Examensarbete

HÖGSKOLAN
I SKÖVDE

EVALUATING THE EFFECTIVENESS OF TEST COVERAGE CRITERIA USING MUTATION ANALYSIS

An evaluation of test coverage criteria in C#

Examensarbete inom huvudområdet
Informationsteknologi
Grundnivå 30 Högskolepoäng
Vårtermin År 2016

Erik Johansson

Handledare: Joe Steinhauer
Examinator: Jonas Mellin

Abstract

Test coverage criteria introduces metrics to measure the adequacy of a test suite as well as defined rules for performing software testing. This makes it possibly to formally define requirements for testing in various industries where software quality is essential, such as in aviation. This bachelor thesis aims to study the effectiveness of 11 different coverage criteria in two ways. Firstly how effective they are at finding faults and secondly their cost effectiveness. Test cases were created for each individual criteria for three different programs. The effectiveness of these test cases was then measured using mutation analysis. The results revealed that the scale of the experiment was too small to truly draw any conclusions regarding effectiveness. It was however shown that due to the "test noise" effect, the effectiveness of the test criteria can differ notably. It was also shown that test coverage criteria alone may not be sufficient in order to perform efficient testing.

Table of Contents

1 Introduction

Software testing is an essential part of the software development process and is carried out in order to increase the quality and reliability of the software. A problem is that a program may have millions of input combinations and may execute in a countless number of different ways. This means that testing is always about compromise since it is practically impossible to fully test a program of larger size and complexity. Test coverage criteria introduces rules and requirements for how the testing should be conducted. An example of a criteria is statement coverage, where each statement in the software is required to be executed at least once. In order for the tester to make a well-informed decision when choosing criteria, knowing how effective various coverage criteria are could be highly useful. This bachelor thesis aims to evaluate the major coverage criteria in use using mutation analysis. Mutation analysis is a technique for automatically seeding faults into a program. Test quality can then be measured by the ability to find these faults, resulting in a mutation score. Two different properties of each criteria are measured, its ability to find faults and its cost-effectiveness. The field of mutation analysis can be traced back to the 1970's, but has only recently begun to reach a level of maturity where it might be better used in the industry (Jia & Harman, 2012).

In chapter 2 the reader is presented with a background knowledge in coverage criteria and mutation analysis. Chapter 3 then follows and introduces the problem statement, its aims and objectives. The method strategy and its validity threats is then presented in chapter 4 and the method implementation in chapter 5. Following this, the results of the experiment are presented in chapter 6, a discussion of the results in chapter 7 and finally a summary of the bachelor thesis, conclusions and future work in chapter 8.

2 Background

This section aims to present the required fundamental knowledge in coverage criteria and mutation analysis. A short introduction of coverage criteria is first presented in section 3.1. This is then followed by a presentation of graph coverage criterions in section 3.2 and logic coverage criterions in section 3.3. Finally, section 3.4 presents a summarization of mutation analysis.

2.1 Test coverage criteria

This section aims to define what coverage criteria is and present the criterions that are evaluated in the experiment. Unless otherwise stated, everything in this and the following two sections (3.1 – 3.3) all refer to information presented in *Introduction to Software Testing* (Ammann & Offutt, 2008).

Software testing is always about compromise. Ideally, each program would be fully tested to ensure reliability. Meaning that each possible execution path and every possible input is tested. This is however practically impossible for most programs, as the number of test cases would be of such a magnitude that it would be practically impossible to implement. Test coverage criteria introduces a way to formalize stopping criteria and rules for designing purposeful tests. An example of the most simple criteria is statement coverage, where each individual statement creates a separate requirement and each requirement must be met in order to meet the criteria. Several criteria have a hierarchical structure, where a more strict criteria may subsume a lesser one. A criteria subsuming another means that each test requirement of the subsumed criteria is also met by the criteria subsuming it. A wide amount of coverage criteria exists, this article focuses on the criteria presented and formalized by Ammann & Offutt (2008) which is presented in detail in the following two sections.

2.2 Graph coverage

The coverage criteria concerning the structure (such as statement coverage) and data flow of a program has requirements that apply to a directed graph. By abstracting the software artifact to a graph model it's possible to define criteria that are universally applicable regardless of the programming language used. The graph model has the following requirements:

- It has at least one initial starting node
- It has at least one final node

The graph type that is usually used for modelling source code is called a *control flow graph* (CFG). In a control flow graph, each edge represents a branch in the source code. Each node represents a basic block of statements. A basic block is a set of statements where if one statement is executed, every other statement in the block is also executed. It has only one point of entry and one point of exit. In Figure 1 below, a control flow graph modeled after a program with two if-else statements is presented.

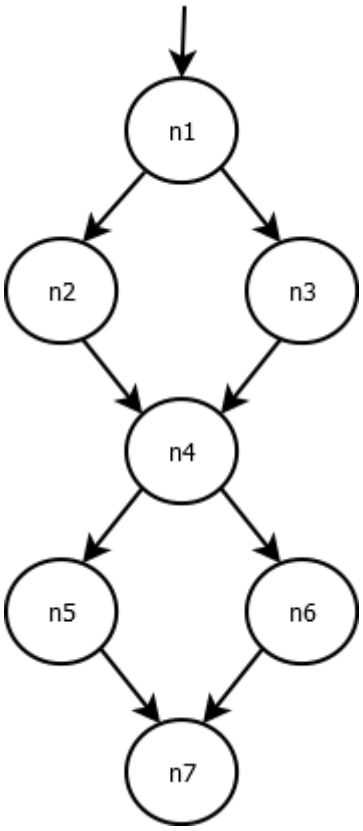

Figure 1 – A Control Flow Graph

In the presented graph node *n1* is the initial node and *n7* the final one. Control flow graphs may also have multiple points of entry as well as multiple points of exit. Knowing how and why graphs are used, the following general definition can be presented for graph coverage:

"Given a set TR of test requirements for a graph criterion C, a test set T satisfies C on graph G if and only if for every test requirement tr in TR, there is at least one test path p in path(T) such that p meets tr" (Amman & Offutt, 2008, pp 33)

Each different criteria will generate a set of test requirements *TR*. Each requirement *tr* being represented by either paths that needs to be traversed or nodes that has to be visited by at least one test path *p*. To represent test paths, the following notation is used: *path1* = [n1, n2, n4, n5, n7]. Each element represents a node, and each node is visited in the order from left to right. The notation for showing test requirements has the following structure: TR = {n1, n2, n3}. Each element within the curly brackets represent a single test requirement. Each requirement may either be a node or a test path. In the next sections each coverage criteria covered by the study is shortly summarized. The following two sub-sections present a summary of the coverage criteria as presented by Ammann and Offutt (2008). For each criteria, a definition is first presented followed by a short explanation as well as an illustrative example when required.

2.2.1 Structural Graph Coverage

Node Coverage (NC)
Node coverage requires that each node in a graph is reached. This means that each node in the graph creates a single test requirement. The graph presented in figure 1 above would get the following requirements: TR = [n1, n2, n3, n4, n5, n6, n7]. To meet those requirements each node needs to be visited by at least one test path. The previously identified test requirements can be met with the following two paths:
path1 = [n1, n3, n4, n6, n7]
path2 = [n1, n2, n4, n5, n7]

These two paths will meet each test requirement as they collectively visit each node.

Edge Coverage (EC)
Edge coverage requires that each edge in the graph is covered. In order for edge coverage to subsume node coverage even in the case of nodes without connecting edges. It is explicitly required that each node is also covered. This would create the following test requirement for the graph in figure 1:
TR = { [n1,n2], [n1,n3], [n2,n4], [n4,n5], [n4,n6], [n5,n7], [n6,n7] }

Prime Path Coverage (PPC)
Prime Path uses the notion of prime paths and simple paths. A path is simple if each node in the path is only visited once, with the exception of the first and the last node which may be identical. A prime path has two rules, it has to be a simple path and it may not appear as a subpath of another simple path. This makes it possible to cover loops without getting an infinite number of test requirements. To illustrate, a prime path in the node in figure 1 is [n1, n3, n4, n6, n7]. Prime path coverage requires that each prime path of a graph is traversed.

2.2.2 Data Flow Graph Coverage

Data flow coverage criteria concerns the flow of the data in the software, such as making sure that each definition of a variable should reach a statement where it is used. In order to apply data flow coverage criteria, the notions of *definitions* (def) and *uses* (use) are used. A definition occurs when a value for a variable is stored into memory. A use occurs when a variable is accessed. In order to apply coverage criteria, defs and uses also needs to be presented in the CFG. A CFG with uses and defs of variables *a* and *b* is presented in figure 2 below.

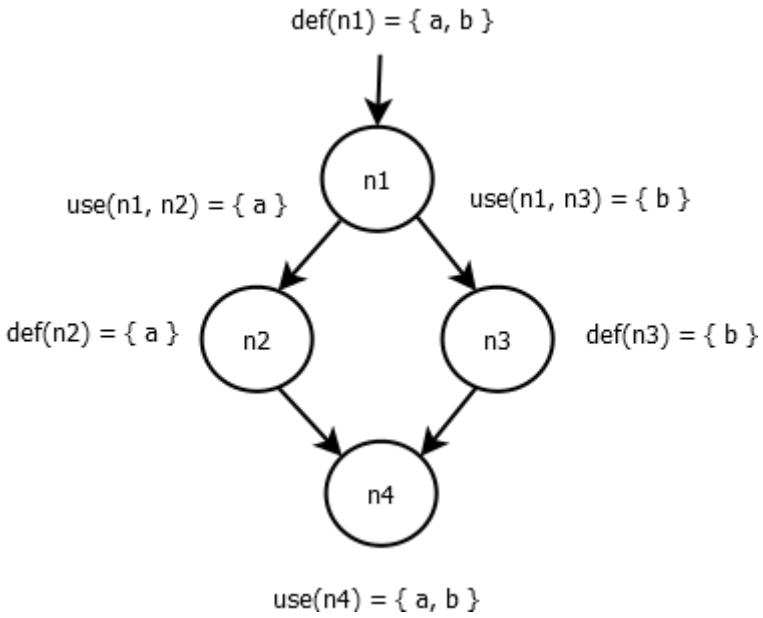

Figure 2 - CFG with uses and defs

Data flow coverage is mainly concerned with *du-pairs*. A du-pair is the association between a def and a use of a variable. Each def may not reach every one of its uses. This may either be due to reachability (no connecting edge) or because it reaches another def before it reaches the use. A path is said to be *def-clear* for a variable if there is no new def along the path. A du-path is a def-clear simple path between the def and use for a given variable.

All-Defs Coverage

All-Defs Coverage requires that each definition reaches at least one use it has a def-clear path to. Using the graph presented in Figure 2, the following test requirements can be identified:

TR a = { [n1, n2], [n2, n4] }
TR b = { [n1, n3], [n3, n4] }

All-Uses Coverage

All-Uses Coverage requires that each definition reaches *all* of its uses it has a def-clear path to. Due to this, All-Uses Coverage subsumes All-Defs Coverage.

All-du-Paths Coverage

All-du-Paths Coverage requires that each definition reaches *every* def-clear path to *all* of its uses. All-du-Paths subsumes All-Uses Coverage.

2.2.3 Subsumption relationships for Graph Coverage Criteria

As mentioned in the previous sections, there are several subsumption relationships among graph coverage criteria. These are more clearly illustrated in Figure 3 below.

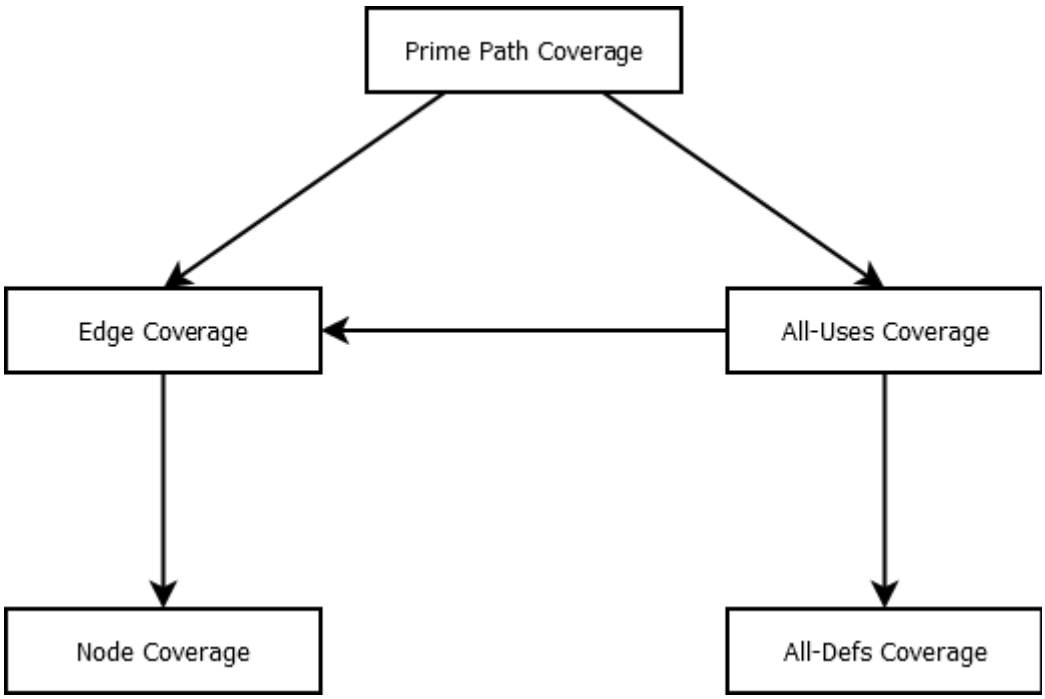

Figure 3 - Subsumption relationships for Graph Coverage Criteria

2.3 Logic coverage

Logic coverage criteria are based on logical expressions: predicates and clauses. A predicate is an expression that evaluates to a boolean value, such as (a Λ b). Clauses are predicates that does not contain any of the logical operators as presented in the list below. A predicate may be made up of several clauses, ((a > b) V (a < c)) contains two clauses, (a > b) and (a < c). In order to use the coverage criteria, the predicates and clauses in the source code first needs to be translated to generalized expressions using the logical operators listed below:

- ¬ - the negation operator
- ^ - the and operator
- v - the or operator
- → - the implication operator
- ⊕ - the exclusive or operator
- ↔ - the equivalence operator

The most obvious part of the source code that consists of predicates and clauses are if-statements. The following if-statement:

```
if (( a < b) && ((x > y)
|| C)
          doThis();
else
          doThat();
```

gets translated to ((a < b) Λ ((x > y) V C).

Predicate Coverage

Predicate Coverage requires that each predicate evaluates to both true and false. For the predicate ((a > b) Λ C) the input (a = 6, b = 4, C = true) and (a = 4, b = 6, C = false) can be used to satisfy predicate coverage.

Clause Coverage

Clause coverage is similar to predicate coverage, the difference being that the requirement is for each individual clause. Using the same predicate ((a > b) ^ C), the clauses (a > b) and C individually needs to be evaluated to both true and false. The test requirements can be met with the same input listed for predicate coverage. It should be noted that clause coverage does not subsume predicate coverage, each clause can individually be evaluated to both true and false without having the whole predicate be evaluated to both.

Active Clause Coverage

Active Clause Coverage is similar to Clause Coverage with the additional requirement that each clause should independently evaluate the whole predicate to either true or false. The clause determining the predicate is referred to as the *major* clause and the other clauses as *minor*. Active Clause Coverage is divided into two coverage criteria: CACC and RACC. The difference between the two is the requirements placed on the minor clauses.

a	b	c	a ^ (b v c)
T	T	T	T
T	T	F	T
T	F	T	T
F	T	T	F
F	T	F	F
F	F	T	F

Table 1 - Truth table for (a Λ (b v c))

Correlated Active Clause Coverage (CACC)

CACC places two requirements on each major clause. It should evaluate to both true and false when the values of the minor clauses are chosen in a way so that the major clause determines the whole predicate. This is essentially Active Clause Coverage as described above. There are no requirements on whether the minor clauses stay the same as long as the major clause determines the predicate.

A valid test requirement for a in the predicate a ^ (b v c) can be identified by observing the truth table in Table 1 above: TR a = { (a=true, b=true, c= true), (a=false, c=true, c=false) }. As illustrated, the minor clauses can differ while still meeting the criteria.

Restricted Active Clause Coverage (RACC)

The difference between RACC and CACC is the requirements placed on the minor clauses. RACC requires that the minor clauses stay the same when the predicate is evaluated to true and false. This means that the identified test requirements presented under CACC are not applicable to RACC, due to the minor clauses being different. A valid test requirement for clause a in the predicate presented in Table 1 above can then be identified as TR a = { (a = true, b = false, c = true), (a = false, b=false, c=true) }. RACC subsumes CACC.

Inactive Clause Coverage

Inactive Clause C*ove*rage is essentially the opposite of Active Clause Coverage. The values of the minor clauses should instead be chosen in such a way such that the major clause can't determine the predicate. As with Active Clause Coverage there are two variants, RICC and GICC. The difference between them being the requirements placed on the minor clauses. Every Inactive Clause Coverage subsumes Predicate Coverage.

General Inactive Clause Coverage (GICC)
GICC places four requirements on each major clause. It should evaluate to true when the predicate is true, it should evaluate to false when the predicate is true, it should evaluate to true when the predicate is false and finally it should evaluate to false when the predicate is false. GICC places no requirement for varying the values of the minor clauses among the four cases.

Restricted Inactive Clause Coverage (RICC)
RICC places the same requirements as GICC on each major clause. The only difference being that the minor clauses must not differ when the predicate is evaluated to true or when the predicate is evaluated to false.

2.3.1 Subsumption relationships for Logic Coverage Criteria

As mentioned in the previous sections, there are several subsumption relationships among logic coverage criteria. These are more clearly illustrated in Figure 4 below.

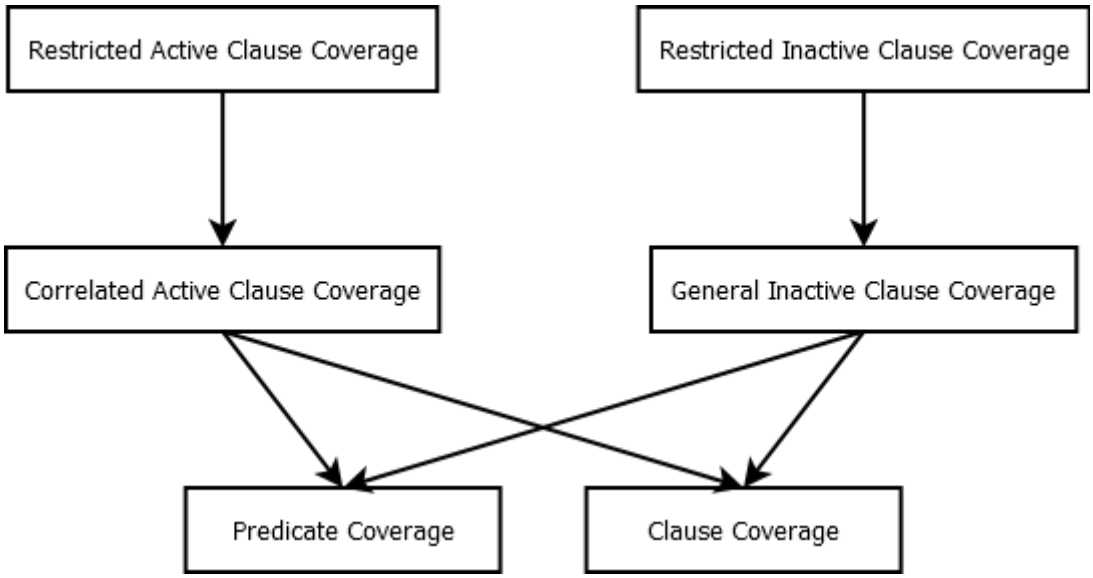

Figure 4 - Subsumption relationships for Logic Coverage Criteria

2.4 Mutation analysis

Mutation analysis is a fault-based testing technique that provides a method for seeding faults into a program. Each introduced fault in the program creates a mutant, a mutated version of the original program which is intended to represent a real error. A mutant is said to be eliminated, or "killed", when running the mutated program causes different behavior compared when to the original program. A mutant can be eliminated either "strongly" or "weakly". Mutants are strongly killed when executing the mutant causes observable different behavior, such as different output. Weakly killing a mutant simply requires that the mutation causes an invalid state. The latter is much harder to detect, mutation analysis is therefore mainly concerned with the strongly killed mutants. The result of the analysis is evaluated as the percentage of eliminated non-equivalent mutants. For example, if 1000 mutants were generated for program p and 900 of those were eliminated with test suite t then the mutation score would be 90% for t. (Offutt & Ammann, 2008, Jia & Harman, 2011).

Mutation analysis can be used as a test coverage criteria, where a desired mutation score sets the requirement. It has also been extensively used as a method to generate faults where the number of known faults are low or absent (Just, Jalali, Inozemtseva, Ernst, Holmes & Fraser, 2014).

2.4.1 Equivalent mutants

One of the biggest challenges of mutation analysis is identifying equivalent mutants. A mutant is equivalent if it always causes the same behavior as the original program, it is syntactically different but functionally equivalent. (Ammann & Offutt, 2008, Jia & Harman, 2011). An example illustrating this is presented in Figure 5 below. In the figure it can be observed that the mutant produces functionally equivalent behavior to the original program, the for-loop will always terminate at the same time. According to Offutt (2007), automatically detecting every equivalent mutant is impossible. This means that detecting every equivalent mutant is not possible without manual analysis. This produces a significant overhead in the mutant analysis process and has been one of the barriers for making mutation analysis a more viable testing technique. There are however partial solutions that can detect a subset of all the equivalent mutants, Baldwin and Sayward (1979) presents such a technique that utilizes compiler optimization rules. Not identifying every equivalent mutant creates an invalid mutation score, since non-equivalent mutants are by definition impossible to eliminate.

```
Original statement:
for(int i = 0; i < 10; i++)
Mutant:
for(int i = 0; i != 10; i++)
```

Figure 5 - Equivalent mutant

2.4.2 Mutation Operators

Mutation analysis seeds faults into a program with the use of a well-defined set of mutation operators. As programming languages differ, mutant operators will also do depending on the language and testing. Relational Operator Replacement, or ROR, is an example of a mutant operator which has the following definition: "*Each occurrence of one of the relational operators (<, ≤, >, ≥, =, ≠) is replaced by each of the other operators and by falseOp and trueOp"(Ammann & Offutt, 2008, pp 183).* The full set of mutants generated with the ROR operator for the clause a <= 0 is presented in Table below.

Operator replacement	Mutant
<	a < 0
>	a > 0
≥	a ≥ 0
=	a = 0
≠	Side1 ≠ 0
falseOp	false
trueOp	true

Table 1 - The ROR operator applied to the clause a <= 0

As the table shows, a total of seven mutants are created with just one mutant operator on one applicable expression. This means that for each other expression that is valid for the operator, seven more will be generated. Each other mutant operator will also in turn generate a set of mutants for each fitting expression. This shows how the amount of mutants can grow to very large numbers even for smaller programs. For large programs, the amount of mutants may number in millions (Gopinath, Alipour, Ahmed, Jensen & Groce, 2015). A field of study in the research of mutation analysis is how this set can be reduced while still producing a useful result. For example, it has been found that by using a sample size of 1000 mutants it is possible to approximate the mutation score with 95% certainty (Gopinath et.al., 2015).

2.4.3 The validity of mutation analysis

The theoretical validity of mutation testing is based on two fundamental hypotheses. The first is the Competent Programmer Hypotheses, it states that programmers are competent and create programs that are close to being correct. Meaning that the created faults are simple in nature and can be corrected with merely simple syntactical changes (DeMillo, Lipton & Sayward, 1978). The other hypotheses is the coupling effect that states *"Test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors"* (DeMillo, Lipton & Sayward, 1978, pp 35). This essentially means that by uncovering simple errors, more complex and deeper errors can be found due to being coupled to the simple ones.

Several studies has focused on researching the coupling effect and results show that it is supported by both theoretical and empirical results (Offutt, 1989, Offutt, 1992, Just et.al., 2014). This is further supported in a study that shows a clear correlation between the ability to detect mutants and real faults (Andrews, Briand & Labiche, 2005).

3 Problem Statement

This section presents the problem that is investigated in this paper. First a general description of the problem and the motivation behind it is presented. This is then followed by the aim of the paper and the identified objectives.

3.1 Problem Description & Motivation

Testing is a vital part of the software development process and is performed in order to increase the reliability and quality of software. However, it is practically impossible to fully test a software and ensure reliability with 100% certainty. Test coverage criteria consists of rules and requirements that needs to be met in order to fulfill the chosen criteria. These rules have two functions, they provide stopping criteria for defining when testing can be declared sufficient and they give the tests purpose. An example of a criteria is statement coverage, this simply requires that each statement is executed at least once.

Different criteria have varying levels of strictness, choosing a more strict criteria will yield a higher probability of finding faults but require a higher cost to develop and maintain the test suite. In order for the tester to make well informed decisions, knowing the effectiveness of various coverage criteria could prove highly useful. This paper aims to investigate the effectiveness of coverage criteria using mutation analysis and the C# programming language. The chosen criteria are versions of the major coverage criteria in use today, as formalized by Ammann & Offutt (2008). The criteria are presented below:

- Node Coverage
- Edge Coverage
- Prime Path Coverage
- All-Defs Coverage
- All-Uses Coverage
- All-DU-Paths Coverage
- Correlated Active Clause Coverage
- Restricted Active Clause Coverage
- General Inactive Clause Coverage
- Restricted Inactive Clause Coverage

The technique used for evaluating coverage criteria is mutation analysis. Mutation analysis has been shown to be an effective way to predict the ability of a test suite to find real faults (Just et al., 2014, Ramler, Kaspar, 2012, Andrews, Briand, Labiche, 2005, Offutt, 1992).

3.2 Aims & Objectives

The aim of the investigation is to measure the effectiveness of the chosen criteria in C# with respect to its ability to find faults and its cost. In order to meet this aim, the following objectives has been identified:

1. **Evaluate how effective the coverage criteria are at finding faults**
   The used metric for measuring effectiveness is mutation score. This seems appropriate because mutation score represents the ability of the test cases to find faults.

2. **Evaluate the cost effectiveness of the criteria**
   In order to measure cost-effectiveness, the relationship between test suite size and its mutation score is used. Formalized as follows:

   Mutation Score(MS) / Test suite size (number of test cases)

   This is a not a perfect measure since there are several other factors that may should be taken into consideration when measuring the cost of meeting a criteria. Examples being creating graphs and manually reviewing test criteria. The author of this bachelor thesis could however not find a good way to generalize these factors into a formula. The chosen formula should still show a good indication regarding cost effectiveness.

3.3 Delimitations & Expected outcome

The purpose of this study is not to be able to say that certain criteria are more or less effective on an absolute scale. Instead, the intention is to compare the criteria to one another in an effort to gain insight to their effectiveness. The expected outcome of the study are the following artifacts:

A set of data for each criteria showing effectiveness measured in mutation score, the amount of test cases required as well as the measured cost effectiveness. The subsuming criteria are expected to perform better than the criteria they subsume, for example it is expected that Prime Path Coverage performs better than Node Coverage.

4 Method & Approach

This section first presents the method and approach for achieving the objectives listed in section 3.2. What is evaluated is essentially the cause-effect relationship between using a certain criteria and its ability to find faults as well as its cost effectiveness. The chosen method to fulfill this is to conduct an experiment. The experiment will consist of creating test suites for each test criteria for three small programs and performing mutation analysis on each test suite.

4.1 Method strategy

The strategy chosen in order to meet the objectives is to conduct an experiment. Experiments are well suited for examining cause-effect relationships, such as the relation between criteria and test effectiveness. (Wohlin et al., 2012). Variables can be changed and the resulting effects can then be observed. The experiment consists of creating test suites for each criteria for 3 different programs. It would be desirable to conduct the experiment on a bigger pool of programs, this is however not possible due to time constraints and lack of automation. Studies investigating similar problems, as presented in section **Error! Reference source not found.**, also conducted similar experiments from which this experiment draws inspiration.

4.1.1 Alternative strategy

An alternative way to meet the objective of evaluating the effectiveness of coverage criteria is to perform a literature study. Gathering existing data from papers examining coverage criteria and evaluating it. There are however several problems with this approach. The author was not able to identify any papers examining each of the desired coverage criteria. The method for measuring effectiveness differs in many articles, making them harder to compare. Several overlaps between existing papers and this one could however be identified. Where applicable, the results of this paper will be compared to other similar studies.

4.2 Validity threats

In order for the experiment to provide meaningful results the fundamental question of validity needs to be evaluated. Wohlin et al. (2012) presents four different categories of validity threats: construct, internal, external and conclusion validity. A complete list of these threats is presented in appendix D. The most relevant threats and how they are handled will be presented in the sections below, a complete analysis of every threat presented by Wohlin et al. (2012) is presented in appendix D.

Conclusion validity

Conclusion validity is concerned with the correctness of the results, for example statistical validity. A relevant threat is that of low statistical power. To increase the statistical significance, three programs are used as a basis for testing. As previously mentioned it should be more but it is not possible due to time constraints. This means that found patterns may not have as strong statistical validity as a more extensive study. The study conducted in this bachelor thesis should still present an indication regarding effectiveness and lay the groundwork for further studies. Another threat is reliability of measures, the experiment mainly uses mutation analysis to measure effectiveness. Mutation analysis has been shown in earlier studies to be an effective and reliable way of measuring test quality as presented in section 2.4.3. Further, the result is compared to earlier known measures of effectiveness where it's possible. Fishing is also a potential threat, meaning that the experiment

could be manipulated to skew the result towards certain criteria. This is mostly handled by offering complete transparency in the experiment implementation, as well as using sound theories and accepted methods. By hiding no details of the experiment, any person with domain experience should be able to verify whether or not the result is impartial.

Instrumentation is a relevant threat, if the tool used for experiment execution, Visual Mutator, is badly designed or otherwise flawed, the result of the experiment may be negatively affected. Mutation analysis is still a relatively immature area that has not been extensively implemented in the industry, even more so for C#. This means that the tools are limited and has not been extensively tested through years of use in the industry. Visual Mutator was chosen due to being the most actively developed among the available tools.

Internal validity

Internal validity concerns threats that in some way influence the experiment's variables without the researcher's knowledge or intention. In the context of this report, if there are any threats that may affect the result other than the choice of coverage criteria. Instrumentation is a relevant threat, if the testing tool is badly designed it may generate incorrect results. The selected programs that the tests are designed for are also chosen with respect to applicability for mutation testing. This means that there needs to be code that each mutation operator can be applied to, otherwise the full set of mutation operators will not be used and the result may be misrepresenting.

Construct validity

Construct validity mainly concerns the design of the experiment, whether it is correct or not. Inadequate preoperational explication of constructs is a relevant threat, this is handled by clearly the defining the used constructs and their meaning in chapter 2 of the report. The constructs that the experiment is built on, coverage criteria and mutation analysis, are clearly defined. Mono method bias is another threat that needs to be taken into consideration. Mutation score is the only metric used for measuring efficiency at finding faults. However, as presented in section 2.4.3, mutation analysis has been shown to be an efficient and reliable way of measuring the effectiveness of test suites.

External validity

In order to achieve external validity, the results needs to be generalizable. In the context of the experiment it means that the software artifacts should be representative of software in the industry. The external validity of the study is therefore not granted, as the pool of programs is small and they are relatively simple open source programs. In order for the result to be more generalizable, the programs should in the best case represent industrial software products. It is hard to tell whether the testing tool and strategy are generalizable as neither the tools nor the method has reached a state of maturity in the industry. It should be noted that even if the results may not be generalizable, the study can still provide useful information and observations from which more finely grained research questions can be found.

The chosen software for performing mutation analysis however seems to be the most actively developed tool that exists for C#.

4.3 Ethical aspects

The most significant ethical aspect in the experiment is that the result should be possible to reproduce for another person using the supplied data. This is made possible by clearly formulating the various criteria, supplying the source code for the programs, the graphs of the programs, the created test cases and information about the software used to perform the mutation analysis. Using this information, anyone should be able to reach the same results that was achieved in the experiment.

5 **Method** Implementation

As previously stated, the experiment consists of creating a test suites for each criteria for three different programs. The programs were selected from the pool of programs used in *Introduction to Software Testing* (Ammann & Offutt, 2006) and the programming language is C#. The programs were chosen with the following criteria in mind: varying structures, feasible size and suitability for mutation testing. The programs should have varying structures in order to better represent reality. Certain criteria may be more or less effective depending on the structure of the program. For example, a program having few or very simple predicates will not be thoroughly tested using Active Clause Coverage, as it would largely collapse to clause and predicate coverage. The programs can neither be too big, as the test cases are created manually and bigger programs would not be possible due to time constraints. Suitability for mutation testing is also an important aspect and this overlaps with the programs having different structures. This means that there has to be sufficient syntax that the mutation operators can apply to. The source code for the selected programs are presented in appendix A-C.

5.1 Generating test requirements

Before creating any test cases the requirements must first be generated. For the graph based criteria, this requires first creating control flow graphs for each program. The process for creating the graphs is described in section 2.2. The graphs are modelled with *defs* and *uses* represented as those are required for the data-flow coverage criteria. With the graphs created test requirements can then be generated. For the experiment, a web tool created by Wuzhi Xu et al. (2015) as a supplement for the book by Ammann & Offutt(2006) was used. The requirements were then manually reviewed in order to determine which were infeasible.

5.1.1 Cal

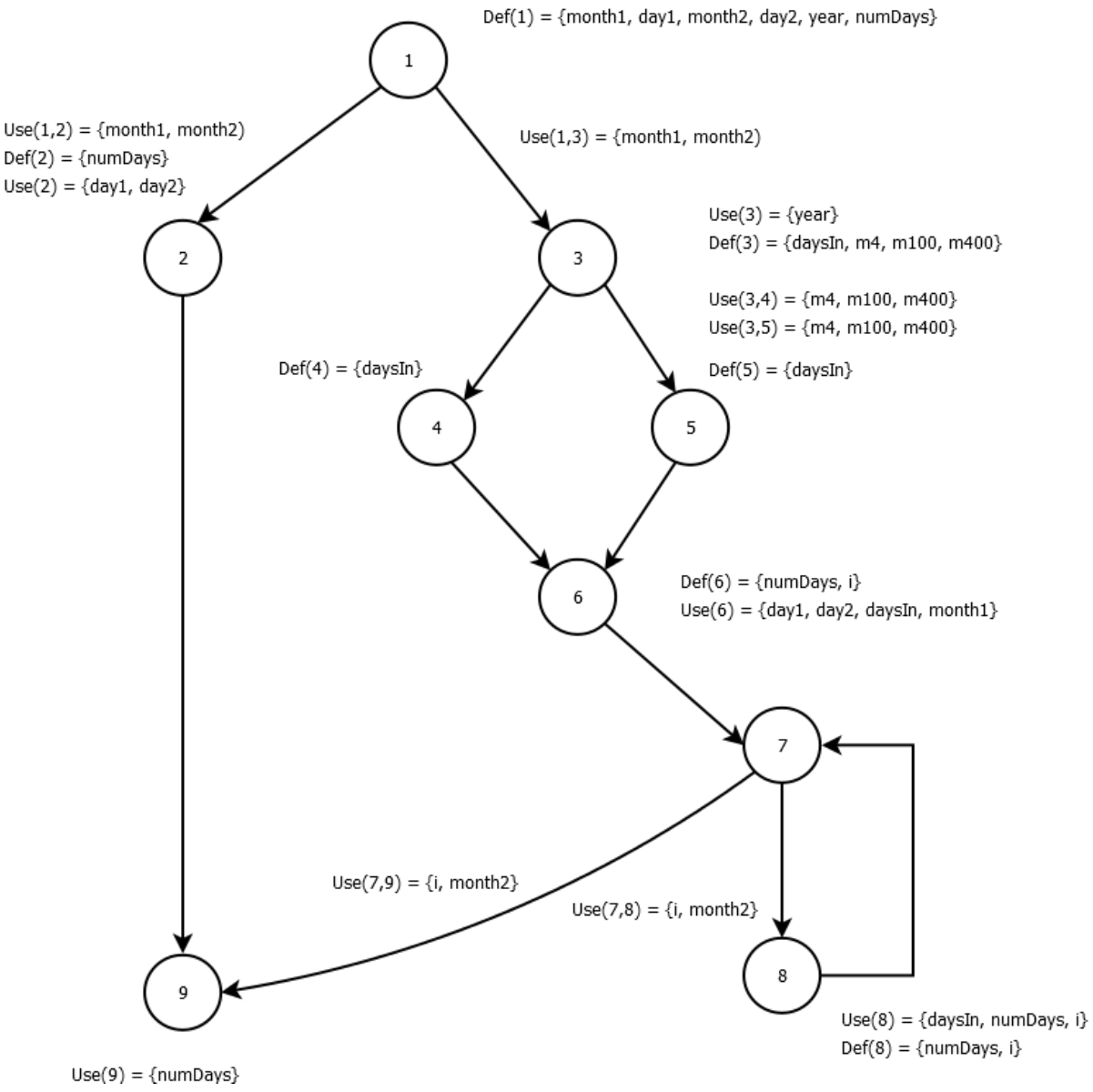

Figure 6- Control Flow Graph for the Cal program

The program Cal counts the days between two dates. To do this, it takes five arguments: month1, day1, month2, day2 and year. As can be seen from the graph it has a relatively simple structure with only a few branches and a loop. The output is the amount of days between the two chosen dates. The source code is presented in appendix A.

5.1.2 TriTyp

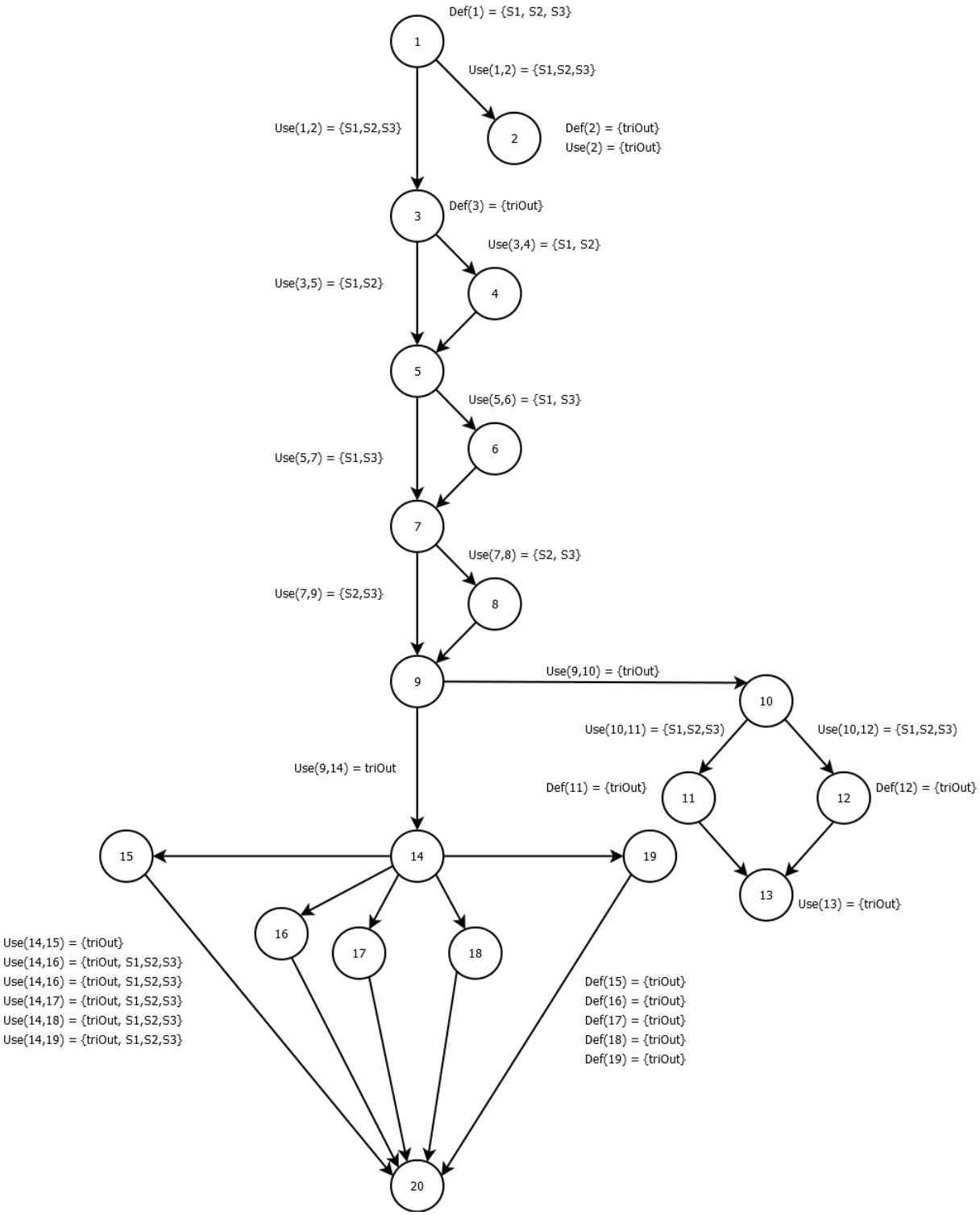

Figure 7 - Control Flow Graph for the TriTyp program

TriTyp is a program that classifies triangles into one of the following four categories:

- Scalene represented by output 1
- Isosceles represented by output 2

- Equilateral represented by output 3
- Not a triangle represented by output 4

Compared to the other two programs TriTyp has a higher cyclomatic complexity, meaning that it has a higher amount of branches. It takes three input values: Side1, Side2 and Side3. The source code is presented in appendix B.

5.1.3 TestPat

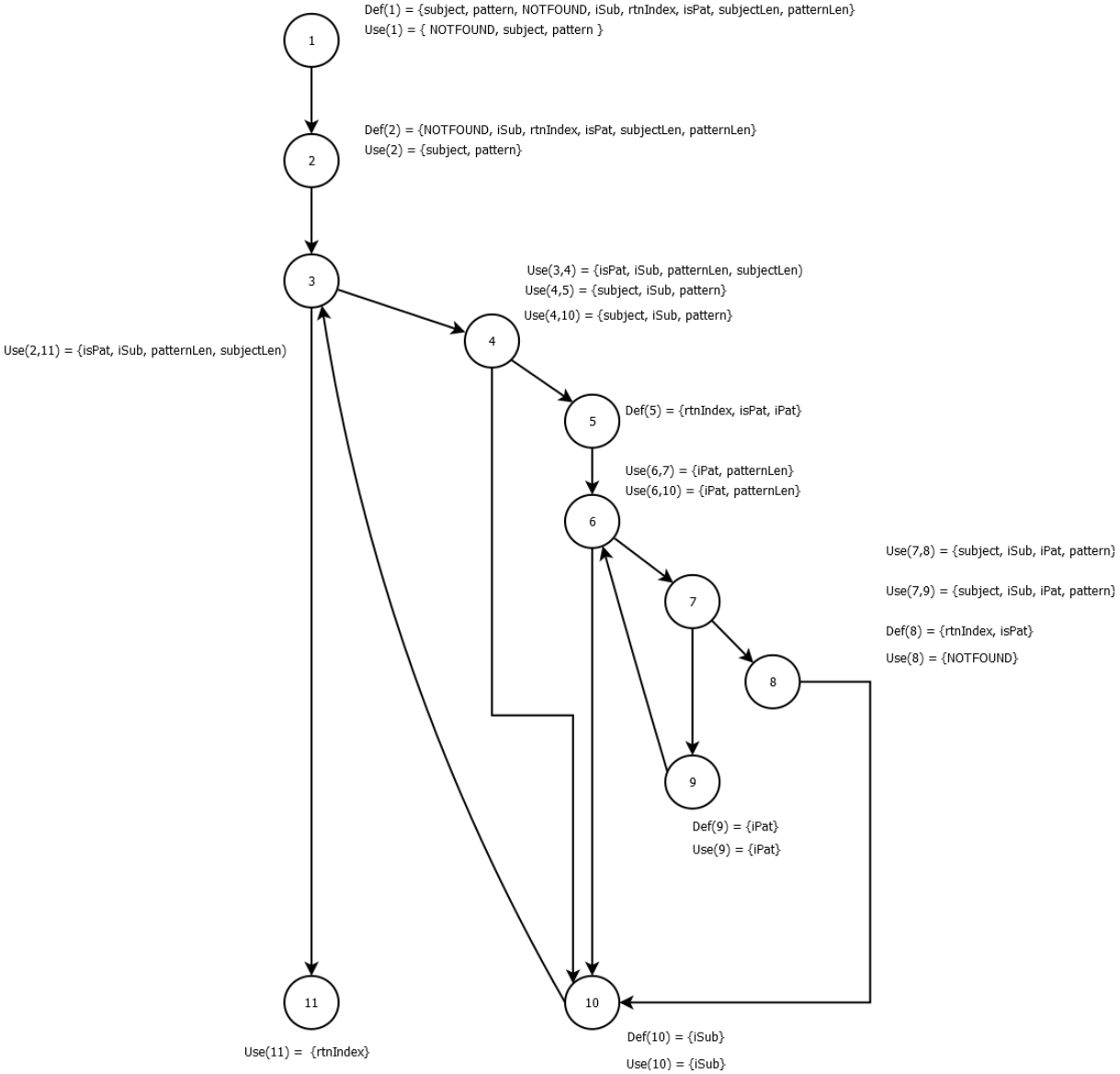

Figure 8 - Control Flow Graph for TestPat

TestPat tests if a given pattern has a match in a string. It takes two arguments: subject and pattern. If a match is found the index of where the pattern was found is returned. If no match is found -1 is returned.

5.1.4 Dealing with infeasible test requirements

Part of the test requirements may be infeasible. For graph coverage, this may be a path that is impossible to traverse. This is possible because the test requirements that are generated from a graph are not taking the program's logic into consideration. For logic coverage, infeasible requirements are common for active and inactive clause coverage. Either viable pairs of values may not exist at all or the requirements may be impossible to fulfill. For example, the predicate *year % 4 != 0 || (year % 100 == 0 & year % 400 != 0)* yields a requirement for RACC that requires both the first clauses to be true. In order for this to be fulfilled, year needs to be a value that is evenly divided by 100 but not by 4, which is impossible. In the experiment, infeasible requirements are simply ignored. When criteria are ignored, test cases are added when required to ensure that the subsumption relationship between criteria as described in figure x (LÄGG TILL FIGUR I BAKGRUND) is held. This is the approach recommended by Ammann & Offutt (2006).

5.2 Creating the test cases

Test cases were created from the requirements found in the previous step. Test cases were created to ensure that every requirement was met for every criteria. Determining input values for the test cases in such a way that they meet the requirements requires a significant overhead and puts a restraint on the magnitude of the experiment. The created test cases for each program is presented in appendix E-G.

5.3 Mutation Analysis

Mutation analysis is performed on each individual test suite, having test cases for a coverage criteria. The chosen tool to carry out the mutation analysis is Visual Mutator (Warsaw University of Technology, 2016). The tool was chosen due to being the most actively developed of the tools available for C# as well as its ease of integration into the programming environment.

5.4 Measuring cost effectiveness

The second objective builds upon the result of the first objective. Each criteria has three test suites, one for each program. To measure efficiency, the relationship between test suite size and mutation score is used. This is the same measure as used by Andrews et al. (2006) in their study. As they point out, it is not a perfect measure. It is however hard to create a generalized measurement because there are so many different factors that plays a role in the cost of a criteria. Certain criteria requires higher degrees of of manual analysis and this is not necessarily reflected in test suite size.

6 Results

This section of the report first presents the result of performing the experiment as described in section 5. A mutation score was acquired for each examined criteria for each individual program. Resulting in a total of 33 mutation scores. This section begins by looking at the results for each individual program, presenting a short summary of the results followed by observations for the criteria where relevant. This is then followed by an analysis of each criteria summarized for all the programs.

6.1 Program 1 – Cal

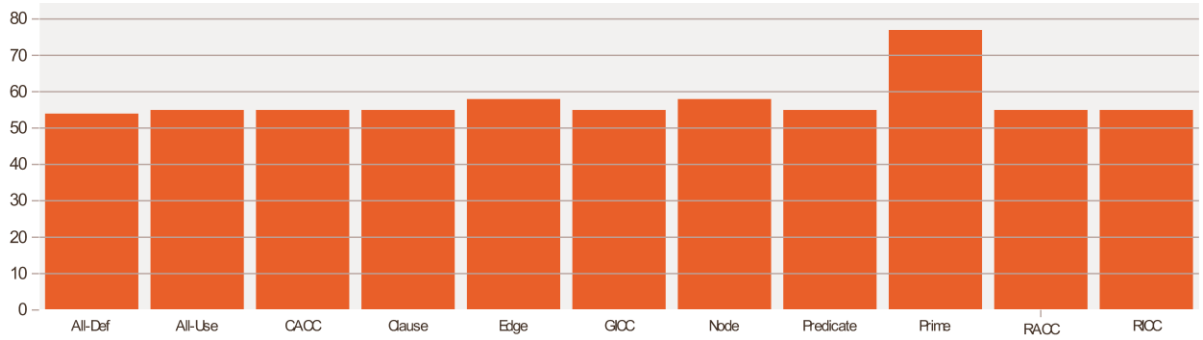

Figure 9 - Mutation scores for Cal

Figure 5 above presents the mutation scores for each individual criteria for the Cal program. As can be seen, the majority of criteria range between 50 and 55%. Prime path coverage however shows a more notable increase in effectiveness over the other criteria. Table 2 presented below shows mutation score along with the amount of test cases and measured cost effectiveness.

Type	Criteria	Test Cases	Mutation Score	MS / #Tests
Graph	Node	3	58%	19,3
Graph	Edge	3	58%	19,3
Graph	All-Def	3	55%	18,3
Graph	All-Use	6	56%	9,3
Graph	Prime	5	76%	15,8
Logic	Predicate	3	55%	18,3
Logic	Clause	3	55%	18,3
Logic	CACC	5	55%	11
Logic	RACC	5	55%	11
Logic	GICC	3	55%	18,3

Logic	RICC	3	55%	18,3

Table 2 - Test data for Cal

6.1.1 Graph based criteria

Looking at the graph-based criteria there are a few quite interesting observations. Firstly, the result of the graph-based criteria is mostly around 55% with the exception of prime-path coverage which is higher. This is interesting because prime-path coverage does not even have the most test paths. Showing that more test cases does not necessarily mean a higher mutation score. Something interesting and unexpected is that All-Use coverage performed worse than node coverage, despite the fact that All-Use coverage also met the requirements for node coverage. Further studying the results of the mutation analysis showed that this is due to the effect of different input variables. There may be thousands of different input combinations to meet a specific criteria requirement, and as shown in this case, the efficiency of different combinations may differ. The program calculates the amount of days between two dates and the amount of days in February differs depending on if it's a leap year or not. In the test cases for node-coverage, one test-case counted the amount of days between a day in February and a day in mars. There were no such test case in the test suite for All-Use coverage however. The effect of this is that the mutant that made February to 29 days even if it was not a leap year was eliminated by the test case in the node coverage suite but not from any in the all-use coverage suite, because the latter didn't have any test case where the output was dependent on the amount of days in February. This shows a significant weakness in depending on graph-based criteria alone, as they do not place any requirements on the arguments other than that they should traverse a certain path.

6.1.2 Logic based criteria

For the logic based criteria, the results are interestingly exactly the same, 55%. The result is not a big surprise for predicate, clause, GICC and RICC as they all have the same amount of test cases. The program has relatively few predicates as well so the efficiency of testing them is not expected to be high. GICC and RICC had no feasible test requirements and has both essentially collapsed to node and edge coverage. What's more interesting is CACC and RACC, they have more test cases yet no increase in mutation score. Investigations in the result of the mutation analysis showed that this was also due to the effect of different arguments. All of the other criteria had at least one test case that involved a day in February and another month, the suites for CACC and RACC had not. CACC and RACC still meets the test requirements as none of them requires that February is included. Changing one of the test cases to include February increased the mutation score to 70%, an increase of 15%. This further strengthens the indication that criteria alone may be insufficient for efficient testing.

6.2 Program 2 - TriTyp

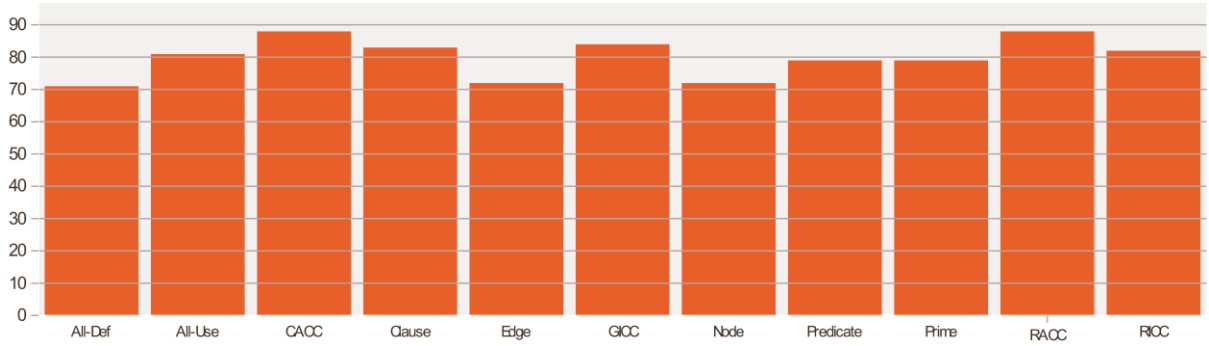

Figure 10 - Mutation scores for TriTyp

TriTyp is significantly more predicate heavy than the other two programs, it has more than twice as many predicates as either of them. It was therefore expected that the logical criteria should perform better for this program. This can be confirmed by looking at the result as presented in Figure 10 above. RACC and CACC performed best of all criteria with a mutation score of 88%. Of the graph based criteria, All-Use performed best among the graph criteria, which is curious when prime path subsumes it. The reason for this is presented in section 6.2.1 below.

Type	Criteria	Test Cases	Mutation Score	MS / #Tests
Graph	Node	8	72	9
Graph	Edge	8	72	9
Graph	All-Def	8	71	8,9
Graph	All-Use	9	81	9
Graph	Prime	10	79	7,9
Logic	Predicate	12	79	6,6
Logic	Clause	14	83	5,9
Logic	CACC	19	88	4,6
Logic	RACC	19	88	4,6
Logic	GICC	16	84	5,3
Logic	RICC	17	82	4,8

Table 3 - Test results for TriTyp

6.2.1 Graph based criteria

The graph based criteria performs worse than the logic based ones but requires fewer test cases and has a higher measured efficiency. An unexpected result is that all-use coverage yielded a higher mutation score than prime path coverage, despite prime path coverage subsuming all-use coverage and having more test cases. As with the previous cases, this effect can be assigned to the effect of different arguments. The table below presents examples of pairs of test cases from the suite for all-use and prime path coverage, where each pair of test cases traverses the exact same path.

Mutant	Criteria	Test path	S1	S2	S3	Mutant Status
S1+S2 <= S3 mutated to S2 <= S1	All-Use	[1,3,5,7,9,10,12,13]	42	41	50	Killed
	Prime	[1,3,5,7,9,10,12,13]	10	15	21	Live
S1+S3 <= S2 mutated to false	All-Use	[1,3,5,7,9,10,11,13]	13	38	402	Live
	Prime	[1,3,5,7,9,10,11,13]	10	70	21	Killed
S1+S3 > S2 mutated to S1 > S2	All-Use	[1,3,5,6,7,9,14,17,20]	140	200	140	Killed
	Prime	[1,3,5,6,7,9,14,17,20]	132	69	132	Live

Table 4 - Pairs of test cases for TriTyp yielding different results. S1, S2 and S3 are arguments.

As the table shows, test cases traversing the same path may not detect the same errors depending on the arguments.

6.2.2 Logic based criteria

All of the logic based criteria performed better than the logic based criteria. This result was expected as the program is relatively predicate heavy and each logic based criteria required more test cases than any graph based. CACC and RACC are both more effective than GICC and RICC, despite inactive clause coverage having more requirements than active. This is because most of the requirements for GICC and RICC are infeasible, resulting in fewer test cases. One can observe that RICC is less effective than clause coverage, despite subsuming and having more test cases. This can again be explained by the effect of different arguments. Analysis of the mutation analysis data shows that for some test cases that met the exact same requirements, the effectiveness of the tests varies widely.

6.3 *Program 3 – TestPat*

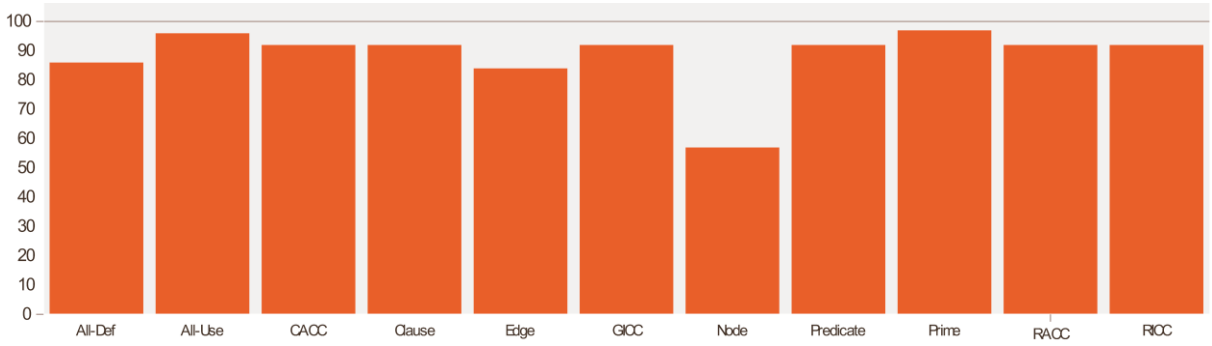

Figure 11 - Mutation scores for TestPat

The effectiveness of the criteria is generally visibly higher when compared to the other programs. The reason for this is unclear, it can be speculated that part of the reason is that the majority of the program runs in a loop. This means that several statements are executed multiple times and with multiple values. Prime path coverage proved to be the most efficient, very closely followed by All-Use.

Criteria	Test Cases	Mutation Score	MS / #Tests
Node	1	57	57
Edge	3	84	28
All-Def	8	86	10,8
All-Use	12	96	8
Prime	10	97	9,7
Predicate	5	92	18,4
Clause	5	92	18,4
CACC	5	92	18,4
RACC	5	92	18,4
GICC	5	92	18,4
RICC	5	92	18,4

Table 5 - Test results for TestPat

6.3.1 Graph based criteria

This is the first of the programs that had no truly unexpected results. Something noteworthy is that all-use coverage that has more test cases than prime path coverage despite prime path coverage subsuming it. This is due to the fact that the test cases in the prime path coverage suite covers longer test paths.

6.3.2 Logic based criteria

As can be seen, the results for all logic based criteria are exactly the same. In fact, as shown in the table presenting the test cases in appendix G, they have the exact same test cases. This is mostly due to the program having few and simple predicates.

6.4 *Summarized results*

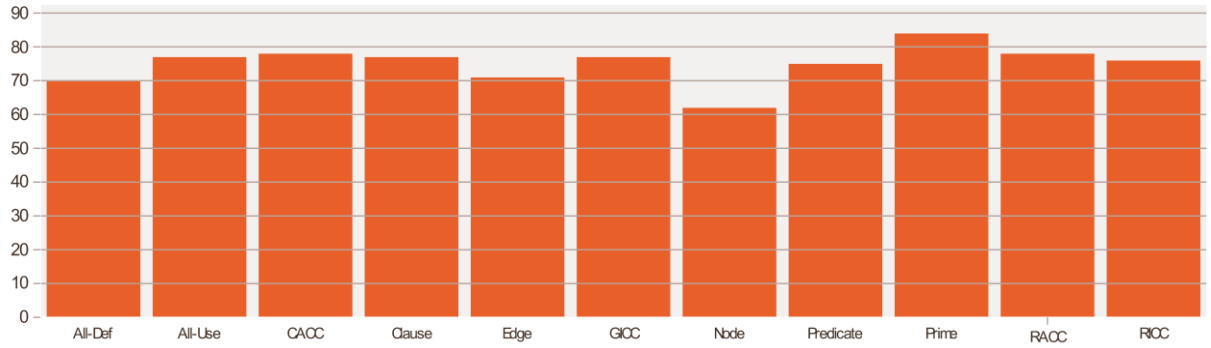

Figure 12 - Mutation score mean for the three programs

The figure above presents a chart showing the mean mutation score for each criteria. From this measure, prime path coverage has been the most efficient at eliminating mutants and node coverage the least. Analysis of the individual programs however showed that the many different arguments that can be chosen to meet test requirements may cause a big difference in efficiency. This means that no truly significant conclusions can be made regarding efficiency from the experiment data. Useful observations can still be made and be used as a starting point for further research. It is interesting to see that the result for the logic criteria barely differs at all. Looking at the analysis for the individual programs this is not surprising. Two of the three programs only has relatively simple predicates (one or two clauses) and all of the requirements even for the stricter criteria could be met with the same amount of test cases. The program that has a significantly higher amount of predicates, TriTyp, presents a bigger difference among the logic criteria.

Criteria	Mean Mutation score	Median	Mean Efficiency	Mean #Test Cases	Min	Max
Node	62	58	28,4	4	57	72
Edge	71	72	18,8	4,7	58	84
All-Def	70	71	12,7	6,3	54	86
All-Use	77	81	8,8	9	55	96
Prime	84	79	11,1	8,3	77	97
Predicate	75	79	14,4	6,7	55	92
Clause	77	83	14,2	7,3	55	92
CACC	78	88	11,3	9,7	55	92
RACC	78	88	11,3	9,7	55	92

GICC	77	84	14	8	55	92
RICC	76	82	13.8	8,3	55	92

Table 6 - Experiment results

Table 6 above also shows the median, min and max values for each individual test criteria as well as the mean efficiency. From the individual results of each criteria it can be observed that the result for several criteria varied widely across the programs. This suggests that the effectiveness of criteria may be widely different from program to program. This can possibly be attributed to several factors. Firstly, the structure of the tested programs. Some programs may lend themselves more or less towards certain criteria. A program consisting of very few and/or simple predicates may not be efficiently tested by logic based testing criteria. Similarly, a program of very high complexity saturated with predicates with several clauses may not be as efficiently tested by graph based criteria. The experiment data is however insufficient to draw any such conclusions. It would be interesting to see if certain criteria are better for eliminating certain types of mutants. This could be used for gaining insight about under which circumstances a criteria may be more effective. This is however out of the scope of this study, and Visual Mutator does not have functionality for presenting such metrics.

Regarding cost efficiency, node coverage can be seen to have the highest amount of detected faults per test case and all-use the lowest. A pattern can be seen from the results, the cost of achieving stricter criteria is not strictly linear to the mutation score. This pattern does not fit to all criteria, for example prime path has a higher cost-efficiency than all-use despite being harder to achieve. This can partly be explained by the fact that the tool that was used to generate requirements created longer test paths, and therefore fewer test cases per requirement than it does for all-uses coverage.

7 Discussion

In this section of the report a discussion of the result is presented. Firstly the identified limitations of the method implementation are discussed. This is then followed by the interpretation of the result, the unexpected results and finally a comparison

7.1 Limitations of the method implementation

During the analysis it was made clear that the results of the experiment had more limitations than was first expected. Mostly due to the unforeseen effect of argument values for the test cases. This effect meant that the results could look very different if the experiment was performed again with different arguments for the test cases. In order to better measure the effectiveness of the criteria, the experiment should be of a significantly larger scale in order to allow meaningful statistical analysis. A larger pool of programs should be used as well as more test suites for each criteria. In order for this to be feasible, parts of the experiment needs to be automated. Manually generating test requirements, reviewing them and then finally creating test cases takes too much time. A method for fully automating the testing process for coverage criteria has not been found by the author however.

The method used for measuring cost effectiveness is also suboptimal. Using only the ratio of test suite size to mutation score does not give the whole picture. Using this criteria it is possible to achieve the same cost effectiveness for a GICC test suite that had very few feasible requirements and ultimately collapsed to the same amount of test cases as a test suite for clause coverage. In this case the overhead of manual review for determining whether requirements are feasible or not as well as making sure that each requirement is covered is not considered. During the course of the experiment, the effort required for the review of requirements could be as high as the creation of the test cases themselves. No other better method for measuring cost effectiveness has been identified however. There are too many variables that play a part in the testing process to create a good universal metric.

7.2 Interpretation of the result

Due to the small size of the experiment, the methods for interpreting results are limited. It would be desirable with a larger set of data to be able to perform more rigorous statistical analysis. When interpreting the results, various factors were taken into consideration. If for example a coverage criteria had a very low score then it had to be established if there were any underlying reasons. This could for example be if there were only few feasible test requirements, if the structure of the program did not lend itself well to testing with the given criteria, resulting in few test requirements. It is possible that the result could be interpreted in another way.

7.3 Unexpected results

The experiment yielded several unexpected results, the biggest being that some criteria performed worse than criteria they were subsuming. Intuitively this was not thought possible at first. The cause of this was shown to be due to the effect of different argument values. Depending on the arguments chosen, the effectiveness of the test cases were shown to vary widely, despite them covering the same test requirements. This means that when arbitrarily choosing values to meet a test requirement, the effectiveness of the tests will have a random element. As mentioned previously,

this means that no true conclusions can be drawn regarding effectiveness when performing an experiment of a smaller scale such as the one performed in this study. These results can however show that criteria alone may not be sufficient in order to achieve efficient testing.

7.4 Related work

7.4.1 How Effective are Code Coverage Criteria?

A study by Hadi Hemmati (2015) compared statement, branch, MC/DC, def-use and loop coverage. The first four can be directly compared with node coverage, edge coverage, active clause coverage and all-use coverage as they should be very similar. The experiment design in the study is significantly different. A database with known faults and test cases were used. For each fault two test suites are used, one that did not detect the fault and another improved version that causes at least one test to fail. Hemmati (2015) makes the assumption that if the coverage for criterion *A* has increased then the criteria can be deemed as suitable for identifying the fault. This is then made for a total of 357 known faults. This is different from the method used in the experiment of this bachelor thesis, where a test suite is created for each criteria with 100% coverage. The following results are presented by Hemmati (2015):

- Statement(Node) coverage found 10% of all faults
- Branch(Edge) coverage found 19% of all faults
- MC/DC(Active Clause) coverage found 19% of all faults
- Def-Use(All-Use) coverage found 79% of all faults

Something interesting is that his results for statement, branch and MC/DC are very different from the results found from this experiment. Def-Use is however very close the result found for All-Uses in the experiment. It's important to once again notice that the method used for measuring effectiveness is different, which likely is the biggest reason for the results is being so different. It is difficult to objectively tell whether one method is better than another. Mutation has however shown to be a reliable method for measuring effectiveness, as presented in section 2.4.3, making it reasonable to believe that the method for carrying out the experiment in this bachelor thesis bears merit.

7.4.2 An Experimental Comparison of Four Unit Test Criteria: Mutation, Edge-Pair, All-uses and Prime Path Coverage

Another study by Li, Praphamontripong and Offutt (2009) investigated mutation, edge-pair, all-uses and prime path coverage. The result of the latter two are interesting to look at as they are directly comparable to the results of the experiment of this paper. It should however be noted that the experiment method is not strictly the same. Instead of using only mutation analysis to seed artificial faults into the program the authors used mostly hand-seeded faults. The authors had also taken into consideration the effect that different test values meeting a criteria can be more or less efficient. They refer to this as "test value noise". In order to reduce this they used a pool of test cases to ensure that the same requirements always had the same set of values. This does reduce the effect but it does not eliminate it. In order to measure cost effectiveness, the authors used the ratio of the number of tests over the number of found faults. Li et al. (2009) presents the following results:

- All-Uses coverage found 54 of 88 faults (61%)
- Prime path coverage found 56 of 88 faults (63%)

This is a notable difference both from the results of this experiment as well as the result of Hemmati´s (2015) study presented above. It is unclear what this may be attributed to, likely it is due to the difference in experiment design. It may also be due to the tested programs themselves. It can still be observed that despite the percentage of found faults are widely different, the difference between prime-path and all-use is closer. Li et al.'s (2009) study found a 2 percentage point difference between the two criteria while the experiment in this paper found a 7 percentage point difference.

Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria

Andrews, Briand, Labiche and Namin (2006) uses mutation analysis to assess the effectiveness of various coverage criteria and also investigate whether mutation score is a good predictor of the actual fault detection rate. The investigated criteria are Block, Decision, C-use and P-use (Rapps & Weyuker, 1985). An experiment is made using both mutation analysis as well as known faults to measure effectiveness. A correlation is found between the detection rate of known faults and the detection rate of mutants, suggesting that mutation score is a good predictor. The authors also investigate cost-effectiveness as the relation between test suite size and fault detection rate. The experiment differs from the one conducted in this bachelor thesis in several ways. The set of coverage criteria is not the same, the experiment focuses on a smaller set of coverage criteria that are not strictly the same. Programming language also differs, their experiment is focused on a single program in C.

7.4.3 Conclusion of the comparisons

The results of the compared studies were different from the result achieved in this experiments paper where they could be compared. The compared studies had also achieved a different result to one another. This suggests that the generalizability of the experiment, and possibly the experiments of the other studies as well, is relatively low.

7.5 Conclusions

Despite the limitations of the experiment, it still gave several important insights. Firstly, the unexpected results showed that the "test noise" effect had a significant impact on effectiveness. This showed that by only using coverage criteria as a basis for testing, the effectiveness will have a random element. This effect will also be very important to take into consideration in further studies. Another important insight is that the effectiveness for different criteria varied very widely from program to program. This suggests that it may not be feasible to rely on a single criteria when choosing a testing strategy, as some others may be more suitable depending on the program structure. Identifying which structures are more effective for which criteria would be an interesting research question in further studies.

8 Conclusion and Future Work

8.1 Summary

The purpose of this paper has been to examine and compare the effectiveness of coverage criteria by using mutation analysis and the programming language C#. Effectiveness has been measured in terms of effectiveness at finding faults as well as cost effectiveness. An experiment was conducted where test cases where created for a total of 11 test criteria and 3 different programs, resulting in 33 test suites that mutation analysis was performed on. The result presented several interesting insights. Firstly, the effect referred to as “test noise” (see 7.4.2) was not taken into consideration in the experiment design, this effect proved to influence the result very significantly. This means that the result may look different if the experiment was performed again with the effect taken into consideration. A method to reduce the test noise effect could be to pick a number of random arguments for each test requirements. It was also shown that coverage criteria alone may be insufficient for efficient testing, as they ignore several important aspects of the tested software. Among the tested criteria, prime path coverage was deemed to be the most efficient and consistent. When comparing the result to similar studies, it was shown to be different. This may suggest that the result has a low generalizability in terms of effectiveness. However, the observations that were made should still be generalizable and provide a good starting point for further studies.

8.2 Contributions

The study has compared and evaluated 11 different test criteria. Only a subset of them has been compared in other studies and none of them using C# and mutation analysis. While analysis of the result showed that no true conclusions could be drawn regarding efficiency, several useful observations were made. It was shown that by using only criteria as a basis for testing, several important aspects of a program may not be tested. Both due to the “test noise” effect as well as the fact that coverage criteria does not place any requirements on what the program is intended to do. This suggests that testing strategies only requiring that certain criteria is met may need to be re-evaluated. The result data may also serve as a starting point and as reference for further examinations regarding coverage criteria.

8.3 Future work

During the course of the study several opportunities for future work was identified. First and foremost is to conduct a similar experiment but on a larger scale. The study should then encompass several more programs, possibly of higher complexity and size to increase generalizability and statistical power. Having a bigger set of data would make it feasible to use established quantitative methods, this would leave the result less open to interpretation. In order to reduce the effect of the “test noise” effect, several test suites with randomly chosen values should be created for each criteria for each tested program. In order to make an experiment of this magnitude more feasible, possible options for automatization should be researched. It would also be interesting to perform a deeper analysis of the criteria, in particular comparing the different types of criteria. The analysis could examine if certain types of criteria are more efficient at eliminating certain types of mutants. This could then be used to determine if certain criteria are more suitable for certain types of programs structures. Another aspect that could be interesting to examine is the effectiveness of combinations of criteria.

9 References

Ammann, P. & Offutt, A. J. (2008). *Introduction to Software Testing.* Cambridge: Cambridge University Press

Andrews, J.H., Briand, L.C. & Labiche, Y. (2005) *Is mutation an appropriate tool for testing experiments?*. ICSE 2005. Proceedings. 27th International Conference on Software Engineering. pp. 402-411, 15-21 May. DOI: 10.1109/ICSE.2005.1553583

Andrews, J.H., Briand, L.C., Labiche, Y. & Namin, A.S. (2006) *Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria*. IEEE Transactions on Software Engineering, 32(8) pp.608-624. DOI: 10.1109/TSE.2006.83

Baldwin, D. & Sayward, F.G. (1979). *Heuristics for Determining Equivalence of Program Mutations*. (Report 276). New Haven, Conneticut: Department of Computer Science, Yale University.

Baker, R & Habli, I (2013) *An Empirical Evaluation of Mutation Testing for Improving the Test Quality of Safety-Critical Software*. IEEE Transactions on Software Engineering, 39(6) pp.787-805. DOI: 10.1109/TSE.2012.56

Gopinath, R. Alipour, A. Ahmed, I. Jensen, C. & Groce, A. (2015*) How hard does mutation analysis have to be, anyway?*. 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE). Gaithersburg, USA 2-5 Nov. 2015. pp.216-227.
DOI: 10.1109/ISSRE.2015.7381815

Hemmati, H. (2015) *How Effective Are Code Coverage Criteria?.* 2015 IEEE International Conference on Software Quality, Reliability and Security (QRS). Vancouver, USA 3-5 Aug. pp.151-156. DOI: 10.1109/QRS.2015.30

Jia, Y. & Harman, M. (2011) *An Analysis and Survey of the Development of Mutation Testing*. IEEE Transactions on Software Engineering, 37 (5) pp. 649-678. DOI: 10.1109/TSE.2010.62

Li, N, Praphamontripong, U. & Offutt, J. (2009) *An Experimental Comparison of Four Unit Test Criteria: Mutation, Edge-Pair, All-Uses and Prime Path Coverage*, International Conference on Software Testing, Verification and Validation Workshops, 2009., Denver, USA, pp. 220-229. DOI: 10.1109/ICSTW.2009.30

Just, R., Jalali, D., Inozemtseva, L., Ernst, M.D., Holmes, R. & Fraser, G. (2014). *Are mutants a valid substitute for real faults in software testing?.* Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. New York, USA, pp. 654-665. DOI: 10.1145/2635868.2635929

Offutt, A. J. (1989). *The Coupling Effect. Fact or Fiction*. ACM SIGSOFT Software Engineering Notes, 14(8) pp 131-40.

Offutt, A. J. (1992). *Investigations of the software testing coupling effect*. ACM Transactions on Software Engineering and Methodology, 1(1) pp.5-20

Offutt, A.J. (1997). *Automatically Detecting Equivalent Mutants and Infeasible Paths*. The Journal of Software Testing, Verification and Reliability, 7(3) pp.165-192

Ramler, R. & Kaspar, T. (2012) *Appplicabilty and benefits of mutation analysis asa n aid for unit testing*. 2012 7th International conference on Computing and Convergence Technology. Seoul, South Korea 3-5 Dec, pp.920-925

Rapps, S. and Weyuker, E. (1985). Selecting Software Test Data Using Data Flow Information. *IEEE Transactions on Software Engineering*, SE-11(4), pp.367-375

Warsaw University of Technology (2016). *Visual Mutator* (Version 2.0). [Software] Available: http://visualmutator.github.io/web/

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M., Regnell, B. and Wesslén, A. (2012). *Experimentation in Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg.

Appendix

A. Cal.java.cs

```
/*
 * Aresh Saharkhiz
 * saharkiz@gmail.com
 * Associate Professor / Mapua Institute of Technology / Philippines
 */

using System;
using System.IO;

// Introduction to Software Testing
// Authors: Paul Ammann & Jeff Offutt
// Chapter 3, section 3.4, page 132

internal class cal_class
{

    public static int cal(int month1, int day1, int month2, int
day2, int year)
    {

//*********************************************************
        // Calculate the number of Days between the two given days
in
        // the same year.
        // preconditions : day1 and day2 must be in same year
        //               1 <= month1, month2 <= 12
        //               1 <= day1, day2 <= 31
```

```
        //                month1 <= month2
        //                The range for year: 1 ... 10000

//*********************************************************
        int numDays;


        if (month2 == month1) // in the same month
        {
            numDays = day2 - day1;
        }
        else
        {
            // Skip month 0.
            int[] daysIn = new int[] { 0, 31, 0, 31, 30, 31, 30, 31,
31, 30, 31, 30, 31 };
            // Are we in a leap year?
            int m4 = year % 4;
            int m100 = year % 100;
            int m400 = year % 400;
            if ((m4 != 0) || ((m100 == 0) && (m400 != 0)))
            {
                daysIn[2] = 28;
            }
            else
            {
                daysIn[2] = 29;
            }

            // start with days in the two months
            numDays = day2 + (daysIn[month1] - day1);
```

```
        // add the days in the intervening months
        for (int i = month1 + 1; i <= month2 - 1; i++)
        {
            numDays = daysIn[i] + numDays;
        }
    }
    return (numDays);
}

public static void Main(string[] argv)
{ // Driver program for cal
    int month1, day1, month2, day2, year;
    int T;

    Console.WriteLine("Enter month1: ");
    month1 = N;
    Console.WriteLine("Enter day1: ");
    day1 = N;
    Console.WriteLine("Enter month2: ");
    month2 = N;
    Console.WriteLine("Enter day2: ");
    day2 = N;
    Console.WriteLine("Enter year: ");
    year = N;
    // preconditions : day1 and day2 must be in same year
    //                 1 <= month1, month2 <= 12
    //                 1 <= day1, day2 <= 31
    //                 month1 <= month2
```

```
        //               The range for year: 1 ... 10000


        if ((month1 < 1) || (month1 > 12))
        {
            month1 = 1;
            Console.WriteLine("invalid month1, choosing 1.");
        }
        if ((month2 < 1) || (month2 > 12))
        {
            month2 = 1;
            Console.WriteLine("invalid month2, choosing 1.");
        }
        if ((day1 < 1) || (day1 > 31))
        {
            day1 = 1;
            Console.WriteLine("invalid day1, choosing 1.");
        }
        if ((day2 < 1) || (day2 > 31))
        {
            day2 = 1;
            Console.WriteLine("invalid day2, choosing 1.");
        }
        while (month1 > month2)
        {
            Console.WriteLine("month1 must be prior or equals to
month2");
            Console.WriteLine("Enter month1: ");
            month1 = N;
            Console.WriteLine("Enter month2: ");
```

```
        month2 = N;
    }
    if ((year < 1) || (year > 10000))
    {
        year = 1;
        Console.WriteLine("invalid year, choosing 1.");
    }

    T = cal(month1, day1, month2, day2, year);

    Console.WriteLine("Result is: " + T);
}

// ====================================
// Read (or choose) an integer
private static int N
{
    get
      {
         int inputInt = 1;
         string inStr;

         try
         {
                  inStr = Console.ReadLine();
                  inputInt = Convert.ToInt32(inStr);
         }
         catch (IOException)
         {
```

```
                    Console.WriteLine("Could not read input,
choosing 1.");
            }
            catch (FormatException)
            {
                    Console.WriteLine("Entry must be a number,
choosing 1.");
            }

            return (inputInt);
        }
    } // end getN

}
```

B. TriTyp.java.cs

```
/*
 * Aresh Saharkhiz
 * saharkiz@gmail.com
 * Associate Professor / Mapua Institute of Technology / Philippines
 */

using System;
using System.IO;

// Introduction to Software Testing
// Authors: Paul Ammann & Jeff Offutt
// Chapter 3, section 3.3, page 121

// Jeff Offutt--Java version Feb 2003
// Classify triangles

internal class trityp
{
    private static string[] triTypes = new string[] { "", "scalene",
"isosceles", "equilateral", "not a valid triangle" }; // Ignore 0.
    private static string instructions = "This is the ancient TriTyp
program.\nEnter three integers that represent the lengths of the
sides of a triangle.\nThe triangle will be categorized as either
scalene, isosceles, equilateral\nor invalid.\n";

    public static void Main(string[] argv)
    { // Driver program for trityp
        int A, B, C;
        int T;
```

```
    Console.WriteLine(instructions);
    Console.WriteLine("Enter side 1: ");
    A = N;
    Console.WriteLine("Enter side 2: ");
    B = N;
    Console.WriteLine("Enter side 3: ");
    C = N;
    T = Triang(A, B, C);

    Console.WriteLine("Result is: " + triTypes[T]);
}

// ====================================
// The main triangle classification method
private static int Triang(int Side1, int Side2, int Side3)
{
    int triOut;

    // triOut is output from the routine:
    //    Triang = 1 if triangle is scalene
    //    Triang = 2 if triangle is isosceles
    //    Triang = 3 if triangle is equilateral
    //    Triang = 4 if not a triangle

    // After a quick confirmation that it's a valid
    // triangle, detect any sides of equal length
    if (Side1 <= 0 || Side2 <= 0 || Side3 <= 0)
    {
```

```
            triOut = 4;
            return (triOut);
        }

        triOut = 0;
        if (Side1 == Side2)
        {
            triOut = triOut + 1;
        }
        if (Side1 == Side3)
        {
            triOut = triOut + 2;
        }
        if (Side2 == Side3)
        {
            triOut = triOut + 3;
        }
        if (triOut == 0)
        { // Confirm it's a valid triangle before declaring
            // it to be scalene

            if (Side1 + Side2 <= Side3 || Side2 + Side3 <= Side1 ||
Side1 + Side3 <= Side2)
            {
                triOut = 4;
            }
            else
            {
                triOut = 1;
```

```
        }
        return (triOut);
    }

    // Confirm it's a valid triangle before declaring
    // it to be isosceles or equilateral

    if (triOut > 3)
    {
        triOut = 3;
    }
    else if (triOut == 1 && Side1 + Side2 > Side3)
    {
        triOut = 2;
    }
    else if (triOut == 2 && Side1 + Side3 > Side2)
    {
        triOut = 2;
    }
    else if (triOut == 3 && Side2 + Side3 > Side1)
    {
        triOut = 2;
    }
    else
    {
        triOut = 4;
    }
    return (triOut);
} // end Triang
```

```
    // ====================================
    // Read (or choose) an integer
    private static int N
    {
        get
          {
             int inputInt = 1;
             string inStr;

             try
             {
                      inStr = Console.ReadLine();
                      inputInt = Convert.ToInt32(inStr);
             }
             catch (IOException)
             {
                      Console.WriteLine("Could not read input,
choosing 1.");
             }
             catch (FormatException)
             {
                      Console.WriteLine("Entry must be a number,
choosing 1.");
             }

             return (inputInt);
          }
    } // end getN
```

```
} // end trityp class
```

C. *TestPat.java.cs*

```
/*
 * Aresh Saharkhiz
 * saharkiz@gmail.com
 * Associate Professor / Mapua Institute of Technology / Philippines
 */

using System;

// Introduction to Software Testing
// Authors: Paul Ammann & Jeff Offutt
// Chapter 2, section 2.3, page 56

internal class TestPat
{

    public static void Main(string[] argv)
    {
        const int MAX = 100;
        char[] subject = new char[MAX];
        char[] pattern = new char[MAX];
        if (argv.Length != 2)
        {
            Console.WriteLine("java TestPat String-Subject String-
Pattern");
            return;
        }
        subject = argv[0].ToCharArray();
        pattern = argv[1].ToCharArray();
```

```
        TestPat testPat = new TestPat();
        int n = 0;
        if ((n = testPat.pat(subject, pattern)) == -1)
        {
            Console.WriteLine("Pattern string is not a substring of
the subject string");
        }
        else
        {
            Console.WriteLine("Pattern string begins at the
character " + n);
        }
    }

    public TestPat()
    {
    }

    public virtual int pat(char[] subject, char[] pattern)
    {
        // Post: if pattern is not a substring of subject, return -1
        //       else return (zero-based) index where the pattern
(first)
        //       starts in subject

        const int NOTFOUND = -1;
        int iSub = 0, rtnIndex = NOTFOUND;
        bool isPat = false;
        int subjectLen = subject.Length;
        int patternLen = pattern.Length
```

```
;
        while (isPat == false && iSub + patternLen - 1 < subjectLen)
        {
            if (subject[iSub] == pattern[0])
            {
                rtnIndex = iSub; // Starting at zero
                isPat = true;
                for (int iPat = 1; iPat < patternLen; iPat++)
                {
                    if (subject[iSub + iPat] != pattern[iPat])
                    {
                        rtnIndex = NOTFOUND;
                        isPat = false;
                        break; // out of for loop
                    }
            }
            iSub++;
        }
        return (rtnIndex);
    }

}
```

D. Table of validity threats

Conclusion validity	
Low statistical power	See 4.2
Violated Assumption of statistical tests	No such tests were made
Fishing and the error rate	See 4.2
Reliability of measures	See 4.2
Reliability of treatment implementation	The mutation analysis is automated and performed in the exact same fashion for every criteria.
Random irrelevancies in experimental setting	Not relevant to the experiment
Random heterogeneity of subjects	Not relevant to the experiment
Internal validity	
History	The mutation analysis is automated and performed in the exact same fashion for every criteria, at any time.
Maturation	Not relevant, neither the subjects(criteria & test cases) or the treatment(mutation analysis) change during the course of the experiment.
Testing	To ensure that the result didn't change from application to application of the treatment, mutation analysis was performed at least twice on each criteria with the same result being achieved.
Instrumentation	See 4.2
Statistical regression	Not relevant
Selection	Not relevant
Mortality	Not relevant
Ambiguity about direction of causal influence	The experiment has a clear direction of causal influence. The test cases influences the mutation score.

Interactions with selection	Not relevant
Diffusion of imitation of treatments	Not relevant
Compensatory equalization of treatments	Not relevant
Compensatory rivalry	Not relevant
Resentful demoralization	Not relevant
Construct validity	
Inadequate preoperational explication of constructs	See 4.2
Mono-operation bias	?!?
	See 4.2
Confounding constructs and levels of constructs	???!
Interaction of different treatments	Not applicable
Interaction of testing and treatment	The treatment is automated and always consistent.
Restricted generalizability across constructs	?!
Hypotheses guessing	Not applicable, no persons involved
Evaluation apprehension	Not applicable, no persons involved
Experimenter expectancies	Not applicable, no persons involved
External validity	
Interaction of selection and treatment	
Interaction of setting and treatment	
Interaction of history and treatment	

E. Test Cases for Cal

Criteria	**month1**	**day1**	**month2**	**day2**	**year**	**Output**
Node	1	5	1	6	2001	1
	2	12	3	15	2017	31
	7	20	9	25	2000	67
Edge	1	5	1	6	2001	1
	2	12	3	15	2017	31
	7	20	9	25	2000	67
All-Defs	1	5	1	6	2001	1
	1	12	2	15	1993	34
	7	20	9	25	2000	67
All-Uses	7	20	9	25	2000	67
	1	12	2	15	1993	34
	1	12	2	15	1992	34
	1	5	1	6	2001	1
	5	3	8	19	2004	108
	7	20	9	25	2007	67
Prime	1	12	2	15	1993	34
	3	12	5	20	2005	69
	2	17	3	19	2000	31
	10	17	10	20	1969	3
	5	3	8	19	2004	108
Predicate	5	15	7	22	1995	68
	6	16	7	20	2000	34
	2	10	2	15	1993	5

Clause	5	15	7	22	1995	68
	6	16	7	20	2000	34
	2	10	2	15	1993	5
CACC	5	15	7	22	1995	68
	6	16	7	20	2000	34
	5	15	7	22	1996	68
	5	15	7	22	1900	68
	2	10	2	15	1993	5
RACC	5	15	7	22	1995	68
	6	16	7	20	2000	34
	5	15	7	22	1996	68
	5	15	7	22	1900	68
	2	10	2	15	1993	5
GICC	2	10	2	15	1993	5
	6	16	7	20	2000	34
	5	15	7	22	1995	68
RICC	2	10	2	15	1993	5
	6	16	7	20	2000	34
	5	15	7	22	1995	68

Table 7 - Test cases for Cal

F. Test cases for TriTyp

Criteria	Side1	Side2	Side3	Output
Node	-1	5	4	4
	310	310	310	3
	10	10	15	2
	16	7	16	2
	50	97	97	2
	500	100	100	4
	1	50	13	4
	43	51	38	1
Edge	-1	5	4	4
	310	310	310	3
	10	10	15	2
	16	7	16	2
	50	97	97	2
	500	100	100	4
	1	50	13	4
	43	51	38	1
All-Defs	10	70	21	4
	750	750	750	3
	125	125	83	2
	7	5	7	2
	70	53	53	2
	563	500	432	1
	340	45	45	4
	-50	3	-19	4

All-Uses	90	90	140	2
	140	200	140	2
	847	450	450	2
	1500	600	600	4
	13	38	402	4
	42	41	50	1
	322	322	1000	4
	420	5000	420	4
	-402	-4500	-5000	4
Prime paths	-1	5	10	4
	10	70	21	4
	10	15	21	4
	50	50	50	3
	95	95	40	2
	132	69	132	2
	738	500	500	2
	30	30	1000	4
	299	10000	299	4
	738	50	50	4
Predicate	0	0	0	4
	1	1	1	3
	1	2	3	4
	2	2	3	2
	2	3	2	2
	3	2	2	2
	1	2	2	2

	2	1	2	2
	2	3	4	1
	2	2	4	4
	2	4	2	4
	4	2	2	4
Clause	0	0	0	4
	1	1	1	3
	1	2	3	4
	2	3	6	4
	6	2	3	4
	2	6	3	4
	2	2	3	2
	2	3	2	2
	3	2	2	2
	2	3	4	1
	2	2	5	4
	2	5	2	4
	1	2	1	4
	5	2	2	4
CACC	0	1	1	4
	1	1	1	3
	1	0	1	4
	1	1	0	4
	1	2	2	2
	2	1	2	2
	1	2	3	4

	2	3	6	4
	2	3	4	1
	6	2	3	4
	2	6	3	4
	2	2	3	2
	2	3	3	2
	2	2	5	4
	2	3	2	2
	2	5	2	4
	3	2	2	2
	1	2	2	2
	5	2	2	4
RACC	0	1	1	4
	1	1	1	3
	1	0	1	4
	1	1	0	4
	1	2	2	2
	2	1	2	2
	1	2	3	4
	2	3	6	4
	2	3	4	1
	6	2	3	4
	2	6	3	4
	2	2	3	2
	2	3	3	2
	2	2	5	4

	2	3	2	2
	2	5	2	4
	3	2	2	2
	1	2	2	2
	5	2	2	4
GICC	0	0	0	4
	1	1	0	4
	0	1	1	4
	1	1	1	3
	1	2	2	2
	2	2	3	2
	1	2	3	4
	1	10	5	4
	5	1	10	4
	2	3	4	1
	3	2	1	4
	2	2	4	4
	2	3	3	2
	2	4	2	4
	4	2	2	4
	2	3	2	2
RICC	0	0	0	4
	1	0	0	4
	0	1	0	4
	0	0	1	4
	1	1	1	3

	1	2	2	2
	2	2	3	2
	1	2	3	4
	1	10	5	4
	5	1	10	4
	2	3	4	1
	3	2	1	4
	2	2	4	4
	2	3	3	2
	2	4	2	4
	4	2	2	4
	2	3	2	2

Table 8 – Test cases for TriTyp

G. Test cases for TestPat

	subject	pattern	output
Node	aab	aaa	-1
Edge	aa	aa	0
	aa	ab	-1
	a	b	-1
All-Defs	aa	aaa	-1
	aa	ab	-1
	aa	ba	-1
	aaa	bb	-1
	aa	a	0
	aba	aa	-1
	aaa	aab	-1
All-Use	aa	aaa	-1
	aa	ab	-1
	aa	ba	-1
	aaa	bb	-1
	aa	a	0
	aba	aa	-1
	aaa	aab	-1
	aa	aa	0
	ab	b	1
	aba	bb	-1
	abb	bb	1
	aaa	aaa	0
Prime	aaa	ab	-1

	aaa	bb	-1
	aaaa	aaaa	0
	a	aa	-1
	aaa	aab	-1
	aa	aa	0
	aab	ab	1
	ab	b	1
	aaba	aaa	-1
	aa	a	0
Predicate	aa	aa	0
	ab	aa	-1
	a	aa	-1
	aa	ba	-1
	aa	a	0
Clause	a	aa	-1
	aa	aa	0
	ab	aa	-1
	aa	ba	-1
	aa	a	0
CACC	a	aa	-1
	aa	a	0
	aa	aa	0
	aa	ba	-1
	ab	aa	-1
RACC	a	aa	-1
	aa	a	0

	aa	aa	0
	aa	ba	-1
	ab	aa	-1
GICC	a	aa	-1
	aa	a	0
	aa	aa	0
	aa	ba	-1
	ab	aa	-1

Table 9 - Test cases for TestPat