coverage, with a confidence interval of $0.37 < r < 0.98$. Recalculation of the coefficient for basis path coverage resulted in +0.96, with a confidence interval of $0.79 < r < 0.99$.

As shown by Table 6.1 in Section 6.5.1, chances of a positive correlation to exist is likely for all three measured metrics. Thus, when applying coverage metrics, increasing the coverage percentage is likely to result in a higher number of faults to be found.

SQ2: At what coverage percentages will the effect of increasing coverage diminish? Overall results provided in Section 6.5.1 show that the linear growth of the fault percentage is continuous, even when reaching higher coverage percentages. This holds for all three measured metrics. An exception on this trend occurs at the statement coverage metric in case study 2 (see Figure 6.5), where the fault percentage does not significantly increases when the coverage percentage exceeds 70 percent. However, as this is the only exception that occurred, we assume that in general the effect of increasing coverage does not diminish at all.

Therefore, independent of the already achieved coverage percentage or used metric, increasing the coverage percentage is likely to result in more faults to be found. Obviously, this only holds for the coverage percentages tested in this experiment.

SQ3: How does an increase or decrease of the coverage percentage influence the testing costs? In Section 6.5.1 we showed that the testing costs grow linearly with the coverage percentage. There was no increased steepening in costs when achieving higher coverage percentages. This linear increase in costs is likely caused by the low cyclomatic complexity of the involved case studies. As shown in Section 5.5, the cyclomatic complexity varied between 1.3 and 1.9. As a consequence, achieving higher coverage percentages did not require flipping an exceptional amount of decisions to reach uncovered code.

7.1.2 Main question

RQ: How should coverage testing be used by the software industry to improve software quality in a profitable way? The software industry itself is a way too broad target to answer our main question. As discussed in Section 5.4, test cases used in this experiment cannot serve as a random sample for the software industry. Therefore, the conclusions mentioned here should only be considered valid for the company Info Support and their specific test process, and in lesser extent to software companies that use the test-driven development methodology as their main method to create test cases.

The normal to strong correlation between the coverage percentages of several metrics and software quality suggests that increasing the coverage percentage will likely increase the number of faults that are found by a test suite. This holds for each tested metric and coverage percentage, i.e., independent of the

used metric and/or already achieved coverage percentage. However, the exact correlation remains uncertain, as we performed a limited number of measurements during the experiment.

We observed that the different metrics performed equally, i.e., find almost the same amount of faults. What should be taken into account on this result is that (1) tested coverage percentages differed among the metrics, and (2) the cyclomatic complexity was relatively low, resulting in a reduced difference among metrics. For instance: a method that has a cyclomatic complexity of 1, can be tested on a 100 percent statement coverage and a 100 percent branch coverage using the same test case.

The exact percentage of the total number of faults that are found by a certain metric and corresponding coverage percentage remains uncertain, as they differed significantly among case studies. Moreover, even in best-case scenarios, most metrics were able to find as little as half of the total faults(see Section 6.5). Therefore, coverage testing should definitely not be the only method in a test process. After investigating the faults that were (not) found, we conclude that faults that manipulated the behavior of the program under normal conditions were found mostly. Faults that only occurred in more exceptional situations (See Section 6.5 for details) were mostly left undetected. It is very likely that boundary-value analysis [26] and equivalence partitioning [26] would have exposed most of these undetected faults in our case studies.

The costs of increasing the coverage percentage grows linearly. There are no exceptional peaks or steepened increases whatsoever. Therefore, there are no obvious boundaries that should be taken into account from an economic point of view. However, as discussed in SQ2, the linear relation is likely to be maintained by the low cyclomatic complexity of the involved case studies. Therefore, we recommend keeping the cyclomatic complexity of any project as low as possible, as this likely keeps the test effort to reach higher coverage percentages under control.

Based on the motives mentioned above, we recommend using statement and branch coverage. We do not recommend using basis path coverage, as the current tool support is very minimal.

We do not recommend any specific coverage percentage to achieve. This decision should be made by the managers based on their specific demands. However, the results of this experiment can be used as a guideline to find a balance between software quality and spending resources on testing. Besides using coverage testing, we recommend adding boundary-value analysis and equivalence partitioning to the test process.

7.2 Future work

Based on our results there are several interesting directions for future work. First, we tested very low coverage percentages for the basis path coverage metric. However, despite the low coverage percentages, test suites that did achieve these percentages covered a relatively high amount of faults. Future research may investigate the increase in fault percentage for higher coverage percentages.

Second, test cases used in this research were created based on the test-driven development methodology. Obviously, this has a direct consequence on the way the system is tested, and therefore the number of faults that are found. Further

research could specifically focus on the influence that the type of test cases have on the efficiency of coverage metrics.

Third, as we mentioned in our results, most faults that were not found by the coverage metrics, could have been exposed using boundary-value analysis and/or equivalence partitioning. Therefore we suggest investigating their efficiency using industrial case studies.

Finally, similar studies are needed to obtain an increased confidence on the correlation between coverage and software quality, and to validate our result. A recommendation would be to increase the number of injected faults compared to this experiment. This will improve the exactness of the correlation coefficients. But also to add case studies that have a relative high cyclomatic complexity, to investigate the effect this has on the efficiency of the metrics and the costs.

References

[1] Codign website. `http://www.codign.com/`.

[2] Eclemma website. `http://www.eclemma.org/`.

[3] Eclipse documentation. `http://www.eclipse.org/documentation/`.

[4] Junit website. `http://www.junit.org`.

[5] Ncover website. `http://www.ncover.com/`.

[6] Nunit website. `http://www.nunit.org`.

[7] C. Agruss and B. Johnson. Ad hoc software testing: A perspective on exploration and improvisation. pages 68–69, 2000.

[8] F. E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, 1970.

[9] S. Amland. Risk-based testing:: Risk analysis fundamentals and metrics for software testing including a financial application case study. *Journal of Systems and Software*, 53(3):287 – 295, 2000.

[10] Kent Beck. *Test Driven Development. By Example (Addison-Wesley Signature)*. Addison-Wesley Longman, Amsterdam, 2002.

[11] T. A. Budd. *Mutation analysis of program test data*. PhD thesis, New Haven, CT, USA, 1980.

[12] X. Cai and M. R. Lyu. The effect of code coverage on fault detection under different testing profiles. In *Advances in Model Based Testing*, 2005.

[13] Jacob Cohen. *Statistical Power Analysis for the Behavioral Sciences (2nd Edition)*. Routledge Academic, 2 edition, January 1988.

[14] E.W. Dijkstra. Structured programming. In *In Software Engineering Techniques. NATO Science Committee*, August 1970.

[15] D.E. Eckhardt, A.K. Caglayan, J.C. Knight, L.D. Lee, D.F. McAllister, M.A. Vouk, and J.P.J. Kelly. An experimental evaluation of software redundancy as a strategy for improving reliability. *Software Engineering, IEEE Transactions on*, 17(7):692 –702, jul. 1991.

[16] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., Boston, MA, USA, 1998.

[17] R. A. Fisher. On the probable error of a coefficient of correlation deduced from a small sample. *Metron*, 1:3–32, 1921.

[18] P. G. Frankl and O. Iakounenko. Further empirical studies of test effectiveness. In *SIGSOFT FSE*, pages 153–162, 1998.

[19] P. Garg. Investigating coverage-reliability relationship and sensitivity of reliability to errors in the operational profile. In *Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research (CASCON '94)*, page 19. IBM Press, 1994.

[20] D. Garvin. What does product quality really mean? *Sloan Management Review*, 26:25–45, 1984.

[21] J. R. Horgan and S. London. Data flow coverage and the c language. In *Proceedings of the Symposium on Testing, Analysis, and Verification (TAV4)*, pages 87–97, New York, NY, USA, 1991. ACM.

[22] ISO/IEC. *ISO/IEC 9126. Software engineering – Product quality*. ISO/IEC, 2001.

[23] M. R. Lyu, Z. Huang, S. K. S. Sze, and X. Cai. An empirical study on testing and fault tolerance for software reliability engineering. In *Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE '03)*, page 119, Washington, DC, USA, 2003. IEEE Computer Society.

[24] S. McConnell. *Code Complete, Second Edition*. Microsoft Press, Redmond, WA, USA, 2004.

[25] J. D. Musa. Operational profiles in software-reliability engineering. *IEEE Softw.*, 10(2):14–32, 1993.

[26] G. J. Myers and C. Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.

[27] National Institute of Standards and Technology. Economic impacts of inadequate infrastructure for software testing. `http://www.nist.gov/director/planning/upload/report02-3.pdf`, May 2002.

[28] Toyota Motor Sales TMS. Toyota announces voluntary recall on 2010 model-year Prius and 2010 Lexus HS 250h vehicles to update ABS software. `http://pressroom.toyota.com/pr/tms/toyota-2010-prius-abs-recall-153614.aspx`, February 2010.

[29] A Watson and T NIST McCabe. Special publication 500-235, structured testing: A software testing methodology using the cyclomatic complexity metric. In *U.S. Department of Commerce/National Institute of Standards and Technology*, 1996.

[30] A. H. Watson, T. J. Mccabe, and D. R. Wallace. Special publication 500-235, structured testing: A software testing methodology using the cyclomatic complexity metric. In *U.S. Department of Commerce/National Institute of Standards and Technology*, 1996.

[31] J.M. Wing. A specifier's introduction to formal methods. *IEEE Computer*, 23(9):8, 10 –22, 24, sep. 1990.

[32] W. E. Wong, J. R. Horgan, S. London, A. P. Mathur, Md Joseph, and R. Horgan. Effect of test set size and block coverage on the fault detection effectiveness. In *Proceedings of the Fifth IEEE International Symposium on Software Reliability Engineering*, 1994.

APPENDIX A

Test suite design costs

A.1 Case study 1

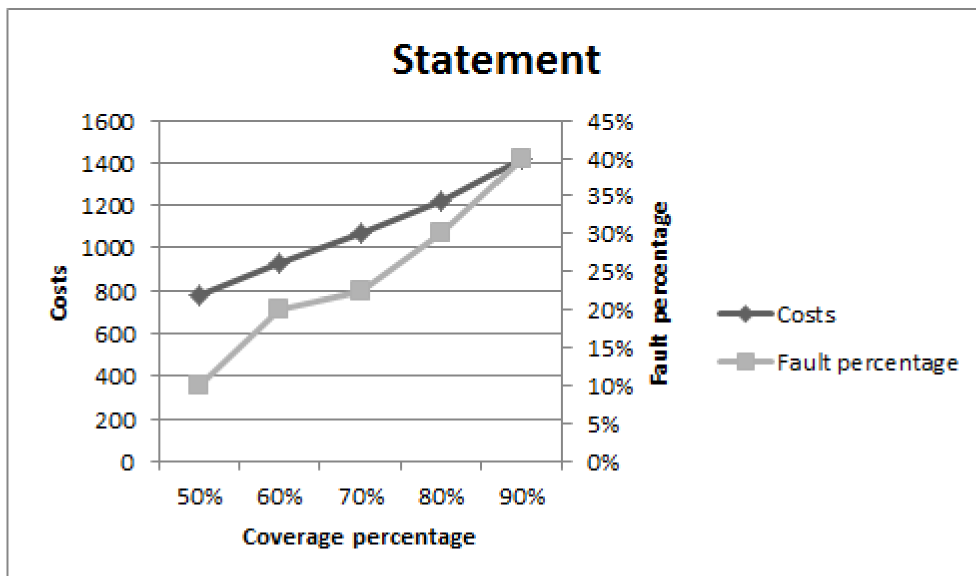


Figure A.1: Case study 1 - Statement coverage

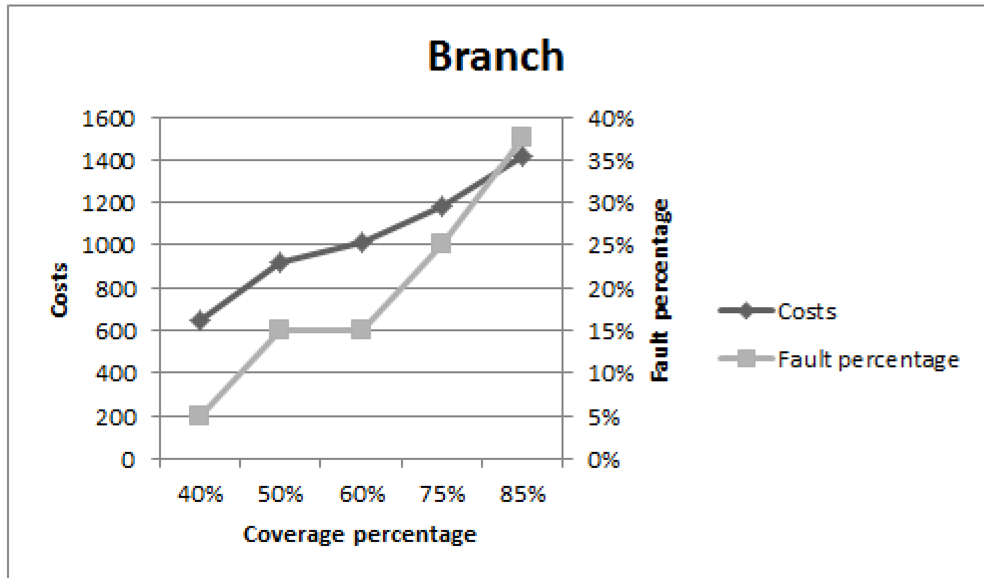


Figure A.2: Case study 1 - Branch coverage

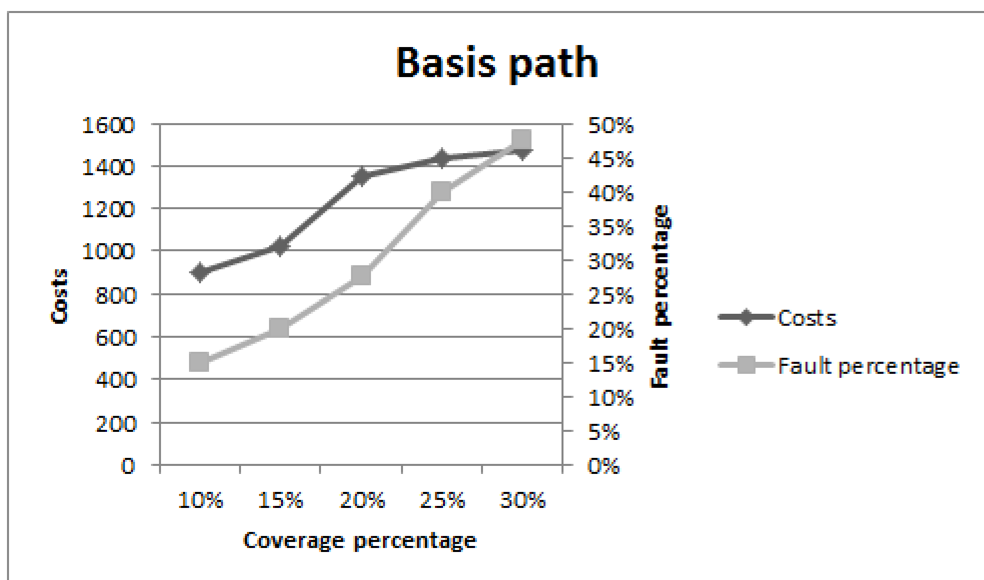


Figure A.3: Case study 1 - Basis path coverage

A.2 Case study 2

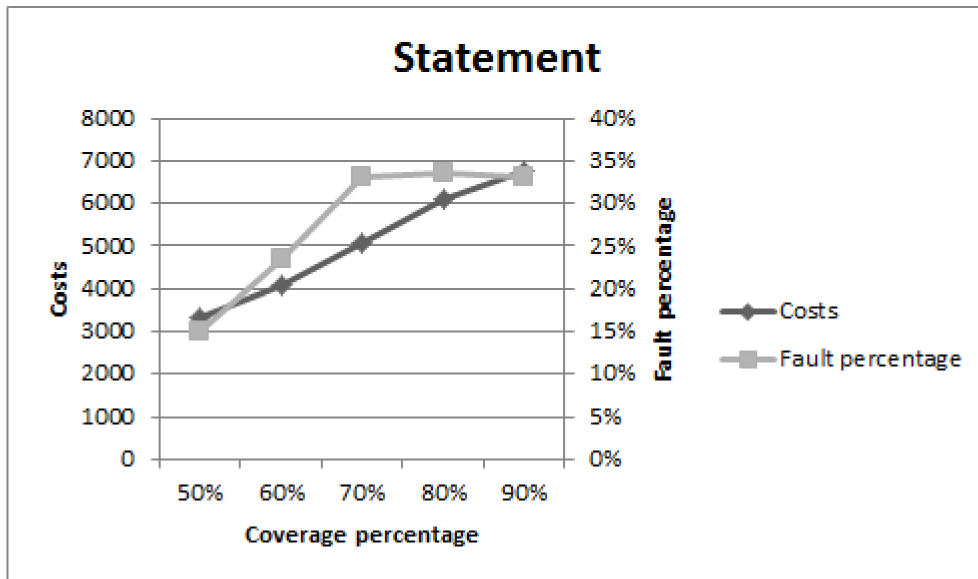


Figure A.4: Case study 2 - Statement coverage

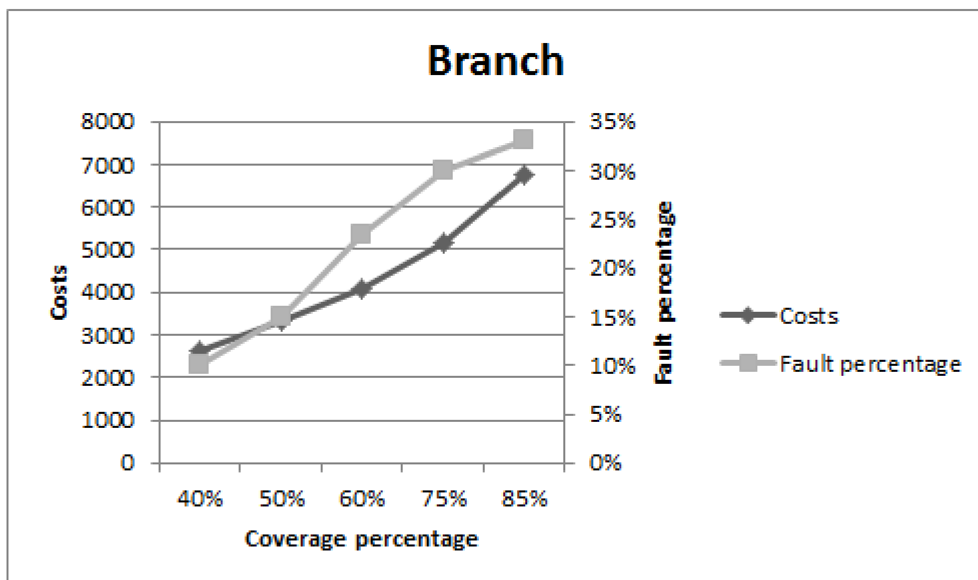


Figure A.5: Case study 2 - Branch coverage

A.3 Case study 3

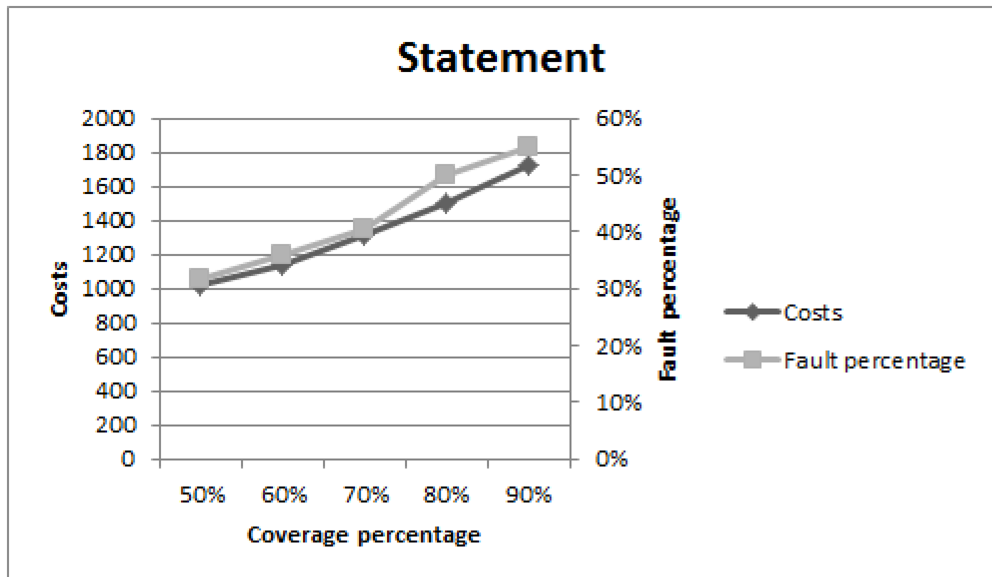


Figure A.6: Case study 3 - Statement coverage

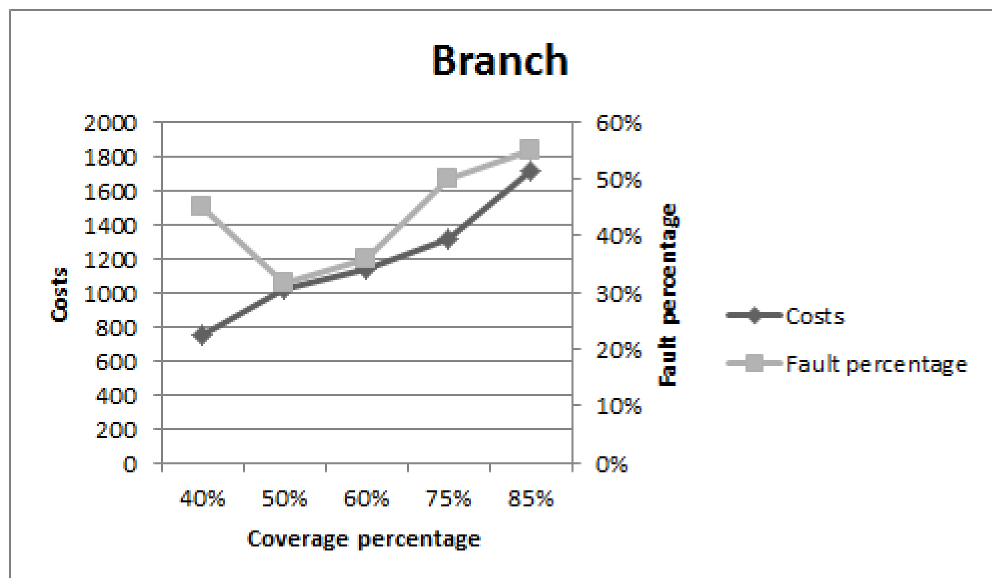


Figure A.7: Case study 3 - Branch coverage

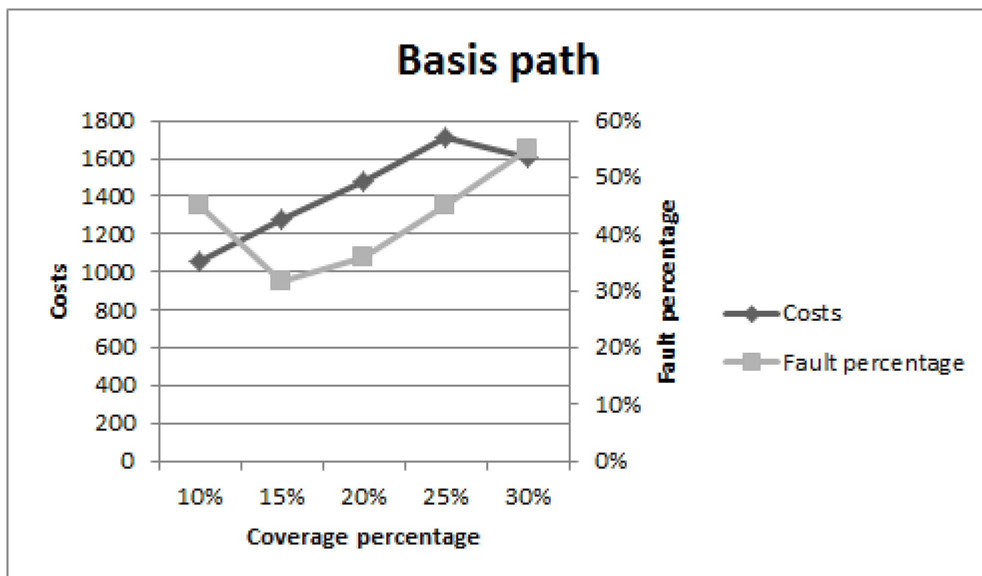


Figure A.8: Case study 3 - Basis path coverage