ON SOFTWARE TESTING AND SUBSUMING MUTANTS
An empirical study

Master Degree Project in Computer Science
Advanced level 15 credits
Spring term 2014

András Márki

Supervisor: Birgitta Lindström
Examiner: Gunnar Mathiason

Course: DV736A
WECOA13: Web Computing Magister
Summary

Mutation testing is a powerful, but resource intense technique for asserting software quality. This report investigates two claims about one of the mutation operators on procedural logic, the relation operator replacement (ROR). The constrained ROR mutant operator is a type of constrained mutation, which targets to lower the number of mutants as a “do smarter” approach, making mutation testing more suitable for industrial use. The findings in the report shows that the hypothesis on subsumption is rejected if mutants are to be detected on function return values. The second hypothesis stating that a test case can only detect a single top-level mutant in a subsumption graph is also rejected. The report presents a comprehensive overview on the domain of mutation testing, displays examples of the masking behaviour previously not described in the field of mutation testing, and discusses the importance of the granularity where the mutants should be detected under execution. The contribution is based on literature survey and experiment. The empirical findings as well as the implications are discussed in this master dissertation.

Keywords: Software Testing, Mutation Testing, Mutant Subsumption, Relation Operator Replacement, ROR, Empirical Study, Strong Mutation, Weak Mutation
# Table of contents

1  Introduction......................................................................................................................... 1
2  Background.......................................................................................................................... 2
2.1 Testing and quality in general............................................................................................. 2
2.2 Mutation testing – an overview.......................................................................................... 4
2.3 Mutation operators ............................................................................................................ 9
2.4 Weak and strong mutation ............................................................................................... 10
2.5 Relation to other testing methods ................................................................................... 11
2.6 Making mutation more suitable for general testing ....................................................... 13
3  Problem .............................................................................................................................. 14
4  Method and Approach........................................................................................................ 18
4.1 Methodology ..................................................................................................................... 18
4.2 Objective 1: Identify the relations for the two claims .................................................... 19
4.3 Objective 2: Experimental design ................................................................................... 21
4.4 Objective 3: Implementation and data gathering ......................................................... 21
4.5 Objective 4: Data analysis .............................................................................................. 22
5  Related work ....................................................................................................................... 24
6  Results................................................................................................................................ 25
6.1 Objective 1: The relations important for the claims .................................................. 25
6.2 Objective 2: Investigation on the experimental design .................................................. 27
6.2.1 The programming language used ............................................................................. 28
6.2.2 The data gathering process ....................................................................................... 28
6.2.3 The test programs ....................................................................................................... 31
6.2.4 Test cases................................................................................................................... 33
6.3 Objective 3: Experimental setup and data gathering ................................................... 34
6.4 Objective 4: Data analysis .............................................................................................. 37
6.4.1 First hypothesis: The subsumption graph ................................................................ 37
6.4.2 Second hypothesis: The top level mutants ............................................................... 40
7  Conclusion............................................................................................................................. 45
8  Discussion............................................................................................................................ 48
8.1 Achievement of goals ...................................................................................................... 48
8.2 Validity and limitations of the study ............................................................................. 49
8.3 Discussion of ethics ......................................................................................................... 52
8.4 Contributions .................................................................................................................. 52
8.5 Future research ............................................................................................................... 53
1 Introduction

Mutation testing is considered as a very strong but costly and tool dependent testing technique. It is a good way to ensure quality as it uses large number of test requirements. On the other hand it also needs heavy automation. It is considered costly because of the high number of requirements it creates compared to other testing techniques and it is tool dependent for the same reason. There is a strong effort to lower the cost of mutation testing by reducing the number of mutants generated, thus making it more appealing to the industry as well as making it more useful in general.

Kaminski, Ammann & Offutt (2011) present two theories on how to make mutation testing cheaper without losing effectiveness. This report presents an empirical evaluation on the two theories and investigates if they are valid or not.

The second chapter presents a background to the problem and describes the theories and concepts necessary to understand the rest of the document, the third chapter presents the problem alongside with the aim of this report, while the fourth chapter describes the method and the approach as well as presents the related work. The fifth chapter presents the results of this work, while the sixth draws conclusions on it. Chapter eight presents a discussion on the results alongside with its limitations and the contributions.
2 Background

2.1 Testing and quality in general

Software surrounds people in their everyday lives. People are dependent on software devices: Cell phones, computers, cars, washing machines and many more. In case the software fails, the results can vary from minor nuisances like a crashing application on a phone to disasters like a crashing vehicle. Software testing is an important tool to assess software reliability, thus quality (Morell, 1990).

Using software has risks, as software can fail. Testing is a way to minimize the risk of using given software. Testing is a tool which enables software developers to measure the system against given criteria and to get a picture on the quality of the system. “If you don’t measure, then you’re left with only one reason to believe you are still in control: hysterical optimism” (DeMarco, 1986, pp. 6), something that should be avoided in case some quality is needed to be known.

Testing is used to create predictable quality (Juristo, Moreno & Vegas, 2004). As Demillo, Guindi, McCracken, Offutt & King (1988) put it, "Software testing attempts to provide a partial answer to the following question: If a program is correct on a finite number of test cases, is it correct in general?". This question is an expression of the fact that no general answer can be given and the uncertainty about whether quality is achieved, even when using a given form of testing. It is possible that all the finite number of tests evaluate the same functionality, leaving much to wonder about the general correctness of the program. A large number of tests do not necessarily mean quality. Analysis techniques determine which test cases should be selected so that when running the tests, the examination of the tested system can be done efficiently (Juristo, Moreno & Vegas, 2004). Testing is also a costly task, as it may consume 50% of the development costs, which number can be higher with safety critical software (Harrold, 2000).

Testing is performed on software by executing test cases on programs. A test suite (sometimes denoted test set) contains a number of test cases. A test requirement on a given piece of software is satisfied when it is covered (or satisfied) by a test case and a coverage criterion is the rule(s) describing the set of test requirements that a test suite must fulfill (Ammann & Offutt, 2008, pp. 17). A test case is useful, if it can reveal a defect in the program. In case no failures are found, then either the program is free of faults or the test suite is inadequate. A reliable test suite would detect whenever the program is incorrect, thus if the program would pass the reliable test suite, then the program would be free of defects (Howden, 1976).

Such a reliable test suite could however not be constructed and it cannot be ensured in the general case that no incorrect program passes the test suite (Morell, 1990; Budd, DeMillo, Lipton & Sayward, 1980 and Howden, 1976). There are two mathematically undecidable problems which are the reason why there is no general solution. Firstly, in some cases the only way to know the correct results to test the program against is to build the correct program, which is called the Oracle problem (DeMillo, Guindi, McCracken, Offutt & King, 1988). The second problem has to do with generating, or even recognizing the reliable test suite, which can be traced back to the halting problem (Offutt, Jin & Pan, 1999; Morell, 1990;
Budd & Angluin, 1980). In practice however, much can be approximated (Offutt, 2011; Morell, 1990).

The Reachability, Infection and Propagation (RIP) model describes how much effect the various defects can have on the program execution (Yao, Harman & Jia, 2014 and Ammann & Offutt, 2008, pp. 12). Reachability denotes if a given static fault, which is a syntactical difference from the “right” program, can be reached by executing a given test case. A fault is reachable for a test case if the instruction containing the fault is executed by the test case, but it does not mean that the fault will have an effect. If no possible test case can reach a given fault then it is unreachable. Infection requires that the fault is executed and the execution results in a state difference. Infection happens if execution of a fault results in an internal error state and the defect is denoted as an error. In case the infection propagates to the interface so that the software displays different behaviour from what is expected externally, then the software defect is classified as failure. Reachability is necessary, but not sufficient to infection, while infection always means that the reachability property is fulfilled. Infection is necessary, but not sufficient for a failure. Having a failure on the other hand always means that there was infection resulting in a state difference in the end. Therefore, the only defects, which can be detected by looking at the results, are the failures. The term software defect is sometimes used to denote problems in the code without specifying if it is a fault, error or failure.

Faults which does not infect cannot be detected with executing tests even when the inner states of the program can be observed. It can be noted that the RIP model is based on previous models which describe the fault types in a similar way, but use slightly different notion (Offutt & Lee, 1997; Morell & DeMillo, Guindi, McCracken, Offutt & King, 1988), and some other research which traces defects back to specifications (Morell, 1990; Howden, 1982 and Hamlet, 1977).

Figure 1 alongside with Figure 2 displays a simple example program to illustrate the different types of software defects as well the RIP model. The first program is defect free, as it returns the value true in case the number is bigger than zero and false otherwise. The second program is a modified variant of the first variant, with a defect at the sixth row. The output is set to an incorrect value. Consider the following test cases: TestCase1 with executing isPositive(5) and TestCase2 executing with isPositive(-5). TestCase1 returns true in case of the original program and false in case of the second variant. The location of the fault is reached, where there is an infection resulting in an error state (output becomes false instead of true resulting a difference in the inner state) and there is propagation, where the error becomes failure that can be detected by looking at the results given by the two functions. However, for isPositive(-5), even though the static fault is present, its position is never reached, so there is no error, nor failure, because the tests result in the value false for both program variants.
2.2 Mutation testing – an overview

Mutation testing uses the idea to insert faults automatically into programs, building test suite(s) to detect them and using the test cases to decide whether the program is correct. The seeded faults are intended to be similar to the ones developers make when creating software. In case the test suite detects the seeded faults, then the real defects created by programmers should also be detected by these test cases. A program variant with a seeded fault is called a mutant. In case the test cases cannot detect any faults in the program, it can then be assumed that it contains none of the seeded faults. Mutation testing is a fault based technique, which has been applied to software (both procedural and component based), formal specifications, automata, security policies and network policies among others (Delamaro, Chaim, Vincenzi, Jino, Maldonado, 2011; Jia & Harman, 2011; Offutt, 2011; Offutt & Lee, 1994; Offutt & Lee, 1991; Howden, 1982).

The mutants are generated by mutation operators which are seeding faults into the system by following the rule that defines the mutation change, which is done in a systematic and repeatable way so that the program can be then effectively tested (Derezinska, 2006; Ma, Kwon & Offutt, 2002 and Alexander & Offutt, 2000). Faults are used as indicators to assert if a test can differentiate them from the original program (Budd, 1981). The operators can be different for different programming languages and depending on the used paradigms, like

```java
public int isPositive(int input) {
    boolean output;
    output = false;

    if (input > 0) {
        output = true;
    }

    return output;
}
```

Figure 1 Example program returning a true value if the input integer is greater than zero

```java
public int isPositive(int input) {
    boolean output;
    output = false;

    if (input > 0) {
        output = false;
    }

    return output;
}
```

Figure 2 An example on software with defect, as line six is set to an incorrect value
procedural or object oriented, different mutation operators may be needed. These theoretical differences are also supported by empirical studies (Kurtz, Ammann, Delamaro, Offutt & Deng, 2014; Ammann & Offutt, 2008, pp. 182-185; Derezinska, 2006; Kim, Clark & McDermid, 2000; Offutt, Pan, Tewary & Zhang, 1996 and Wong, Maldonado, Delamaro & Mathur, 1994). The semantic impact of the mutants can also vary (Yao, Harman & Jia, 2014; Ma, Kwon & Offutt, 2002 and Offutt, Payne & Voas, 1995).

The idea behind seeding faults is based on two main theories. The first one is the competent programmer theory: Programmers make mistakes, but they write programs which are relatively close to the desired program (Andrews, Briand & Labiche, 2005 and Budd, Lipton, Sayward & DeMillo, 1978). This small difference between the correct and the incorrect program is called a simple fault. Such simple faults can be corrected by a single source statement change (Offutt, 1992; DeMillo, Guindri, McCracken, Offutt & King, 1988 and DeMillo, Lipton & Sayward, 1978). The second theory is the coupling effect. There can be more complex problems with the program, which could not be addressed by making a single source statement change. The coupling effect describes that in such cases, the test suite that can differentiate the simple faults tend also to differentiate the complex faults, therefore mutation testing can be used to test larger software with more than one fault. The coupling effect is supported by both theoretical and empirical results (Offutt, 1992; Offutt, 1989 and DeMillo, Lipton & Sayward, 1978). Mutants with multiple statement changes are called complex or higher order mutants (HOM) and the higher order they are, the less likely they are not being detected after running the test suite on them (Ma, Offutt & Kwon, 2005; Wah, 2003; Offutt, 1992; Offutt, 1989; DeMillo, Guindri, McCracken, Offutt & King, 1988 and Budd, Lipton, Sayward & DeMillo, 1978). The practical reason to not generate higher order mutants is that their number (thus the requirements for the test suite) explode, even the 28 lines long FIND program which has 1029 first order mutants has 528906 second order ones, a difference that can be simply too large in practice (Harman, Jia & Langdon, 2010 and Offutt, 1989). Therefore, in most cases only first order mutants are used, even though it can be noted that higher order mutants have better realism as in most cases, real defects require multiple fault fixes (Harman, Jia & Langdon, 2010 and Eldh, Punnekkat, Hansson & Jönsson, 2007).

The process of mutation testing consists of several steps. The mutants are created from the original program using well-defined mutation operators and the test suite is prepared. The tests are then executed and the results gathered from the different mutants are compared to the result of the original program. The goal is to find test cases that display a different behaviour when used on the mutants compared to the original software (Demillo, Guindri, McCracken, Offutt & King, 1988 and DeMillo & Lipton, 1978).

A mutant can have different status depending on if a difference could be detected between them and the original program. Stillborn mutants contain syntactical problems which prevent them from being compiled and they are discarded directly before the tests are run (Offutt, Voas & Payne, 1996). A mutant which is distinguished from the original is considered to be detected (sometimes also denoted as killed) and if a mutant crashes under execution it is also considered detected (Demillo, Guindri, McCracken, Offutt & King, 1988). A mutant is alive if it is not yet detected by any test at that point and a mutant is trivial if it is detected by almost any tests. There are also equivalent mutants. Such equivalent mutants cannot be detected by any possible test case and are therefore considered as dead weight which does not add anything to the testing as they cannot even infect, but detecting them is computationally undecidable (Yao, Harman & Jia, 2014; Offutt & Untch, 2000; Hierons, Harman & Dancic,
There are indications that the number of second order equivalent mutants is lower (Papadakis & Malevris, 2010). Some mutants are considered stubborn and it cannot be determined automatically if they are equivalent or just really hard to detect, thus need manual analysis (Yao, Harman, Jia, 2014; Hierons, Harman & Danicic, 1999 and Offutt, Jin & Pan, 1999). The number of stubborn compared to equivalent mutants depends on the mutation operator (Yao, Harman, Jia, 2014).

Mutation analysis is a strategy that defines several test criteria (Offutt & Untch, 2000). Such criteria are satisfied in terms of coverage. 100% coverage means that the criterion is fully satisfied, which happens when all non-equivalent mutants are dead, in which cases the quality of tests should be good (Offutt, Pan, Tewary & Zhang, 1996). A test suite is considered mutation adequate relative to a mutation criterion when the test suite performs 100% in terms of mutation score (Delamaro, Maldonado & Mathur, 2001; Offutt & Untch, 2000; Kim, Clark & McDermid, 1999; Offutt, Lee, Rothermel, Untch & Zapf, 1996; Offutt, Rothermel, Untch & Zapf, 1996; DeMillo & Offutt, 1991 and DeMillo & Lipton, 1978):

\[
\text{mutation score} = \frac{\text{number of dead mutants}}{\text{number of all mutants} - \text{identified equivalent mutants}}
\]

The first mutants are relatively easy to detect, but the higher the mutation score gets, the harder it is to increase the score further. 40-50% are trivial and can be detected with random tests while a dynamic symbolic approach can get over 95% mutation score (Offutt, 2011; Frankl, Weiss & Hue, 1996; Offutt, Voas & Paas, 1996 and DeMillo & Lipton, 1978). A high mutation score is difficult to achieve as 4-10% of the mutants are usually equivalent and only 45-50% of them can be automatically identified using different methods. This number can however vary depending on the program structure (Madeyski, Orzeszyna, Torkar & Józala, 2014; Offutt & Pan, 1997; Offutt & Craft, 1996 and Budd, 1981). Depending on different program structures, empirical studies reported between 3% and 40% equivalent mutants, meaning that in some cases, manual analysis on them all can take a worker year, making manual analysis infeasible in some cases (Schuler & Zeller, 2012; Grün, Schulter & Zeller, 2009 and Schuler, Dallmeier & Zeller, 2009). It is observed that mutants that change control flow produce fewer equivalent mutants that those changing the data (Schuler & Zeller, 2012). Besides equivalent mutants and inefficient test cases, inherent fault tolerance can also make testing more difficult since the system may detect and correct an error state before the mutant is detected (Demillo, Guindi, McCracken, Offutt & King, 1988).

A test case can be described effective (or alternatively high quality) if it can detect one or more mutants, while test cases which do not detect a single mutant are considered ineffective (Offutt, 2011; Ma, Offutt & Kwon, 2005; Offutt, Lee, Rothermel, Untch & Zapf, 1996 and DeMillo, Guindi, McCracken, Offutt & King, 1988). Test cases should also be saved for later evaluation so that ineffective test cases can be removed, as they do not add anything to the quality, but take time to run (Offutt, 2011 and Offutt & Untch 2000).

Eventually in practice less than 100% mutation score can still be adequate for a test suite, as it can be really hard to decide if a mutant is equivalent or stubborn. As long as the number is higher than the required mutation score, the test suite can be mutation adequate even with lower than 100% mutation score. Even though the released program is not guaranteed to be correct, better adequacy score means that it is less likely that a fault remains and the program can be trusted (Frankl, Weiss & Hu, 1996 and Weyuker, Weiss & Hamlet, 1991).
The number of mutants can be high. As mutation adequacy requires that the mutants are detected by the test suite, detecting mutants translates to test requirements. There is one test requirement for every mutant, to detect (also to kill) it. The number of mutants is found to be $O(\text{References} \times \text{Variables})$ depending on the used mutation operator and is independent from the number of program lines, even though the later sometimes (incorrectly) appear in the literature (Offutt, Lee, Rothermel, Untch & Zapf, 1996; DeMillo, 1989 and Budd, DeMillo, Lipton & Sayward, 1980). Mutation testing is empirically found to be effective, however it needs automatization due to the high number of test requirements, and it may not be suitable in all cases because of economical reasons (Smith & Williams, 2009; Ma, Offutt & Kwon, 2005; Offutt, Pan, Tewary & Zhang, 1996; Wong, Maldonado, Delamaro & Mathur, 1994 and DeMillo, Guindi, McCracken, Offutt & King, 1988). An empirical study performed by DeMillo and Offutt (1991) is a good illustration on the high number of test requirements, as the 27 line long Fortran77 program Triangle had 970 mutants (with 107 being equivalent) and 420 test cases were needed to detect all but one of the mutants under a time of 10 minutes on a Sun 3/50 workstation.

Figure 3 Traditional Mutation Testing Process (Offutt & Untch, 2000)
Three factors illustrate well how much mutation testing has evolved over time. There were 390 articles published on the topic until 2010 with their numbers growing, thus the topic can be considered active and it draws more and more research attention (Jia & Harman, 2011). The number of programs has increased, with more and more real-world programs asserted with mutation testing, with program size growing over the years as the technique matured (Delamaro, Chaim, Vincenzi, Jino, Maldonado, 2011 and Jia & Harman, 2011). There is also evidence to show that the tools to support mutation testing are getting better and better to run T tests on P programs, with more and more parts of the being automated as it can be seen on Figure 3 and Figure 4, where solid boxes are automated and dashed boxes should be performed manually (Ammann & Offutt, 2008, pp. 181 and Offutt & Untch, 2000).

![Diagram](image)

**Figure 4** New Mutation Testing Process (Ammann & Offutt, 2008, pp. 181 and Offutt & Untch, 2000)
2.3 Mutation operators

Mutation operators are used to seed faults into the programs. There can be differences depending on which operators can be used on which programming languages/document types, and the tool support for the different operators may vary. The Fortran77 based Mothra system had 22 mutation operators (Jia & Harman, 2011; Offutt & Pan, 1997; Offutt & Voas, 1996; Offutt & Lee, 1994; DeMillo & Offutt, 1991; King & Offutt, 1991 and DeMillo, Guindi, McCracken, Offutt & King, 1988). Ammann & Offutt (2008, pp. 182-185) also adapted the original Fortran77 operators to Java which resulted in the following 11 operators:

- ABS – Absolute Value Insertion
- AOR – Arithmetic Operator Replacement
- ROR – Relational Operator Replacement
- COR – Conditional Operator Replacement
- SOR – Shift Operator Replacement
- LOR – Logical Operator Replacement
- ASR – Assignment Operator Replacement
- UOI – Unary Operator Insertion
- UOD – Unary Operator Deletion
- SVR – Scalar Variable Replacement
- BSR – Bomb Statement Replacement

The number of mutation operators is dependent on the syntax of the language they are applied to and the paradigm which is used. For example the number and name of mutation operators are different for C and Java and the object-oriented paradigm needs mutation operators different compared to procedural paradigm (Derezinska, 2006; Ma, Offutt & Kwon, 2005 and Ma, Kwon & Offutt, 2002). Below is the description on the ROR mutation operator based on the description by Ammann & Offutt (2008, pp. 182-185). A comprehensive description on all mutation operators is not discussed here, because the only relevant mutation operator for this study is the ROR operator.

ROR replaces each occurrence of the logical operators ("<"; ">"; "<="; ">="; "=="; "!=") with all the other logical operators and also true and false. For example if(x<y) becomes if(x>y); if(x<=y); if(x>=y); if(x==y); if(x!=y), if(true) and if(false). Table 1 displays the different mutants created for the respective operators.

**Table 1 Mutants created for the respective relation operators by ROR (Ammann & Offutt, 2008, pp. 183)**

<table>
<thead>
<tr>
<th>Original</th>
<th>Mutants</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>&gt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>&lt;=</td>
<td>&lt;</td>
</tr>
<tr>
<td>&gt;=</td>
<td>&lt;</td>
</tr>
<tr>
<td>==</td>
<td>&lt;</td>
</tr>
<tr>
<td>!=</td>
<td>&lt;</td>
</tr>
</tbody>
</table>
2.4 Weak and strong mutation

There are different notations depending on how a mutant can be detected in theory: *Strong mutation* and *weak mutation*. Strongly detecting a mutant requires a state difference at the end state. The result of the mutant program should be different from the result of the original program to consider the mutants strongly detected. Therefore propagation is required. Internal state difference, also denoted as infection is sufficient to weakly detect a mutant (Offutt & Ammann, 2008, pp. 178-179).

There is no significant difference when it comes to *strong mutation* in the literature. A different output is easy and straightforward to detect. In case it is not stated in case of a tool or a research paper which type of mutation is used, then in most cases strong detection can be assumed. The only exception is an article which differentiates strong state and strong output mutation; however this distinction is needed for a formal mathematical proof (Hierons, Harman & Danicic, 1999). It should be noted that no other article could be found making this distinction.

There are two things which should be taken into consideration with *weak mutation*: Observability and the theoretical background related to it. Observability is a problem, because program states can be difficult to observe. To detect everything, all variables in all components should be tracked along with the program counter, which can be very difficult, as for example object orientation means that some of the values and some internal state changes should be protected from the outside world. There is also the chain of failures, where a failure for a given test $T$ in a component which can be detected on the return value of that component results in an error state for the whole program, without necessarily propagating to a system failure. The test $T$ would be able to detect the mutant weakly if component level state difference can be detected, but not strongly as the final state would be the same (Yao, Harman & Jia, 2014; Ammann & Offutt, 2008, pp. 178-179; Hierons, Harman & Danicic, 1999 and DeMillo & Lipton, 1978).

There are different descriptions in the literature on what weak mutation means in practice, besides the infection based definition. Weak mutation can be done by executing components for unit testing instead of the whole program to reduce costs, which is a less strict criteria (hence the name), but it is not precisely described what a component is (Kim, Ma & Kwon, 2012; Offutt & Lee, 1994; Offutt, 1991; Demillo, Guindo, McCracken, Offutt & King, 1988 and Howden, 1982). As the definition on what can be considered is a bit unclear, ex-weak, st-weak, bb-weak/1 and bb-weak/n was also proposed, with different observation techniques: Empirical results displayed that depending on the implementation, weak mutation is better than strong mutation at finding defects which is in line with the fact that weak mutation does not need propagation (difference in the final state) to detect mutants (Kintis, Papadakis & Malevris, 2010 and Offutt & Lee, 1994). Flexible weak mutation is also proposed, where execution is detected at the anomalous state and compared with the original state multiple times (Mateo, Usaola & Offutt, 2010). In a way weak mutation is what was also discussed as *firm mutation* earlier, even though no such tool which stated that it can perform firm mutation was developed (Jia & Harman, 2011; Offutt & Lee, 1994 and Woodward & Halewood, 1988). Firm mutation can also be described as a generalization of strong and weak mutation, as *weak and strong mutation is basically the two extremes of a spectrum* (Hierons, Harman & Danicic, 1999 and Offutt & Lee, 1994). Strong output mutation is the easiest to observe, while a full trace on state changes even in the functions
which are executed when executing the whole program is the other extreme of weak mutation.

It should be noted that when it comes to tools, there can be differences on how the developers interpret weak detection criterion if it is supported. The referenced articles give indications on how the program defects were detected, as well as looking on the method chapters. The difference in effect between strong and weak mutation criteria to assert program quality to create test suites is however small in practice (Offutt & Ammann, 2008, pp. 178).

### 2.5 Relation to other testing methods

In order to define the relation between different testing criteria, the term *subsumption* can be used. It can be said that the criterion $C_1$ subsumes the criterion $C_2$, if any test suite that satisfies $C_1$ also is guaranteed to satisfy $C_2$ (Frankl & Weyuker, 1988). This relation is denoted in this report as $C_1 \rightarrow C_2$. Subsumption is sometimes denoted as *inclusion*, but the two describe the same relation (Offutt, Pan, Tewary & Zhang, 1996; Frankl & Weyuker, 1993; Weiss, 1989 and Clarke, Pogdurski, Richardson & Zeil, 1985). If there is no subsumption relation between $C_1$ and $C_2$, none of them subsumes the other and $C_1$ and $C_2$ are *incomparable* (Weyuker, Weiss & Hamlet, 1991). A subsumption relation also means that $C_1$ is likely more difficult to satisfy than $C_2$ (Mathur & Wong, 1994). Subsumption is absolute in terms. In case a criterion $C_1$ subsumes criterion $C_2$, then a test suite $T$ satisfying $C_1$ will always satisfy $C_2$.

Another type of subsumption is defined for the mutation operator ROR where a mutant $m_1$ is said to subsume another muther $m_2$ if any test that detects $m_1$ is guaranteed to detect $m_2$. This relation is denoted in this report as $m_1 \rightarrow m_2$.

For example a mutation operator ROR creates seven mutants for the function program snippet:

```java
int less(){return(a<b);}
```

Two of them are:

- `mutant1 less(){return(a<=b);}`
- `mutant2 less(){return(a==b);}`

In this case, if there is a subsumption relation $m_1 \rightarrow m_2$, then if a test $T$ can detect $m_1$, it is guaranteed to detect $m_2$.

To distinguish different logic based criteria, the following terms are important (Ammann & Offutt, 2008, pp. 104-105 and Offutt & Voas, 1996):

- A *decision* may contain boolean variables, non-boolean variables compared with relational operators {“>”, “<”, “==”, “>=”, “<=”, “!”} and function calls. A decision affects the control flow.
- A *condition* is a decision which does not contain any logical operators. Conditions can either evaluate to FALSE or to TRUE in value, and the negation operator NOT can be applied to them.

**Coverage or logic based criteria** are the following (Myers, 2004, pp. 36-40 and Offutt & Voas, 1996):

- *Statement coverage [SC]*: Every statement must be executed at least once in a given program.
- *Decision coverage [DC]*: (also known as branch testing / all-edges): Every decision should be evaluated to both true and to false at least once.
• **Condition coverage [CC]**: Each condition in each decision should evaluate to both true and false at least once.

• **Decision / condition coverage [DCC]**: All combinations of each decision and each condition should evaluate to both true and false at least once.

• **Modified Condition / Decision Coverage [MC/DC]**: Same as DCC with the more strict criteria. Each condition should independently affect the outcome of its respectable decision. Same as active clause coverage (Kaminski, Ammann & Offutt, 2011).

• **Multiple-Condition Coverage [MCC]**: Also known as extended branch coverage: All possible combination of the condition outcomes on all decisions outcomes should be covered.

Mutation based criteria offers higher effectiveness as coverage based criteria for a higher testing cost, even the industry standard MC/DC, can be strengthened further by using the ROR operator resulting in the Relation Operation Replacement Global (RORG) adequate test suite (Kaminski, Ammann & Offutt, 2011 and Offutt, Pan, Tewary & Zhang, 1996).

Criteria can also be defined based on graph coverage. Some of these include (Ammann & Offutt, 2008, pp. 35 & 48):

• **All-Uses Coverage (AUC)**: The test suite contains at least one path in the graph between the definition of a variable and every place it was used.

• **Edge-Pair Coverage (EPC)**: The tests are required to cover each reachable path P, which paths have a length of up to two.

• **Prime Path Coverage (PPC)**: The tests are required to contain every P path in a graph which P paths have the attributes that they do not contain any nodes twice, nor are they a subpath of any path which does not contain any node twice.

An empirical comparison on some smaller programs found that mutation testing was the most effective compared to Edge-Pair, All-Uses and Prime Path, as even though it needed more tests to cover the larger amount of test requirements, it could find the most faults (Li, Praphamontripong & Offutt, 2009). Table 2 shows a comparison to Edge-Pair, All-Uses and Prime Path criteria in terms of number of tests, test requirements, feasible test requirements, number of faults found and the costs versus benefits ratio (Li, Praphamontripong & Offutt, 2009). The number of test requirements is the highest for mutation testing; however the number of tests to cover all test requirements is actually the lowest (DeMillo & Offutt, 1991 and DeMillo, Guindi, McCracken, Offutt & King, 1988). Hence, mutation was more cost-effective than any other techniques in this study.

**Table 2** Comparison between Edge-Pair, All-Uses, Prime Path and Mutation (Li, Praphamontripong & Offutt, 2009)

<table>
<thead>
<tr>
<th></th>
<th>Number of tests</th>
<th>Number of test requirements</th>
<th>Number of feasible test requirements</th>
<th>Number of faults found</th>
<th>Costs versus benefits ratio (tests/faults)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edge-Pair</td>
<td>348</td>
<td>684</td>
<td>677</td>
<td>54</td>
<td>6.3</td>
</tr>
<tr>
<td>All-Uses</td>
<td>364</td>
<td>453</td>
<td>391</td>
<td>53</td>
<td>6.7</td>
</tr>
<tr>
<td>Prime Path</td>
<td>429</td>
<td>849</td>
<td>674</td>
<td>56</td>
<td>7.8</td>
</tr>
<tr>
<td>Mutation</td>
<td>279</td>
<td>2919</td>
<td>2357</td>
<td>75</td>
<td>3.6</td>
</tr>
</tbody>
</table>
2.6 Making mutation more suitable for general testing

There are three approaches in the literature on how the computational costs can be lowered, by either reducing the computational tasks or by reducing the burdensome tasks associated with mutation that humans need to do like running test cases on code or analysing equivalent mutants by hand (Offutt & Untch, 2000):

- **“Do fewer”:** Only a selection of mutants is created either with a subset of mutation operators, or only a randomly selected set of mutants are chosen of the set of all mutants, which results in faster execution time as the number of mutants (and therefore requirements) is smaller (Jia & Harman, 2011). Selective mutation (or alternatively constrained mutation) uses only a subset of the mutation operators. It is shown that 60% reduction on the number of mutants can still give a mutation score with 99% using the 6-selective technique and it is also shown that five mutation operators (ABS, AOR, LCR, ROR and UOI) are sufficient for both extended branch coverage and 99% mutation coverage (Offutt, Lee, Rothermel, Untch & Zapf, 1996; Wong, Maldonado, Delamaro & Mathur, 1994; Offutt, Rothermel & Zapf, 1993 and Mathur, 1991). Mutation sampling is a technique which uses only a given percent of all the created mutants, which selection can be done randomly or using a Bayes sequential procedure to perform tests to determine the probability that the program is sufficiently tested (Jia & Harman, 2011; Offutt & Untch, 2000; Mathur & Wong, 1994; King & Offutt, 1991; Sahinoglu & Spafford, 1990; DeMillo, Guindi, McCracken, Offutt & King, 1988; Offutt & King, 1988; Acree, 1980 and Budd, 1980). Higher Order Mutation in combination with strong subsumption on lesser order mutants is considered a “do fewer” approach as it creates less test requirements and less tests, but this approach still needs empirical validation to be considered useful (Kintis, Papadakis & Malevris, 2014; Madeyski, Orzeszyna, Torkar & Józala, 2014; Harman, Jia & Langdon, 2010 and Papadakis & Malevris, 2010). There is a research path which considers using higher order mutants to generate minimal set of mutants to assert a set of tests, which can be regarded as a combination of selective and higher order mutation (Ammann, Delamaro & Offutt, 2014). Genetic algorithms are proposed to reduce the number of mutants (de Oliveira, Camilo-Junior & Vincenzi, 2013 and Adamopoulos, Harman & Hierons, 2004).

- **“Do smarter”:** Weak mutation as a tactic for unit testing is considered as a do smarter technique, as well as solutions created on different architectures (like vector processors, SIMD and MIMD machines) and parallelized solutions (Mateo & Usaola, 2012; Offutt & Untch, 2000 and Fleyshgakker & Weiss, 1994).

- **“Do faster”:** Depending on how much time the compilation takes compared to the execution time, two different solutions exist here. In case of Mutant Schema Generation (MSG) all mutants are compiled at once to a meta-mutant which is used for example in case of Javalanche and MuJava along with bytecode translation (Li, Praphamontripong, Offutt, 2009; Schuler & Zeller, 2009 and Untch, Offutt & Harrold, 1993). A second approach is where all mutants are individually created, linked and run, which is suitable when compilation times are smaller than execution times, something which is used by Proteum (Delamaro & Maldonado, 1996).
3 Problem

Studies have shown that from the original 22 Mothra mutation operators, a selection of five (ABS, AOR, LCR, ROR and UOI) is the best choice to perform effective mutation testing in a cheaper way (Offutt, Lee, Rothermel, Untch & Zapf, 1996 and Demillo, Guindi, McCracken, Offutt & King, 1988). Out of the five investigated operators, ROR turned out to be less effective than the others. However ROR is the only mutation operator in the selection of five which ensures branch coverage criteria (Offutt & Voas, 1996). Branch coverage is something that is considered crucial in testing. According to a proposition, ROR can be made cheaper without losing the ability to ensure branch coverage or becoming less effective, if the theory of subsumption is applied on it. The basic idea is to select three out of seven mutants since a test suite that detects these three is guaranteed to detect the other four.

A way to “do fewer” is proposed by Kaminski, Ammann & Offutt (2011) based on a hierarchical subsumption graphs used in conjunction with the ROR mutation operator. They made two claims in their paper “Better Predicate Testing” about how ROR (Relation Operator Replacement) predicates could be used to effectively lower the number of ROR mutants needed to be detected, without losing any coverage. Their argument is properly grounded, but there is no empirical study that supports their theory with practical results. This is important, as factors like program structure can have effects on the execution of the program and the actual achieved coverage.

The statements of Kaminski, Ammann & Offutt (2011) are:

- “For all six relation operators, we can immediately see that tests detect three of the ROR mutants are guaranteed to detect all seven ROR mutants”. According to Kaminski, Ammann & Offutt (2011), there are subsumption relations between the mutants created by the ROR mutation operator, with an example displayed on Figure 5. The graph states that in case a test case detects the mutant m1, then it should detect the mutant m2 if there is an edge m1 -> m2. There is a subsumption relation between the mutated operators for a given original operator; therefore a test suite that detects all three top level mutants should also detect the mutants subsumed by them. Therefore, according to Kaminski, Ammann & Offutt (2011), it should be enough to generate the three top-level mutants, as test detecting them should also detect the underlying mutants too. This statement is also presented in Kaminski, Ammann & Offutt (2013).

- “Also any test that detect a mutant at the top level of a hierarchy is guaranteed not to detect either of the other two mutants at the top level of the hierarchy”, Tests should be mutually exclusive on detecting the top-level mutants in the hierarchy of the proposed subsumption graphs. This would mean a lower bound on the number of test cases needed for a ROR adequate test suite.

To give an example on the first statement, consider that the original program has a “<” operator which is mutated using the ROR operator, displayed on Figure 5. The top level mutants in this case are “FALSE”, “<” and “!” in the hierarchy displayed in a form of the subsumption graph. According to the first claim, a test suite which detects the three top level mutants “FALSE”, “<” and “!” is guaranteed to detect all the seven mutants. For example, in this case a test which detects “FALSE” is also guaranteed to detect “==”, “>” and “>=”. To give an example on the second statement, consider a program containing the original
operator “<” which is mutated using ROR. In this case, a test which would detect “FALSE”
could not detect neither “<=”, nor “!” according to the second statement.

**Figure 5** ROR subsumption graph for the original operator “**less than**” (‘<’) with
top level mutants marked (Kaminski, Ammann & Offutt, 2011)

There are similar graphs for all relational operators. These graphs are displayed on Figure 5 -
Figure 10.

**Figure 6** ROR sumption graph for the original operator “**greater than**” (‘>’) with
top level mutants marked (Kaminski, Ammann & Offutt, 2011)

**Figure 7** ROR subsumption graph for the original operator “**less or equal to**”
(‘<='’) with top level mutants marked (Kaminski, Ammann & Offutt, 2011)
Figure 8 ROR subsumption graph for the original operator “greater or equal to” (‘\(\geq\)’) with top level mutants marked (Kaminski, Ammann & Offutt, 2011)

Figure 9 ROR subsumption graph for the original operator “equal to” (‘\(==\)’) with top level mutants marked (Kaminski, Ammann & Offutt, 2011)

Figure 10 ROR subsumption graph for the original operator “not equal to” (‘\(!=\)’) with top level mutants marked (Kaminski, Ammann & Offutt, 2011)
In case the proposition of Kaminski, Amman & Offutt (2011) holds, it would also mean that cheaper branch coverage can be achieved, as 57% less ROR mutants would be sufficient. Given that their theory holds, the ROR criteria would be cheaper, lowering the overall cost of mutation testing. The work of Kaminski, Ammann & Offutt (2011) has been embraced by the research community and there are ongoing research based on their theories. This means that an empirical evaluation is urgent.

The article presented by Kaminski, Ammann & Offutt (2011) does not consider if the subsumption is true for weak or for strong mutation, so it is assumed that the subsumption graph holds on the same level. Therefore it is assumed that in case the subsuming mutant is detected strongly, the subsumed mutant should be detected too. The proposition does not discuss either if the claims are on high order or on simple mutants. It is assumed that they hold when using mutants containing simple faults, i.e. mutants with one syntactical difference from the original program.

As the goal of this research is to investigate the theory based on the two claims, the research is considered positivist, therefore the hypothetico-deductive logic applies (Lee, 1991). The major premise for the first claim is that “the subsumption relation between the mutants hold”, with the minor premise “a test T1 detects a mutant which is subsumed by another mutant”. This result in the first hypothesis: “In case there is a relation m1->m2 in one of the subsumption graphs (Figure 5 - Figure 10) and there is a test case t that strongly detects m1, then t will also strongly detect m1”.

For the second claim, the major premise based on the hypothetico-deductive logic is that “if a test detects one of the mutants on the top level of the hierarchy, it cannot detect any of the other two mutants at the top level of the hierarchy”, while the minor premise is that “the actual test T2 detects one of the highest level mutants of the hierarchy for program P2”, then this results in the second hypothesis: “A test case t that detects a top level mutant cannot detect another top level mutant for the same instance of a relational operator that is subject for mutation (Figure 5 - Figure 10).

The aim of this dissertation is to investigate if the two hypotheses based on the claims of Kaminski, Ammann & Offutt (2011) are true in practice. The aim can be divided into four objectives:

1. Identify every relation regarding subsumption and mutual exclusion which should be examined
2. Design the study
3. Implementation and data collection
4. Analysis of data and result

After all the four objectives are completed and the results evaluated, the aim should be fulfilled.
4 Method and Approach

The aim for this report is to inspect for all the six relation operators that there are three ROR mutants generated for each operator which are the top level mutants and each of such mutants are guaranteed to subsume the other four mutants. Also, if a given test detects one of the top level mutants in the hierarchy, it would not detect the other two.

4.1 Methodology

This report focuses on the relation between the theory presented by Kaminski, Ammann & Offutt (2011) and its technical relation with practice, which research on the connection between hypothesis and practice is quite common in case of a positivist approach (Orilowski & Baroudi, 1991).

Procedures associated with this hypothesis are classified as a positivist approach, which positivist understanding should be based on the rules of formal and hypothetico-deductive logic, so that the resulting theory should satisfy the following requirements, as well as it should be falsifiable (Lee, 1991):

- **Logical consistency**: The quality of the research, alongside with the construct validity, require this to be true. Without logical consistency, this article cannot be considered the result of a proper research.
- **Relative explanatory level**: Running an appropriate number of tests on a suitable number of programs results in enough data to either give an empirical check with the given experimental criteria, or in case one or both of the hypothesis fails, from the data trace it should be possible to explain why the results are not in line. The data collected should be enough (statistically significant) to either accept or reject the two hypotheses. Accepting them can happen with a statistical significant amount of data showing the hypotheses hold, while rejecting them can happen when a counterexample can be found.
- **Survival**: For external validity, it is needed that the proofs should be generalizable outside the domain of this research.

Research quality factors for positivist research are defined as objectivity, reliability, internal- and external validity, which factors result in trustworthiness for the research science (Oates, 2006, pp. 292). Objectivity means that bias should be avoided. Bias can be introduced unknowingly too, for example if the researcher who builds a tool to assert something testing related as well as the tested program (Li, Praphamontripong & Offutt, 2009). For this reason only such programs will be tested which are written by a different person.

Reliability in this context means repeatability. For experiments, the variables should be documented. In case software should be built as proof-of-concept, making the code available for review can be a solution, so that others can perform the same result. For literature study, the keywords, the databases and possibly the other search settings should be documented, so that others can perform the same survey. Subjective measurements, like selecting articles for a literature review are less reliable than objective measurements giving the same outcome, like data gathered from a well controlled experiment (Wohlin, Runeson, Höst, Ohlsson, Regnell & Wesselén, 2012, pp. 106).
**Internal validity** describes how well the research is connected to reality, how well they match. In positivist research it translates to how accurate the findings are. Threats against internal validity are also threats against causality, because they can affect the experiment variables (Wohlin, Runeson, Höst, Ohlsson, Regnell & Wesselén, 2012, pp. 106). Therefore, it is important for internal validity that the measurements are showing connection to the variables described by the research. **External validity** on the other hand means that the results are generalizable and give survival for the results. **Conclusion validity** asserts if the conclusion is in line with the data gathered also the explanation is done right (Wohlin, Runeson, Höst, Ohlsson, Regnell & Wesselén, 2012, pp. 104). **Construct validity** means that the constructs are well defined before constructing the experiment (Wohlin, Runeson, Höst, Ohlsson, Regnell & Wesselén, 2012, pp. 108).

### 4.2 Objective 1: Identify the relations for the two claims

The relations should be identified with help of the survey based on the paper holding the original claims as well as looking at the related papers to look at the theory. Objectivity is reached by examining the papers holding related theories in the field of mutation testing relevant to the ROR subsumption graphs. Reliability is achieved by the following description on how the literature study was performed.

The **literature study** is an extension of the method presented by Jia & Harman (2011). They used the search criteria on the papers published between 1977 and 2009 on the databases of IEEE explore, ACM Portal, Springer Online Library, Wiley Inter Science and Elsevier Online Library:

- “mutation testing”
- “mutation analysis”
- “mutants + testing”
- “mutation operator + testing”
- “fault injection”
- “fault based testing”

The repository on the 390 papers Jia & Harman (2011) found is no longer available online as of August, 2014, but they reference a set of 264 articles in their article, which gives a starting point to this survey alongside with the results of a search on the same databases with the same keywords using the filter criteria 2010-present on the published articles, performed on the 23th July, 2014. Jia & Harman (2011) describe that they both asserted the title and the abstract with expressions. This can only be achieved on IEEE and ACM with the publicly available search function. The search of Springer does return a huge number of unrelated (mostly biology themed) papers so only the name is considered, but not the abstract. Wiley can search on both abstract and title, but it can only search on words with AND connections between them and not on phrases, resulting in a huge amount of unrelated data. For example, ‘mutation testing’ results in 930 hits but only 15 are related to the field. For this reason, for every search, ‘software testing verification’ is added to the “every field”, to limit results to that journal. Elsevier has the same problem as Wiley with not being able to search on expressions, thus the limitation “on computer science” was added and the search was only performed on the titles. Reliability is thus achieved as the process is well documented and could be re-enacted.
The resulting articles, together with the articles presented by Jia & Harman (2011), are considered as a starting set of literature. Then, the papers are examined first by their title and if it is not unrelated, then the abstract is examined. In case the abstract is relevant for the aim of this report, the introduction and the results are inspected, with the possibility of considering other parts. In case the article is considered relevant, the references are also examined in the similar process and are added to the set of literature, also snowballing is used (Jalali & Wohlin, 2012). This is done until the point no new previously unidentified, relevant paper can be found. The set of literature is saved and is either directly added to the thesis report or is added to a database file containing short or more detailed information about the given article. The later is mostly the case if the reference is used in several places in this report.

Using an already existing set of articles and relying on the methodic presented in the given article/report to make the literature study makes this study dependent on the quality of the article of Jia of Harman (2011). This is addressed by using the reference list of other comprehensive studies to control that relevant studies were found: Ammann & Offutt (2008) with 594 citations, Juristo, Moreno & Vegas (2004) with 211 citations, Offutt & Untch (2001) with 243 citations. Two smaller less referenced articles is also considered: Offutt (2011) with 14 references as it extends the findings of Offutt & Untch (2001) and Delamaro, Chaim, Vincenzi, Jino & Maldonado (2011) with 3 references, as it focuses on the Brazilian community. Citation numbers are as of 2014-08-23.

The citing articles were considered in some cases, but not in all cases. The reason was that some publications have more than 500 others citing them, so that checking everything citing the set would result in a huge number of articles which are mostly unrelated to the topic. It is assumed that the literature study done on five relevant major databases with six search phases each alongside with their references should find the reports relevant for this work. However to be sure, a check on citing publications was done on some core articles and in cases there are differences in the literature (like Kaminski, Ammann & Offutt, 2011 and generally on the topic of weak mutation). After the described process as the literature study is done, the results relevant to the objective should be presented and with that, the first objective should be achieved.

Other methods were considered, but discarded. Literature survey results in taxonomy, giving sufficient theoretical ground for a study. Interviews can give depth, however finding a sufficient amount of people with relevant knowledge for the study can be difficult, even though they are good to get deep knowledge. It can also be assumed that no company is using the exact technique which is needed to gather the necessary results. Questionnaires give width, however the problem to get the necessary amount of answers is present here too. Therefore, only literature survey is considered in the case of this research.

It can be noted that the method by Jia & Harman (2011) used search strings on number of databases, also the systematic literature review (SLR), while the extension on the references is considered as snowballing (Jalali & Wohlin, 2012). Additionally, in case of snowballing, references are also searched up based on which context they appeared, thus papers are not discarded just because their name is not relevant, which can be a problem with the technique in general (Jalali & Wohlin, 2012).

This part of the study should focus on the claimed subsumption graphs and subsumption in general when it comes to use the set of literature.
4.3 Objective 2: Experimental design

In order to fulfill this objective a survey should be done with the objective of what is needed to be gathered and how can it be done. Both claims proposed by Kaminski, Ammann & Offutt (2011) are based on results of the connections between tests performed on different ROR mutants on given programs. Therefore, the literature should be studied with regards to:

- How can these test results gathered? How can the ROR mutants be generated?
- Which programs are available to be used for mutation testing?
- How should the test suite be generated?

As Li, Praphamontripong & Offutt (2009) names that the number of mutants can be simply too many to do testing manually on them, some kind of software aid is needed.

The set of literature built for the first objective is examined to get answers to the questions. The articles which should be examined are the ones describing the factors that can affect the experimental design. The process of mutation in general and factors associated with it and the reports on the tools should be focused on. When it comes to tool descriptions and empirical research, the method should be taken into consideration as there are considerable differences in the literature when it comes to some terms (weak and strong mutation), which uncertainty can affect the conclusion validity of the examined works. As the process of creating the set of literature is well documented, reliability is true in this case too.

Only literature survey is used in this case too, as both interviews and questionnaires are unsuitable similarly to objective 1. The set of literature is a result of a combination of systematic literature review and snowballing, as the set is the same as in case of objective 1, even though the set of literature was used differently to gather the necessary information.

The objective is fulfilled when the whole set is examined so that the experimental setup can be constructed, also the questions on the literature are answered.

4.4 Objective 3: Implementation and data gathering

After objective 2 is fulfilled, the experimental setup can be built. This is where the experiment is performed using the variables and factors identified in connection to objective one and two. Experiment is suitable in this regard as the aim is to try the hypotheses in practice and evaluate the results (Oates, 2006, pp. 126). Experiment is suitable because it gives high degree of control over the variables and the whole setup (compared to other empirical methods like case study), but is dependent on external and internal validity to be considered as proper research (Oates, 2006, pp. 132-133). Design and creation would focus on the development of new IT products, which is not the case (Oates, 2006, pp. 109). A case study would mean less control over the variables and would focus more on a process in its natural setting, which makes it less suitable (Oates, 2006, pp. 141 & 150). Action research would result in even less control over the variables as a researcher loses its observer status because of the collaborative nature of the method, making it unsuitable to gather the needed results to gather the data to evaluate the two hypotheses (McKay & Marshall, 2001). Research using ethnography spans mostly over 18 months in length and is mostly used to further evaluate experimental results in a natural setting, therefore ethnography is unsuitable for this empirical evaluation (Oates, 2006, pp. 181 and Myers, 1999).
The objective is completed when all the experiments are performed and all the data is gathered by the means identified by the first two objectives. The data generated by the experiment should be enough to give relative explanatory level. Explanatory level means in this case that either such data pattern should be found which rejects the hypothesis, or the data should be sufficient to give statistical evidence that the hypothesis holds, which translates to significance in this case. In both cases, the resulting data should be large and appropriate so that the results survive. Therefore, the experiment should be built in a way that a sufficiently large amount of output data should be created to either accept or reject the hypotheses. The experimental setup should also be built in a way that the finding could be applied to other programs with different attributes. Logical consistency is reached by building the experimental setup according to the findings of the literature survey.

Objectivity is reached by using programs created by others for the experimental design whenever possible. Reliability is achieved by giving the necessary information so that others can replay the experiment to assert if they can get the same result. Internal validity is reached by controlling the experimental variables appropriately, as well as introducing checks to the experimental setup so it can be asserted if the data generated is right. This is done by having a trace on the data, as well as building the experimental setup in a way that manual check on the results is achievable if needed. External validity is achieved by building the setup in a way and generating enough data so that the results can be useful even outside the borders of this experiment. Construct validity is achieved by building the experimental setup according to the literature.

4.5 Objective 4: Data analysis

For objective three the gathered data via the experiment is compared against the two hypotheses to see if they hold true or not. The objective is accomplished when the comparison is made and it can be decided whether the two hypotheses hold or not. By fulfilling these criteria, the aim should also be answered. This answer should also discuss implications, so that the results of the research will have survival (Lee, 1991).

Falsifiability is true if there is a way that a given hypothesis can be rejected. Falsifiability is achieved here by existence. In case the collected result displays that there are results that contradicts the subsumption relations, then the first hypothesis is falsified, while a test case which can detect more than one top level mutant for a subsumption graph for a given original operator falsifies the second hypothesis. For example in case it is shown that there is one ROR mutant R1 subsumed by the mutant R2 (created by the same mutation operator for the same original operator) for the given P program which left alive by the test T so that the subsuming mutant R2 from P is detected (where R1 ! = R2), then the first hypothesis is falsified. In case it is shown one highest order ROR mutant R1 from the program P, which is detected by the test T so, that there is also a ROR mutant R2 which is also dead (where R1 ! = R2 so that the same mutation operator created R1 and R2 from the same original operator from the same P program), then the second hypothesis is also falsified. Thus, the criteria are fulfilled for the project; the positivist approach is therefore feasible.

In case no exception can be found which would reject a given hypothesis, then if the data is statistically significant, the given hypothesis can be accepted. For this, the experiment should provide a large enough data set, with enough instances of subsumption relations and top level mutants so that each of the six subsumption graphs are found valid enough times.
Analyzing the data in a way that it is in line with the existing literature is needed for logical consistency, and doing so results in conclusion validity. This way, all the necessary attributes of research quality is addressed when performing this study.
5 Related work

There are a couple of different tools made for mutation testing for different programming languages: Mothra and an early test system is created for Fortran77 (DeMillo, Guindi, McCracken, Offutt & King, 1988 and Budd, DeMillo, Lipton & Sayward, 1978), Bacteria, Javalanche, Jester, Judy, MAJOR, MuClipse, MuJava and PIT is built for Java (Just, 2014; Delahaye & du Bousquet, 2013; Ramler & Kaspar, 2012; Just, Schweiggert & Kapfhammer, 2011; Madeyski & Radyk, 2011; Mateo, Usaola & Offutt, 2010; Schuler & Zeller, 2009; Smith & Williams, 2009; Ma, Offutt & Kwon, 2005 and Moore, 2001), Proteum and Proteum/IM2.0 is made for C (Delamaro, Maldonado & Vincenzi, 2001 and Wong, Maldonado, Delamaro & Mathur, 1994) and CREAM can be used for C# (Derezinska & Szustek, 2009). In the sense of making the data gathering automated, the findings presented in connection with other mutation tools are related to this research. However, these articles are different as their purpose is to describe a tool which is suitable for mutation analysis. The purpose of this report is to assert two hypotheses, which means that this research may need different kind of tool to create different kind of data using a process which can differ from tools created for mutation analysis.

There are a number of studies on reducing the set necessary for mutation analysis. Random selection and constrained mutations was compared empirically (Wong & Mathur, 1995). There are some empirical reports using mutant sets created with the limited ROR operator based on the first claim, however these studies make the assumption that the first hypothesis holds and have therefore different focus compared to this report (Just, Kapfhammer & Schweiggert, 2012b and Just, Kapfhammer & Schweiggert, 2012a). The mutant subsumption graphs were empirically researched further using Java / MuJava and C / Proteum with the assumption that the first hypothesis holds, focusing on the minimal set and mutant domain, considering cost efficiency for mutation (Kurtz, Ammann, Delamaro, Offutt & Deng, 2014). Even though these articles are similar in a way that they perform empirical studies on reduced number of mutants, they are different as they assert problems which are different from the purpose of this report. To the knowledge of the author, no other study than this one was performed on the article presented by Kaminski, Ammann & Offutt (2011) to evaluate if the proposed subsumption graphs hold in practice, which can also be a result of the fact that such a study may needs tools different from the tools currently available for mutation testing.
6 Results

6.1 Objective 1: The relations important for the claims

The two claims are both based on subsumption graphs presented by Kaminski, Ammann and Offutt (2011). There is one subsumption graph for every relation operator where ROR applies, also for “<” (Figure 5), “>” (Figure 6), “<=” (Figure 7), “>=” (Figure 8), “==” (Figure 9) and “!” (Figure 10).

For the subsumption, the following description is used: If criterion C1 subsumes criterion C2, then a test suite that satisfies C1 also satisfies C2 (Offutt, Pan, Tewary & Zhang, 1996; Frankl & Weyuker, 1993; Weiss, 1989; Frankl & Weyuker, 1988 and Pogdurski, Richardson & Zeil, 1985). As detecting mutants is equivalent to completing test requirements, subsumption means that if a set detects the mutant M1, it will also detect M2 if M2 is subsumed by M1. In case there is no subsumption relation, the mutants are incomparable. Thus, a set detecting or leaving alive M1 can both detect and leave M2 alive too, as detecting them as test requirements is unrelated (Weyuker, Weiss & Hamlet, 1991).

The second claim is about the top level mutants. According to the claim one test can only detect one of them at a time.

The first operator is less than ("<"), which has the proposed subsumption graph displayed on Figure 5. “FALSE” subsumes “==”, > and indirectly (over the medium level mutants) “>=”, so if a test detects FALSE, it should detect the other three subsumed relation requirements too. Similarly if a test detects “<”, it detects “==”, “TRUE”, and indirectly “>=”. In case a test case detects “!”, the “>”, “TRUE” and indirectly “>=” should be detected too. In case a test detects “==”, > or “TRUE”, then “>=” should be detected too. The top level operators are “FALSE”, “<” and “!”, according to the second claim one test case can detect at most one of them.

For the operator greater then (">"), the claimed subsumption graph to be asserted is displayed on Figure 6. False subsumes “==”, “<” and indirectly “=”, so if it is detected by a given T test, the other requirements should be accomplished too. “>=” subsumes “==”, “TRUE” and indirectly “<”, thus if a test detects “>=”, it should detect the three subsumed mutants too. “!” subsumes “TRUE”, “<” and indirectly “=” . “==”, “<” and “TRUE” subsumes “=” thus if at least one of them is detected by a test case, “<” should be also marked as detected by the same test. For the second claim to be true, a test case can detect at most one of the top level operators, also “FALSE”, “>” or “!”.

For the operator less than or equal to (“<=”), the proposed subsumption graph which is to be asserted to validate the first claim is displayed on Figure 7. “TRUE” subsumes “=!”, “>=” and indirectly “>”. “<” subsumes “!=”, “FALSE” and “>”. “==” subsumes “!=”, “FALSE” and “>”. So a test that distinguishes “TRUE”, “<” respective “==” should distinguish the mutation operators subsumed by them too. The medium level operators “!=”, “>=” and “FALSE” all subsume “>” too. The top level operators are “TRUE”, “<” and “==”. A given test case can detect at most one of “TRUE”, “<” and “==” according to the second claim.

The subsumption graph for the operator greater or equal to (<=), described by the first claim, is displayed on Figure 8. “TRUE” subsumes “=!”, “<” and indirectly “=”. The operator “!=” subsumes “FALSE” and indirectly “<”. The operator “==” subsumes “<”, “FALSE” and
indirectly “<”. A test case that detects “TRUE”, “>” respective “==” should also detect the respective subsumed mutants. The medium level operators “!=”, “<=” and “FALSE” also subsume “<”. The second claim needs that from the top level operators “TRUE”, “>” and “==” a test case only detected at most one.

The subsumption graph for the operator equal to (==), according to the first claim, is displayed on Figure 9. FALSE subsume <, > and indirectly !=. <= subsume <, TRUE and indirectly !=. >= subsume >, TRUE and !=. A test case that detects FALSE, <= respective >= should also detect the mutants generated by the subsumed operators for the first claim to hold. The middle level operators <, > and TRUE subsume !=, therefore a test that detects at least one of the medium level mutants should also detect the mutant generated by the operator !=. For the second claim, it should be true for all individual test cases that a test case can only detect at most one from the mutants with the mutated operator FALSE, <= or >=, but no more.

The subsumption graph for the operator not equal to (!=), according to the first claim, is displayed on Figure 10. The results gathered should agree with the subsumption graph so that the claim holds. TRUE subsumes <=, >= and indirectly ==. < subsumes <=, FALSE and indirectly ==. > subsumes >=, FALSE and indirectly ==. It is required for the claim to be true that a test case that detects the TRUE, < or > mutants should also differentiate the requirements subsumes by them. A test that detects at least one of the mutants <=, >= or FALSE should also identify == as a mutant. For the second claim to hold it should be true for all test cases that an individual test case can at most detect one of the mutants TRUE, < or >, but not more.

In order to validate the claims, all six subsumption graphs should be asserted with the experimental setup, as all six original operators should be asserted. The graphs are grouped for the hypotheses and are asserted in groups. In case an exception would be found for one of the graphs, the claim is falsified. The truth table of the mutant subsumptions is displayed on Table 3. “OK” means that the pattern is in line with the hypothesis, while “Not OK” means that the subsumption is broken. Table 4 displays the pattern which should be checked by the experimental setup to assert the second hypothesis. In case a given test case does not detect any top level mutant or it only detects one, the pattern matches the hypothesis and the pattern is “OK”. However, in case a single test case can detect more than one top level mutant, the resulting pattern on them is not in line with the hypothesis. This pattern should be asserted on all the tests which are executed.

**Table 3** Truth table for the mutant subsumption

<table>
<thead>
<tr>
<th>Mutant in question alive</th>
<th>Mutant subsuming it alive</th>
<th>Mutant subsuming it detected</th>
<th>Mutant subsumed by it alive</th>
<th>Mutant subsumed by it detected</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>Not OK</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>Mutant in question detected</td>
<td>OK</td>
<td>OK</td>
<td>Not OK</td>
<td>OK</td>
</tr>
</tbody>
</table>
Table 4  Relation between the status of the three top level mutants m1, m2 and m3 after running a given test and the hypothesis

<table>
<thead>
<tr>
<th>Status of Top Level Mutants</th>
<th>Resulting pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>live</td>
<td>live</td>
</tr>
<tr>
<td>dead</td>
<td>live</td>
</tr>
<tr>
<td>live</td>
<td>dead</td>
</tr>
<tr>
<td>live</td>
<td>live</td>
</tr>
<tr>
<td>dead</td>
<td>dead</td>
</tr>
<tr>
<td>dead</td>
<td>live</td>
</tr>
<tr>
<td>live</td>
<td>dead</td>
</tr>
<tr>
<td>dead</td>
<td>dead</td>
</tr>
</tbody>
</table>

6.2 Objective 2: Investigation on the experimental design

According to the results of objective 1, all six subsumption graphs should be asserted with the experimental setup for both hypotheses. There are a number of variables which should be taken into consideration. Firstly, testing is performed on a given P program using T tests and comparing the results to the excepted results (Howden, 1976). The excepted results to compare the mutants are possibly not computable because of the Oracle problem (DeMillo, Guindi, McCracken, Offutt & King, 1988). This is however not a problem in this case, as the original claimss from Kaminski, Ammann & Offutt (2011) only concerns the relation between the different operators. As long as the results can be compared between the original program and the mutants, comparable results can be gathered.

The two claims are described on the six operators ROR concerns with and all six operators (“<”, “>”, “<=”, “>=”, “==” and “!”) should be checked to cover the requirements described in connection to objective 1.

This leaves that there should be a set of programs to test, with some test cases. These programs are therefore variables in the experiment. Multiple programs are needed to get more generalizable results and to cover all operators. The programs should also contain all six mutation operators ROR mutates for statistical reasons. In case all results are in line with the hypotheses and the hypotheses proposed according to the claims hold, a large enough number of data can provide statistic significance. The appropriate mutants should be created from the original program, so that they can be detected using the test suite. It also expected that the test suite should be re-executed a couple of times on the program, even in case the results are deterministic. The test should detect enough mutants so that the necessary data can be collected to evaluate the hypotheses.

A clear sign of the fact that mutation testing is not really suitable to be performed by hand is the amount of automatization introduced with the new mutation testing process, which is displayed on Figure 4 (Madeyski & Radyk, 2010; Ammann & Offutt, 2008, pp. 181; Ma, Offutt & Kwon, 2005; Offutt & Untch, 2000; Offutt, Pan Tewary & Zhang, 1996 and DeMillo, Guindi, McCracken, Offutt & King, 1988). Therefore, execution of the test should be automated. The check on the two hypotheses should also be automated. A tool capable of doing that automatisation is needed. The tools for creating and running the mutants, as well
as creating, proceeding and analyzing the data are needed to perform the study. In case no capable tools can be found, then such tools should be created.

6.2.1 The programming language used
There are differences between different programming languages when it comes to mutation analysis both in terms of mutant operators, test suites, coding style, minimal test suites and minimal mutant sets (Kurtz, Ammann Delamaro, Ofutt & Deng, 2014; Fraser & Zeller, 2010 and Wong, Maldonado, Delamaro & Mathur, 1994). For example, Proteum does not support ROR for C while some of the OO operators can not be applied to C# because of structural differences (Derezinska, 2007 and Derezinska, 2006).

External quality can be low when using variables which are not really useful for others: In this case Fortran77, Cobol and Ada can be deselected for this reason, as they are not really used anymore when it comes to mutation testing for research purposes (Wohlin, Runeson, Höst, Ohlsson, Regnell & Wesslén, 2012, pp. 110 and Jia & Harman, 2011). Recent articles based on the findings of Kaminski, Ammann & Offutt (2011) use Java and C, so for generalizable results one of them can be used (Kurtz, Ammann, Delamaro, Offutt & Deng, 2014 and Just, Kapfhammer & Schweiggert, 2012a). Taking into consideration the language differences and that Proteum, the most widespread mutation analysis tool for C does not support ROR, for direct comparison, Java was selected (Derezinska, 2007).

6.2.2 The data gathering process
A couple of tools exists for mutation analysis, like Mothra for Fortran77, Mujava, Muclipse, Javalanche and MAJOR for Java, Proteum for C and CREAM for C# based on a survey on tools before 2010 (Jia & Harman, 2011). A common point is the high level of automation, something which is displayed on Figure 3 and Figure 4. Survey articles show domain maturation over the years and optimization and automatization of the mutation analysis process got large attention (Jia & Harman, 2011; Offutt 2011; Ammann & Offutt, 2008 and Offutt & Untch, 2000). The new mutation testing process (Ammann & Offutt, 2008, pp. 181 and Offutt & Untch, 2000) is an example: Much is automated, as the different tasks cannot be done by hand.

The goal with mutation testing is to create a test suite which is strong enough in the sense of detecting enough mutants. With other words, the test suite adequacy is asserted by the use of a set of test requirements which are basically to detect each of the mutants. Because mutation testing is expensive, one of its optimizations is to mark mutants which are already detected, and not test them anymore because they are already covered by a test. Detecting them again would take computational power, but would not affect the mutation scores. In this process, uneffective tests, also the ones that do not detect mutants which are not already dead are removed as well, which makes the process faster too.

Here lays the difference between what is provided by the tools available for mutation testing and what this study needs. In this research, the programs chosen are considerably small and making an appropriate test suite can be done by hand as long as a mutation adequate test suite can be reached for every program. The number of equivalent mutants is low enough so that they can be analyzed by hand, even if it takes around 15 minutes for each (Schuler & Zeller, 2012). The generation of mutants can be done by a tool on the market. The tool used for mutant generation should be checked that it generates the right mutants, as it can affect the data generated by the experiment and thus the experimental results.
The major difference is however that in the case of this experiment, it is needed that all of the tests are run on every mutant. This is the only way to build the sufficient amount of experimental data which can be later analyzed, so that the two hypotheses could be asserted using the results. It is important to note that no mutants are discarded and the whole set of created ROR mutants are used for every program. This is necessary to investigate whether e.g. the subsumption relation holds in every case.

Figure 11 displays the process model created for this experiment based on the two models described by Offutt & Untch (2000). Boxes marked with dashed lines are resources which are used from external sources; otherwise the rest of the system is created for this experiment. For validating the first hypothesis, (all results on (all mutants on all tests)) on all programs should be checked. Every mutant is connected by subsumption to three other mutants, thus for every mutant, the relation should be asserted and checked, if it holds.

Figure 11 The process needed for the experiment
For the second hypothesis, at least the top level mutants should be listed by the program and afterwards checked, if only max one of them is actually detected by any given test case.

The results should then be displayed to see how many of the mutants are detected by which tests and also a general overall score can be seen to help decide if improvements to the test should be made or not.

In order to create the mutants, MuJava was selected. It can mutate Java, it has the ROR mutation operator, it is actively in development and it is actively in use by researchers (Kurtz, Ammann, Delamaro, Offutt & Deng, 2014). To make sure that the tool produces the right mutants, a minimal program was created only containing the necessary operators independent from each other, as well as MuClipse can used as a second check on one of the programs. In case there would be differences between the programs or between the expected mutants (which are known because of the simple program used for the dry run); Judy and Javalanche could also be used for mutant generation. JavaMut and Bacteria could not be downloaded, while Jester and Jumble are known to produce only a subset of mutants which would make them unsuitable for the study (Madeyski & Radyk, 2010). PIT does not have support for ROR.

MuJava only creates first order mutants. The original claims are not discussing if they should apply first or higher order mutants, therefore the validation of the claims should firstly be done on the first order mutants. Based on the simple fault hypothesis and the coupling effect, tests generated for first order mutants containing a single syntactical difference are able to detect more complex faults (Offutt, 1989 and DeMillo, Lipton & Sayward, 1978). There exists empirical research which supports this position, which research also advises that the use of first order mutants is sufficient (Offutt, 1992). As the number of higher order mutants can be much larger, the tool does not support the creation of higher order mutants, as well as there is evidence that tests created based on first order mutants can be regarded sufficient, this study only uses first order mutants when performing this experiment.

The program is built for strongly detecting mutants. There are three reasons behind this decision: The first one is that according to the literature, the difference between test suites based on strong and weak detection is marginal. The second one is the problem with observability. The inner state of a program can be hard to assert and comparing a large number of return values is easier than comparing states. The third problem is with the definition of weak detection. Strong and weak detection are extremes on a spectrum, there is difference if weak mutation testing is a tool for unit testing ("do faster") or a stronger criterion which does not require propagation for a detect, and practical considerations can differ too on how weak mutation should be performed, like ex-weak, st-weak, bb-weak/1, bb-weak/n or the component focused solutions (Jia & Harman, 2011; Mateo, Usaola & Offutt, 2010; Ammann & Offutt, 2008, pp. 178-179; Hierons, Harman & Danicic, 1999; Offutt & Lee, 1994; Offutt, 1991; Guindi, McCracken, Offutt & King, 1988; Woodward & Halewood, 1988; Howden, 1982 and DeMillo & Lipton, 1978). A practical example is the related study of René, Kaphammer & Schweiggert (2012) which is described to use weak mutation, however judged by the run times used by their strong and weak mutation analysis sets, they are using the component based, "do faster" weak mutation criteria. On the other hand, there is an agreement in the literature what strong mutation is. The program results are used as detection criteria for the automated tool when using strong criteria, as in practice, it should reflect the final program state when propagation is true (Kintis, Papadakis & Malevris, 2010; Hierons, Harman & Danicic, 1999 and Offutt & Lee, 1994).
6.2.3 The test programs

It is not needed for the hypothesis that the programs are correct and used by other researchers of other publications; however for external validity and generalizable results, this is desirable. The semantics affect the feasibility of the test requirements through the so called “feasible path problem”, thus the way the program logic flows is a variable (Offutt & Pan, 1997). The flow of the code (also the style of how the structure is built) is a factor alongside with other uses. The oracle problem does not affect this experiment, as mutation testing is based on detecting a difference, and for the subsumption graphs, it is only needed that this difference is detected according to the proposed hypotheses.

An empirical study showed that even though testability is seemingly unrelated, the number of occurrence of constructs, number of unique constructs, number of occurrence of operators and number of unique operators is affected by the programmer writing the code (Hayes & Offutt, 2009). This is in line with the statement that programmers should not test their own code as they can be blind on the defects they make (Myers, 2004, pp. 17).

Every original operator <, >, <=, >=, == and != needs to have enough instances in the set of chosen programs. Every operator should be present in at least one program, so that the ROR operator can create the appropriate mutants so that the hypothesis can be asserted.

The selected set of programs fulfilling the criteria is the following:

- **Cal** (Ammann & Offutt, 2008, pp. 132): Displays the number of days between two dates in a given year. See Appendix A - Cal for implementation.
- **CountNatural** (Ammann & Offutt, 2008, pp. 16): Counts the number of natural numbers (positive integers and 0) in an input array. Originally called for CountPositive, it is renamed because it counts zeros too. See Appendix B - CountNatural for implementation.
- **TestPat** (Ammann & Offutt, 2008, pp. 56): Decides if a pattern is present in a given string. See Appendix C - TestPat for implementation.
- **TriTyp** (Ammann & Offutt, 2008, pp. 121): Decides which type a given triangle is. See Appendix D - TriTyp for implementation.

Table 5 displays detailed information about the four programs, with the numbers and the alternative names based on the findings on Jia & Harman (2011). Cal, TestPat and TriTyp are widely used in research publications, while CountNatural is needed to cover all operators with the programs. In case of Cal and Triangle, there is evidence that they are still used as research examples despite their age (Kurtz, Ammann, Delamaro, Offutt & Deng, 2014 and Yao, Harman & Jia, 2014). Note that the size here displayed is the length of the original implementations and not the size of the Java variants besides CountNatural. There can be differences between different implementations of programs in different languages, and the number of mutants, number of equivalent mutants and number of tests needed for ROR coverage can vary between implementations. An example of a program that did not generate any ROR mutants was CheckIt, which was therefore discarded (Ammann & Offutt, 2008, pp. 130).
It was also considered that in case the mutants are created with external tools, it is important that all mutants are created, and they are made accordingly to achieve internal validity and thus, quality. Additionally, for the tools produced by the author, it was considered important that the results should be observable and the tool should be tested by a test program including all the relational operators. This way it can be checked that all the mutants are actually created. This test program has to be simple enough so that the results can be checked manually to verify the behaviour and detect any faults. For this reason the simple program \textbf{Relation} was written according to the requirements specified above. See Appendix E - Relation for the implementation. All the operators are present in Relation and they are present in separate functions so that they do not affect each other. The program Relation is also the bare minimum to test all functions, so that the programmers writing style don’t affect the results.

The functionality for every program used is contained in a single function. Therefore, strong output mutation means in this case that the \textit{return values are checked on method level} to detect if there is any difference. The observation is therefore finer grained in this experiment, than in most cases when strong mutation is used.

\begin{table}[h]
\begin{center}
\begin{tabular}{|l|l|l|l|l|}
\hline
\textbf{Alternative name} & \textbf{Size (in Loc)} & \textbf{First use} & \textbf{Number of uses in research articles (before 2011)} \\
\hline
Cal & Calendar/Days & 30 & 1988 & 15 \\
CountNatural & CountPositive & 14 & 2008 & 0 \\
TriTyp & Triangle & 30 & 1978 & 25 \\
TestPat & Pat & 20 & 1991 & 10 \\
\hline
\end{tabular}
\end{center}
\caption{Detailed information on the different programs}
\end{table}

\begin{table}[h]
\begin{center}
\begin{tabular}{|l|l|l|l|l|l|}
\hline
\textbf{Cal} & \textbf{CountNatural} & \textbf{TestPat} & \textbf{TriTyp} & \textbf{Relation} & \textbf{Summa} \\
\hline
Sum number of ROR mutants & 34(+1) & 12(+2) & 29(+6) & 119 & 42 & 236 \\
Number of mutants created from $<$ operators & 0 & 5 & 14 & 0 & 7 & 26 \\
Number of mutants created from $>$ operators & 0 & 0 & 0 & 28 & 7 & 35 \\
Number of mutants created from $\leq$ operators & 6 & 0 & 0 & 42 & 7 & 55 \\
Number of mutants created from $\geq$ operators & 0 & 7 & 0 & 0 & 7 & 14 \\
Number of mutants created from $==$ operators & 14 & 0 & 8 & 49 & 7 & 78 \\
Number of mutants created from $\neq$ operators & 14 & 0 & 7 & 0 & 7 & 28 \\
\hline
\end{tabular}
\end{center}
\caption{The number of ROR mutants created for the different programs.}
\end{table}
The distribution of the operators created for the different operators are a bit uneven, as it is displayed on Table 6, but this is somewhat expected as the different operators are not equally used in real software. On the other hand each operator is mutated at least in two of the programs so that the results cover all six proposed subsumption graph. It can be noted that seven mutants are created by applying the mutation operator to every original relation operator but there are three exceptions. The first one is **TestPat**, where the operator `==` in the row

```java
while (isPat == false & & iSub + patternLen - 1 < subjectLen) {
```

only gets a `!=` mutant, as all other mutants would be either stillborn or redundant as in the case of `isPat == true/false and true/false`, which is not generated by the tool. The second place where not all the seven mutants are created is **Cal**. The `<=` operator in the row

```java
for (int i = month1 + 1; i <= month2 - 1; i++) {
```

would be illegal if it would be replaced with a false. The third exception is **CountNatural**, where the row

```java
for (int i=0; i < x.length; i++)
```

results in stillborn mutants for both true and false. True results in a syntactically illegal statement, while false is syntactically incorrect in a for loop.

Stillborn mutants are discarded at this point, as they are not accepted by the compiler and therefore cannot be used as requirements.

### 6.2.4 Test cases

The number of tests performed on a given program is coupled to the program mutated. **Cal**, **CountNatural**, **TriTyp**, **TestPat** and **Relation** take different input data and have different functionality. Therefore they all need different test cases compared to the other programs. The test cases should be 100% mutation adequate so that no subsumption cases will be missed. The number of equivalent mutants is expected to be fewer than 10% (Budd, 1981). With the 236 non-stillborn mutants, it can be expected that there would be no more than 30 equivalent mutants and about the same amount of stubborn (Yao, Harman & Jia, 2014 and Offutt & Pan, 1997). Marking the mutants right is important, as incorrectly marking mutants equivalent results in higher mutation scores, which makes the test suites look better (Frankl, Weiss & Hu, 1996). The test suites are however not needed to be minimal. More tests mean in this case just more data to accept/reject the hypotheses. The tests are created manually without a supporting tool. Because the program flow is known and 100% mutation adequacy is a criterion, using hand crafted tests is appropriate. Randomly creating tests can be inefficient, and as the programs are small, it is expected that test cases can be created by hand without any difficulties (Frankl, Weiss & Hu, 1996). No test case was omitted because it is not in line with the expected results. The results shown here are the results of strong detection. These numbers are based on differences detected on the results if not stated otherwise.

Mutation testing results for Relation (Appendix H) displays the required test cases, as well as the results for the program Relation. As it is displayed, the mutation score is 100%, all the mutants were detected. Since this was the simplest program which was mostly added as an extra check on the external tools, this was expected.
Mutation testing results for Cal (Appendix F) displays the required test cases, as well as the results for the program Cal. The three mutants which could not be detected are equivalent to the original program; therefore, without them the mutation score should be 100%. Note that the lower score displayed in the appendix is a result of the fact that the program cannot detect the equivalents and manual change on the computed data is prohibited as it could result in tampering with the data, something which is unwanted from internal validity point of view. Appendix K contains a detailed explanation why they are equivalent.

Mutation testing results for CountNatural (Appendix G) displays the required test cases, as well as the results of the mutation analysis for the program CountNatural. There is one mutant which could not be killed, without it, the mutation score would be 100%. The analysis on the mutant is discussed in Appendix L.

Mutation testing results for TestPat (Appendix I) displays the required test cases, as well as the results for the program TestPat. Appendix M displays the analysis on the mutants alive after running the test suite: All mutants alive are equivalent, and the mutation score should be 100% without them. Note that the test case 3 is not needed for the minimal test suite and also the test case 1 or 2 could also be omitted (but not both).

Mutation testing results for TriTyp (Appendix J) displays the required test cases, as well as the results for the program TriTyp. Appendix N displays the analysis on the remaining mutants after running the test suite and all three are equivalents. With the equivalent mutants discarded, the mutation score for the test suite is 100%.

All test suites are 100% mutation adequate to strongly detect program mutants, which means that no feasible requirement is omitted by them and all the non-equivalent mutants are detected.

6.3 Objective 3: Experimental setup and data gathering

This part describes the experimental setup which was built according to the results of objective 1 and objective 2, to handle the input variables and to create the necessary data. Table 7 displays the number of mutants and their status. Some mutants are determined to be equivalent after manual analysis, but disregarding the equivalent ones, all mutants are detected by the test cases. The results are deterministic. 100% mutation score is achieved because all the syntactically correct and non-equivalent mutants are detected for every program.

A tool to collect data regarding the relations was created. As faulty results could tamper with the data, the most important factor is traceability. All results can thereby be validated by hand if needed. As observability is important, mostly String data types are used, along with the final modifier. The variables get their values on the creation of objects, so that the data cannot be modified afterwards. String works fine to control the flow of data in the program, however performance could be improved. On the other hand, observability is better this way. There are built-in debug settings in the program to trace data flow, and the tool is built in a way that if something fails, it should fail big.
Using the Java keyword “final” leads to a problem with the in-program mutation score, as there is no way letting the program know which mutants are equivalent. On the other hand, this is a design decision, as the data cannot be tampered with this way. There is also a displayed trace on how the data is built and how the hypotheses are validated. In case needed, the program displays where the original operator is (complete with type, column and row position), alongside with which operation it was mutated to, computed in real time based on the actual program. It is displayed if the mutant is killed or alive. For the first hypothesis, all the three other mutants connected via the subsumption are displayed, complete with the information if the mutant is killed or alive, alongside with the information if other mutant subsumes or is subsumed by the given mutant. The relation is asserted accordingly to the truth table displayed in Table 3.

Another design decision is to include all the mutants in the same project as the test program, alongside with an interface add-on so that the mutants could be placed into containers so that the test could be run efficiently on them, in a way that every mutant is loaded under the whole time of the execution. In a way, the mutant interface serves as a meta mutant, however this solution does not use bytecode like MuJava (Li, Praphamontripong, Offutt, 2009; Schuler & Zeller, 2009 and Untch, Offutt & Harrold, 1993). The obvious problem is that all of the mutants should be added to the project, which would not work with a bigger project or with a larger number of mutants. The modifications to the test programs were only the addition of the interface, as well as the comments and the spacing were slightly affected by the mutation tool. Note however that the main functions are not mutated by MuJava, nor by MuClipse, so those are omitted from the projects. Appendix O to Appendix S contains the modification to the programs to make them compatible with the tool.

The tool has an appropriate exception handling ability. In case one of the mutants crashes the tool remains unaffected, but notes that the mutant behaves differently from the original program (unless the original program crashes the same way). This design decision is similar to the behaviour of the Mothra system (Demillo, Guindi, McCracken, Offutt & King, 1988).

The tool has a length of about 3000 lines of code divided into 15 classes. For this reason, it is not included as an appendix.

For all the executions, a virtual machine using Windows 7 Professional complete with Eclipse and Netbeans was used. The VM was also dedicated to running the system, alongside with

<table>
<thead>
<tr>
<th></th>
<th>Number of mutants created</th>
<th>Stillborn</th>
<th>Equivalent</th>
<th>Detected</th>
<th>Mutation score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cal</td>
<td>35</td>
<td>1</td>
<td>3</td>
<td>31</td>
<td>100%</td>
</tr>
<tr>
<td>CountNatural</td>
<td>14</td>
<td>2</td>
<td>1</td>
<td>11</td>
<td>100%</td>
</tr>
<tr>
<td>TestPat</td>
<td>35</td>
<td>6</td>
<td>3</td>
<td>26</td>
<td>100%</td>
</tr>
<tr>
<td>TriTyp</td>
<td>119</td>
<td>0</td>
<td>3</td>
<td>116</td>
<td>100%</td>
</tr>
<tr>
<td>Relation</td>
<td>42</td>
<td>0</td>
<td>0</td>
<td>42</td>
<td>100%</td>
</tr>
</tbody>
</table>

Using the Java keyword “final” leads to a problem with the in-program mutation score, as there is no way letting the program know which mutants are equivalent. On the other hand, this is a design decision, as the data cannot be tampered with this way. There is also a displayed trace on how the data is built and how the hypotheses are validated. In case needed, the program displays where the original operator is (complete with type, column and row position), alongside with which operation it was mutated to, computed in real time based on the actual program. It is displayed if the mutant is killed or alive. For the first hypothesis, all the three other mutants connected via the subsumption are displayed, complete with the information if the mutant is killed or alive, alongside with the information if other mutant subsumes or is subsumed by the given mutant. The relation is asserted accordingly to the truth table displayed in Table 3.

Another design decision is to include all the mutants in the same project as the test program, alongside with an interface add-on so that the mutants could be placed into containers so that the test could be run efficiently on them, in a way that every mutant is loaded under the whole time of the execution. In a way, the mutant interface serves as a meta mutant, however this solution does not use bytecode like MuJava (Li, Praphamontripong, Offutt, 2009; Schuler & Zeller, 2009 and Untch, Offutt & Harrold, 1993). The obvious problem is that all of the mutants should be added to the project, which would not work with a bigger project or with a larger number of mutants. The modifications to the test programs were only the addition of the interface, as well as the comments and the spacing were slightly affected by the mutation tool. Note however that the main functions are not mutated by MuJava, nor by MuClipse, so those are omitted from the projects. Appendix O to Appendix S contains the modification to the programs to make them compatible with the tool.

The tool has an appropriate exception handling ability. In case one of the mutants crashes the tool remains unaffected, but notes that the mutant behaves differently from the original program (unless the original program crashes the same way). This design decision is similar to the behaviour of the Mothra system (Demillo, Guindi, McCracken, Offutt & King, 1988).

The tool has a length of about 3000 lines of code divided into 15 classes. For this reason, it is not included as an appendix.

For all the executions, a virtual machine using Windows 7 Professional complete with Eclipse and Netbeans was used. The VM was also dedicated to running the system, alongside with

Table 7 Results on the mutation analysis
2GB memory and 2 allocated CPU threads. The host OS was a Windows 7 x64 system on a HP EliteBook 8470p, alongside with an i3-3220, 8GB ram and an SSD. Netbeans 7.4 was used to develop the code.

The experimental flow is displayed on Figure 12. The input programs and the mutant creation are done with external sources and they are therefore marked with dashed lines. The program takes the test suites and runs it on the program P. The program then displays the result data for hypothesis 1, then for hypothesis 2 in a way that it can be copied into an XLS file directly. The general results for the mutation analysis are displayed too. Then, the person running the experiments can decide if all mutants are dead and if the test suite is “good enough”. In case the test suite is sufficient, the experiment is considered done. Otherwise the results as well as the remaining mutants are analysed. The mutants can be checked for equivalence or the test suite can be updated. The analysis on the mutation and the update of the test suite is done manually. Such activities are marked with bold lines. The boxes with normal lines are the activities the program does automatically. The program runs the tests on the original program and on all mutants, then a report on the result differences between all mutants individually and the original program is created for each test.

Figure 12 The process for the experiment (inclusive iterations)

To assert the first hypothesis, a pattern is matched on the different results based on the subsumption graphs. In case the pattern matches, the subsumption graph is considered OK for the given mutant for the given test for the given program, otherwise if the pattern does not match, manual check is performed as a pattern not matching means that the hypothesis does not hold for the given case. Afterwards, a second pattern is used on the different results
to assert the second hypothesis. Similarly to the first case, if the pattern matches OK is displayed and otherwise manual check is performed, as it means that the hypothesis does not hold for the case. It should be noted that the manual checks are done on the original, unmodified programs (and not on the tool adjusted ones), so even if the tool program is buggy (which can happen as showing that it is fault free is impossible), the manual control is unaffected by it and should produce the right results. Manual control was performed on all cases when the results were not in line with the hypotheses.

6.4 Objective 4: Data analysis

In this part the data is presented. Please note that the results for the first hypothesis are only shown as a conclusion here as the raw data is about 4000+ rows long, with 8000+ rows of additional data which can be used for tracing.

6.4.1 First hypothesis: The subsumption graph

Table 8 displays the results of the checks on the hypothesis which describes the subsumption graph. 4642 checks were made. It should be noted that every subsumption m1->m2 was asserted from both directions, effectively doubling the number of checks.

Table 8 ROR mutant results

<table>
<thead>
<tr>
<th>Number of checks</th>
<th>Number of &lt; type mutants</th>
<th>Number of &gt; type mutants</th>
<th>Number of &lt;= type mutants</th>
<th>Number of &gt;= type mutants</th>
<th>Number of == type mutants</th>
<th>Number of != type mutants</th>
<th>Sum number of ROR mutants</th>
</tr>
</thead>
<tbody>
<tr>
<td>153</td>
<td>973</td>
<td>1479</td>
<td>69</td>
<td>1821</td>
<td>147</td>
<td>4642</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>22</td>
</tr>
</tbody>
</table>

The hypothesis fails on the program TestPat and on the program CountNatural. In case of TestPat, it fails on the mutants TestPat_3, TestPat_4, TestPat_5 and TestPat_7, all of them on the original operator “<” in the while loop with the respective mutants “>=” (TestPat_3), “<=” (TestPat_4), “==” (TestPat_5) and TRUE (TestPat_7). CountNatural fails on the mutants CountNatural_7, CountNatural_8, CountNatural_9, CountNatural_10, CountNatural_11, Countnatural_12, all of them on the operator “>=” at the row containing the if condition with the respective mutants “<” (CountNatural_7), “<=” (CountNatural_8), “==” (CountNatural_9), “!=” (CountNatural_10), “FALSE” (CountNatural_11) and “TRUE” (CountNatural_12).

TestPat takes two string variables as an input and checks if the second string is part of the first string. Therefore tests are formulated as “string1, string2” for this program. The tests with the string input “aaa,b”, “a,b”, “aaa”, “ab” and “b,a” for the program TestPat failed to return a pattern in line with the first hypothesis according to the findings of this experiment.

The subsumption graph for the the test “aaa,b” with the relevant mutants for TestPat is displayed on Figure 13 with further information on Table 9. Mutants marked with live* should be dead according to the first hypothesis but could not be detected at function return value level. Mutants detected are marked with grey background and “dead”, while edges
where the subsumption fails are marked with strike-through and partially pointed lines. The mutant marked with live(e) is equivalent. The subsumption relations which hold according to the hypothesis are marked with normal edges.

**Figure 13** TestPat findings illustrated for an instance of < at line 18 with test case “aaa,b”, with the edges and mutants marked where the hypothesis was rejected.

**Table 9** displays a more detailed result for TestPat using the test case “aaa,b”. The original TestPat with the operator “<” controls if the string “b” is part of the string “aaa”, which is not the case. TestPat_3 with the mutated operator “>=” reports that “aaa” does not contain “b”, so the mutant is live. TestPat_4 containing the mutated operator “<=" is detected as it crashes with an exception. TestPat_5 with the mutated operator “==” gives the result that “aaa” does not contain “b”, which means that it is not detected. TestPat_7 containing the mutated operator “<=" crashes with a NullPointerException, so it is dead.

**Table 9** Result for TestPat focusing on the test case “aaa, b”, with the mutated code bold marked.

<table>
<thead>
<tr>
<th>Program: TestPat</th>
<th>Mutated row</th>
<th>Result</th>
<th>Mutant state</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>while (isPat == false &amp;&amp; iSub + patternLen - 1 &lt; subjectLen)</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>TestPat_2</td>
<td>while (isPat == false &amp;&amp; iSub + patternLen - 1 &gt; subjectLen)</td>
<td>No</td>
<td>live</td>
</tr>
<tr>
<td>TestPat_3</td>
<td>while (isPat == false &amp;&amp; iSub + patternLen - 1 &gt;= subjectLen)</td>
<td>No</td>
<td>live</td>
</tr>
<tr>
<td>TestPat_4</td>
<td>while (isPat == false &amp;&amp; iSub + patternLen - 1 &lt;= subjectLen)</td>
<td>NullPointerException</td>
<td>dead</td>
</tr>
<tr>
<td>TestPat_5</td>
<td>while (isPat == false &amp;&amp; iSub + patternLen - 1 == subjectLen)</td>
<td>No</td>
<td>live</td>
</tr>
<tr>
<td>TestPat_6</td>
<td>while (isPat == false &amp;&amp; iSub + patternLen - 1 != subjectLen)</td>
<td>No</td>
<td>live</td>
</tr>
<tr>
<td>TestPat_7</td>
<td>while ((isPat == false) &amp;&amp; true)</td>
<td>NullPointerException</td>
<td>dead</td>
</tr>
<tr>
<td>TestPat_8</td>
<td>while ((isPat == false) &amp;&amp; false)</td>
<td>No</td>
<td>live</td>
</tr>
</tbody>
</table>

It is interesting that the testing tool needed to support exception handling on the tests, as without it, some tests needed to be discarded as they crashed the tool preventing data generation and the results refusing the hypothesis were not present. Also, all evidence which
refuses the hypothesis is connected to a mutant that terminated with an exception. Appendix T gives an explanation on what is happening with test “aaa,b”, with the help of a modified, inner state exposed variant of TestPat displayed in Appendix U. The explanation is that the tool only detects difference on the final state returned by the functions, but cannot detect inner state differences. The mutants TestPat_3 (">=") and TestPat_5 ("==") terminate the loop directly and return that “aaa” does not contain “b”. The original file loops on the other hand, however this different behaviour cannot be observed on the final state resulting in the absence of propagation. A crashing mutant is however different, as a crash propagates. It can be noted that the top level mutant TestPat_8 (“FALSE”) can be detected weakly, as well as all mutants it is subsuming, inclusive TestPat_2 (">”). TestPat_6 (“!”=”) is equivalent.

A manual trace displayed similar reasons for the other test cases for TestPat where the subsumption did not apply. Based on the results of the manual traces, it is highly unlikely that the data not in line with the hypothesis is a result of the tool generating faulty data.

CountNatural takes an array of numbers as an input, and returns the number of natural numbers in the input array. The test with the input [1,-1] on the program CountNatural failed to return a pattern in line with the first hypothesis.

![Subsumption Graph for CountNatural](image)

**Figure 14** CountNatural findings illustrated for an instance of >= at line 14 with the test case [1,-1], with the edges and mutants marked where the hypothesis was rejected

**Figure 14** displays the subsumption graph on the relevant mutants for the program CountNatural with the the input test case [1,-1]. The mutants marked with * are alive, even though they should be detected according to the first hypothesis. The subsumption relations which aren’t in line with the hypothesis are marked. The original program counts numbers which are greater or equivalent to zero, also it returns ‘1’. “TRUE” counts everything, returns ‘2’ and is therefore detected. “>” counts positive numbers and returns ‘1’, thus it is alive. “==” counts only zeroes, returns ‘0’ and is detected. “!=” counts everything but zeroes, also ‘2’ and is therefore detected. “<=” counts negative numbers and zeroes, thus it returns ‘1’ and is therefore alive. “FALSE” counts nothing and returns ‘0’, it is also detected. “<” counts negative numbers, thus it counts ‘-1’ but the error does not propagate.

**Table 10** displays detailed results for the program CountNatural. In this case, there is not exception, but the problem with propagation applies. “<=” returns the right value, but it does this because it counts ‘-1’ instead of ‘1’. There is a state difference, but the error is masked at the return state. The same applies for the mutant containing “<”, it counts ‘-1’ instead of ‘1’, but as the number of negative elements is the same as the positive numbers, the error is masked on the return state and does not propagate.
Table 10 Result for CountNatural focusing on the testcase “[1,-1], with the mutated code bold marked

<table>
<thead>
<tr>
<th>Program: CountNatural</th>
<th>Mutated row</th>
<th>Result</th>
<th>Mutant state</th>
<th>Program behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>if (x[i] &gt;= 0) { 1</td>
<td>live</td>
<td>Counts natural numbers</td>
<td></td>
</tr>
<tr>
<td>CountNatural_6</td>
<td>if (x[i] &gt; 0) { 1</td>
<td>live</td>
<td>Counts positive numbers</td>
<td></td>
</tr>
<tr>
<td>CountNatural_7</td>
<td>if (x[i] &lt; 0) { 1</td>
<td>live</td>
<td>Counts negative numbers</td>
<td></td>
</tr>
<tr>
<td>CountNatural_8</td>
<td>if (x[i] &lt;= 0) { 0</td>
<td>detected</td>
<td>Counts zeros</td>
<td></td>
</tr>
<tr>
<td>CountNatural_9</td>
<td>if (x[i] == 0) { 2</td>
<td>detected</td>
<td>Counts all but zeros</td>
<td></td>
</tr>
<tr>
<td>CountNatural_10</td>
<td>if (x[i] != 0) { 2</td>
<td>detected</td>
<td>Counts all</td>
<td></td>
</tr>
<tr>
<td>CountNatural_11</td>
<td>if (true) { 2</td>
<td>detected</td>
<td>Counts all</td>
<td></td>
</tr>
<tr>
<td>CountNatural_12</td>
<td>if (false) { 0</td>
<td>detected</td>
<td>Counts none</td>
<td></td>
</tr>
</tbody>
</table>

6.4.2 Second hypothesis: The top level mutants

This hypothesis can be checked once per original operator per test, with looking on the top level mutants. In case zero or one of them is detected by a given test for a given original operator, the hypothesis holds, otherwise, it can be rejected. The hypothesis was run on all the programs, however only three test cases of Cal and one case of CountNatural are displayed here in depth.

The first program which generated data acting as a counterproof to the second hypothesis is CountNatural. CountNatural takes an array of numbers as an input and returns an integer which has the value of how many non-negative integer numbers are in the array. Appendix P displays the whole program while the interesting part is displayed below as the program is rather small. Mutants are marked with italic and the mutated operators are marked with bold. Figure 15 displays the results mapped to the proposed subsumption graph: detected mutants are marked with grey background, while the mutants alive have white background and bolded line.

![Figure 15](image)

**Figure 15** Results of CountNatural which displays two top level mutants detected

The program CountNatural is displayed below containing the original operator, as well as the generated ROR mutants. Note that the function is called CountPositive in the original source.
(Ammann & Offutt, 2008, pp. 16). Detailed information about the mutants and the results on executing them using the input [-1,1,-1,-1] can be found on Table 11. The original program counts the natural numbers in the input, ROR6 (“>”) counts the positive, ROR7 (“<=”) counts the negative numbers and zeros, ROR8 (“<”) count negatives, ROR9 (“==”) counts zeros, ROR10 (“!=”) counts everything than zeros, ROR11 counts everything, while ROR12 counts nothing.

```java
public int CountNatural ( int[] x )
{
    int count = 0;
    for (int i = 0; i < x.length; i++) {
        ORIG if (x[i] >= 0) {
            ROR6 if (x[i] > 0) {
            ROR7 if (x[i] <= 0) {
            ROR8 if (x[i] < 0) {
            ROR9 if (x[i] == 0) {
            ROR10 if (x[i] != 0) {
            ROR11 if (TRUE) {
            ROR12 if (FALSE) {
                count++;
            }
        }
    }
}
    return count;
}
```

Table 11 Partial results for CountNatural on the second hypothesis

<table>
<thead>
<tr>
<th>Program: CountNatural</th>
<th>Expression</th>
<th>Result for test input: X=[-1, 1, -1, -1]</th>
<th>Program behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>X[i]&gt;=0</td>
<td>1</td>
<td>Counts natural numbers</td>
</tr>
<tr>
<td>ROR6</td>
<td>X[i]&gt;0</td>
<td>1</td>
<td>Counts positive numbers</td>
</tr>
<tr>
<td>ROR7</td>
<td>X[i]&lt;=0</td>
<td>3</td>
<td>Counts negatives and zeros</td>
</tr>
<tr>
<td>ROR8</td>
<td>X[i]&lt;0</td>
<td>3</td>
<td>Counts negatives numbers</td>
</tr>
<tr>
<td>ROR9</td>
<td>X[i]==0</td>
<td>0</td>
<td>Counts zeros</td>
</tr>
<tr>
<td>ROR10</td>
<td>X[i]!=0</td>
<td>4</td>
<td>Counts all but zeros</td>
</tr>
<tr>
<td>ROR11</td>
<td>TRUE</td>
<td>4</td>
<td>Counts all</td>
</tr>
<tr>
<td>ROR12</td>
<td>FALSE</td>
<td>0</td>
<td>Counts none</td>
</tr>
</tbody>
</table>

Table 11 displays the results for the test case “[-1,1,-1,-1]” for the program CountNatural, which means that the program is tested with this test input. Two top level mutants are detected, while the third, > is alive. The background of the detected mutants is grey. The hypothesis is rejected, as two of the top level mutants are detected by the same test, as it is displayed on Figure 15. The mutant > alive is marked with bold.

A second rejecting example with CountNatural happens with the test case [1,-1] displayed before on Table 10 and Figure 14. Both of the top level mutants “==” and “TRUE” are detected, which acts as a counterproof.

There is also a third rejecting example with CountNatural using the test case [1, 1, 1, -1, -1, 0]. The actual results are displayed on Figure 16 and Table 12.
Figure 16 Results of CountNatural with the test case [1,1,1,-1,-1,0]

Table 12 Partial results for CountNatural on the second hypothesis

<table>
<thead>
<tr>
<th>Program: CountNatural</th>
<th>Expression</th>
<th>Result for test input: $X=[1,1,1,-1,-1,0]$</th>
<th>Program behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>$X[i] \geq 0$</td>
<td>4</td>
<td>Counts natural numbers</td>
</tr>
<tr>
<td>ROR6</td>
<td>$X[i]&gt;0$</td>
<td>3</td>
<td>Counts positive numbers</td>
</tr>
<tr>
<td>ROR7</td>
<td>$X[i] \leq 0$</td>
<td>3</td>
<td>Counts negatives and zeros</td>
</tr>
<tr>
<td>ROR8</td>
<td>$X[i]&lt;0$</td>
<td>2</td>
<td>Counts negatives numbers</td>
</tr>
<tr>
<td>ROR9</td>
<td>$X[i]==0$</td>
<td>1</td>
<td>Counts zeros</td>
</tr>
<tr>
<td>ROR10</td>
<td>$X[i]!=0$</td>
<td>5</td>
<td>Counts all but zeros</td>
</tr>
<tr>
<td>ROR11</td>
<td>TRUE</td>
<td>6</td>
<td>Counts all</td>
</tr>
<tr>
<td>ROR12</td>
<td>FALSE</td>
<td>0</td>
<td>Counts none</td>
</tr>
</tbody>
</table>

In case of the program CountNatural with the test case [1,1,1,-1,-1,0], all the mutants for the original operator are detected. That means that all three top level mutation operators are detected with the test case [1,1,1,-1,-1,0]. It can also be noted that this particular test can detect all ROR mutants created for CountNatural (besides one equivalent mutant). Therefore it is ROR adequate on its own.

Figure 17 Results of Cal with test input “Jan. 1, Dec. 30, Year 100” mapped to subsumption graph

The second program is Cal which generated data acting as counterproof with three test cases which are displayed on Figure 17. Detected mutants are marked with
grey background, while the stillborn mutant FALSE is marked with dotted line and a white background. The original operator “<=" is present in the for-loop in the program Cal (Appendix O displays the whole program). Table 14 displays the dead mutants with marking or the given test cases. The original row as well as the mutants with a short description on their behaviour can be found on Table 13. Compared to the original in case of the test input “January 1, December 30, Year 100”, ROR29 (“>”), ROR30 (”>=") and ROR32 (”==") skips a loop; ROR31 (“<”) and ROR33 (!”) skips one iteration, ROR34 (“TRUE”) crashes with an exception and the mutant with “FALSE” is stillborn.

Table 13 The mutated row in Cal alongside with the mutants and a short description on the effect of the syntactical changes

<table>
<thead>
<tr>
<th>Program: Cal</th>
<th>Expression</th>
<th>Program behavior (compared to original in case of the test input Jan. 1, Dec. 30, Year 100)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORIG</td>
<td>for (int i = month1 + 1; i &lt;= month2 - 1; i++) {</td>
<td>(This is the original program)</td>
</tr>
<tr>
<td>ROR29</td>
<td>for (int i = month1 + 1; i &gt; month2 - 1; i++) {</td>
<td>Skips loop</td>
</tr>
<tr>
<td>ROR30</td>
<td>for (int i = month1 + 1; i &gt;= month2 - 1; i++) {</td>
<td>Skips loop</td>
</tr>
<tr>
<td>ROR31</td>
<td>for (int i = month1 + 1; i &lt; month2 - 1; i++) {</td>
<td>Skips one iteration in loop</td>
</tr>
<tr>
<td>ROR32</td>
<td>for (int i = month1 + 1; i == month2 - 1; i++) {</td>
<td>Skips loop</td>
</tr>
<tr>
<td>ROR33</td>
<td>for (int i = month1 + 1; i != month2 - 1; i++) {</td>
<td>Skips one iteration in loop</td>
</tr>
<tr>
<td>ROR34</td>
<td>for (int i = month1 + 1; TRUE; i++) {</td>
<td>Exception</td>
</tr>
<tr>
<td>STILLBORN</td>
<td>for (int i = month1 + 1; FALSE; i++) {</td>
<td>Stillborn</td>
</tr>
</tbody>
</table>

“FALSE” is not listed because it was stillborn. Dead mutants are marked with grey, while mutants alive are left white. Considering the fact, that the test case “1,1,12,30,100” (meaning January 1, December 30, Year 100) detected every non-stillborn mutant, the hypothesis is already falsified. Cal returns the difference in number of days between two dates within the same year, thus the input “1,1,12,30,100” means that the number of days between January 1 and December 30 in the year 100. The mutated row is the one responsible to decide how many months are present between the two input months. The other two only detect two of the top level mutants (“==” survives in both case), but that is also enough to break the hypothesis. Therefore, the second hypothesis is rejected. It should also be noted that “FALSE” is a medium level mutant, so “1 1 12 30 100” detects all three top level mutants at once.

Table 14 Partial results for Cal on the second hypothesis

<table>
<thead>
<tr>
<th>Results: Cal</th>
<th>Original</th>
<th>Mutants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test input</td>
<td>&lt;=</td>
<td>&gt;</td>
</tr>
<tr>
<td>Jan. 1, Mar. 1,</td>
<td>60</td>
<td>exception</td>
</tr>
<tr>
<td>Year 400</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jan. 1, Dec. 30,</td>
<td>363</td>
<td>60</td>
</tr>
<tr>
<td>Year 100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jan 1, Mar. 29,</td>
<td>88</td>
<td>59</td>
</tr>
<tr>
<td>Year 4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 15 displays a list on every program checked with the second hypothesis with the tool using strong mutation. Both Cal and CountNatural reject the hypothesis. It can be noted that the tool can detect mutants strongly, however the test case “aaa,b” with the program TestPat
which rejects the first hypothesis actually would reject the second hypothesis too if weak
detection criteria would be used. Figure 13 alongside with Appendix T displays that with
weak criteria, the test case “aaa,b” actually detects all the mutants besides the equivalent one.
This is not in line with the second hypothesis either, as two top level mutants are detected
using weak detection criteria.

**Table 15** Results on the checks of the second hypothesis using strong detection
criteria

<table>
<thead>
<tr>
<th>Program</th>
<th>Number of checks</th>
<th>Number of checks ok</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cal</td>
<td>$5^5 = 25$</td>
<td>22</td>
</tr>
<tr>
<td>CountNatural</td>
<td>$6^2 = 12$</td>
<td>9</td>
</tr>
<tr>
<td>Relation</td>
<td>$6^3 = 18$</td>
<td>18</td>
</tr>
<tr>
<td>TestPat</td>
<td>$5^8 = 40$</td>
<td>40</td>
</tr>
<tr>
<td>TriTyp</td>
<td>$17^34 = 578$</td>
<td>578</td>
</tr>
</tbody>
</table>
7 Conclusion

In this chapter, the conclusions are drawn based on the data gathered while performing the experiment.

Table 16 Number of checks on the first hypothesis

<table>
<thead>
<tr>
<th>Sum number of ROR mutants per program</th>
<th>Number of mutants</th>
<th>Number of tests</th>
<th>Number of checks</th>
<th>Times the first hypothesis did not hold</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cal</td>
<td>34</td>
<td>5</td>
<td>170</td>
<td>0</td>
</tr>
<tr>
<td>CountNatural</td>
<td>12</td>
<td>6</td>
<td>72</td>
<td>6</td>
</tr>
<tr>
<td>Relation</td>
<td>42</td>
<td>3</td>
<td>126</td>
<td>0</td>
</tr>
<tr>
<td>TestPat</td>
<td>29</td>
<td>8</td>
<td>232</td>
<td>16</td>
</tr>
<tr>
<td>TriTyp</td>
<td>119</td>
<td>34</td>
<td>4046</td>
<td>0</td>
</tr>
<tr>
<td>Sum</td>
<td>236</td>
<td>56</td>
<td>4646</td>
<td>22</td>
</tr>
</tbody>
</table>

For the first hypothesis, Table 16 displays the number of checks performed on the given operators. It can be noted that “==”, “<” and “>” was responsible for most of the mutants, while “<”, “!” and “>=” had lower numbers. Mutants are detected if they are distinguished from the original program by comparing the returned values after the programs containing the functions are executed.

The pattern for the subsumption relations between m1->m2 are displayed on Table 17. In case of the tested program TestPat, four of eight test cases resulted in experimental data which was not in line with the hypothesis 1 based on the article of Kaminski, Ammann & Offutt (2011). Manual trace available in Appendix T displays that the data presented by the experiment is indeed valid and the pattern does not correspond for TestPat in those four cases to the hypothesis. CountNatural also displays different behaviour in one test case (out of six) from what is expected from the hypothesis. Hence, it can be concluded that the first hypothesis is rejected.

Table 17 The subsumption pattern m1->m2

<table>
<thead>
<tr>
<th>Mutant in question alive</th>
<th>Mutant subsuming it alive</th>
<th>Mutant subsuming it detected</th>
<th>Mutant subsumed by it alive</th>
<th>Mutant subsumed by it detected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutant in question alive</td>
<td>OK</td>
<td>Not OK</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>Mutant in question detected</td>
<td>OK</td>
<td>OK</td>
<td>Not OK</td>
<td>OK</td>
</tr>
</tbody>
</table>

Table 18 displays the number of checks performed on the second hypothesis. Note that these numbers are based on the data gathered by the tool, which used strong detection criteria.
Table 18 Number of checks on the second hypothesis

<table>
<thead>
<tr>
<th>Program</th>
<th>Number of checks</th>
<th>Times the second hypothesis did not hold (strongly)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cal</td>
<td>25</td>
<td>3</td>
</tr>
<tr>
<td>CountNatural</td>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>Relation</td>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td>TestPat</td>
<td>40</td>
<td>0</td>
</tr>
<tr>
<td>TriTyp</td>
<td>527</td>
<td>0</td>
</tr>
<tr>
<td>Sum</td>
<td>622</td>
<td>6</td>
</tr>
</tbody>
</table>

As shown in the case of CountNatural and Cal, a single test could detect more than one top level mutant, which does not correspond to the pattern excepted on the experimental result data displayed on Table 19, which pattern is based on the second hypothesis. As a test case detecting more than one top level mutant for a given original operator does not match the hypothesis, and four such test cases were found, the second hypothesis is rejected.

Table 19 The expected results after running a single test case on the top level mutants found in a single subsumption graph based on a single mutated operator

<table>
<thead>
<tr>
<th>Status of Top Level Mutants</th>
<th>Resulting pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1 live</td>
<td>m2 live</td>
</tr>
<tr>
<td>live</td>
<td>live</td>
</tr>
<tr>
<td>dead</td>
<td>live</td>
</tr>
<tr>
<td>live</td>
<td>dead</td>
</tr>
<tr>
<td>live</td>
<td>dead</td>
</tr>
<tr>
<td>live</td>
<td>dead</td>
</tr>
<tr>
<td>dead</td>
<td>live</td>
</tr>
<tr>
<td>dead</td>
<td>live</td>
</tr>
<tr>
<td>live</td>
<td>dead</td>
</tr>
<tr>
<td>dead</td>
<td>dead</td>
</tr>
<tr>
<td>live</td>
<td>dead</td>
</tr>
<tr>
<td>dead</td>
<td>dead</td>
</tr>
<tr>
<td>live</td>
<td>dead</td>
</tr>
<tr>
<td>dead</td>
<td>dead</td>
</tr>
<tr>
<td>live</td>
<td>dead</td>
</tr>
<tr>
<td>dead</td>
<td>dead</td>
</tr>
<tr>
<td>live</td>
<td>dead</td>
</tr>
<tr>
<td>dead</td>
<td>dead</td>
</tr>
<tr>
<td>live</td>
<td>dead</td>
</tr>
<tr>
<td>dead</td>
<td>dead</td>
</tr>
<tr>
<td>live</td>
<td>dead</td>
</tr>
<tr>
<td>dead</td>
<td>dead</td>
</tr>
<tr>
<td>live</td>
<td>dead</td>
</tr>
<tr>
<td>dead</td>
<td>dead</td>
</tr>
<tr>
<td>live</td>
<td>dead</td>
</tr>
<tr>
<td>dead</td>
<td>dead</td>
</tr>
<tr>
<td>live</td>
<td>dead</td>
</tr>
<tr>
<td>dead</td>
<td>dead</td>
</tr>
<tr>
<td>live</td>
<td>dead</td>
</tr>
<tr>
<td>dead</td>
<td>dead</td>
</tr>
</tbody>
</table>

The second hypothesis is related to the first hypothesis according to the article of Kaminski, Ammann & Offutt (2011) describing both hypotheses, so after the first hypothesis was rejected, it was expected that the second one would be rejected too. On the other hand the article did not discuss in depth how the second hypothesis is resulted from the first hypothesis. It can also be noted that the tool used strong detection criterion to detect mutants: In case weak mutation would be used, there would be at least one test case for TestPat which could weakly detect two top level mutants, resulting in data which is not in line with the second hypothesis either. The difference between weak and strong mutation is however less important in this case, as the hypothesis can be rejected based on the strong detection criteria too.

The findings of this experiment can be recreated without the tool, as the tool’s role in the end was only to detect the exceptions if the hypotheses did not hold, and to collect enough data so that the statistical significance of the results could be discussed. To assert the findings, the respective programs are available via the original source (Ammann & Offutt, 2008) or via this document, the mutants can be created manually or by e.g. MuJava and the test cases can be executed on them via a Java compiler. The results can be validated (and were validated) by hand in the end.
The aim of this report is to examine the two hypotheses. The first hypothesis states that a test T that detects a mutant m, should also detect mutant m’ in case m subsumes m’ according to any of the graphs in Figure 5 - Figure 10. The second hypothesis states if a given test detects one of the top level mutants in the hierarchy of a subsumption graph for a given original operator, it would not detect the other two.

The first hypothesis on subsumption is rejected for strong mutation. Moreover, in several cases the error caused by the mutant was masked in the same loop it occurred. To detect this, a very fine-grained form of weak mutation is required. The term weak mutation is also used very loosely in the terminology, so a tool stating that it can detect mutants weakly may actually mean that it can detect on component level which is faster but course-grained. The problem here is observability. The fine-grained weak mutation used in this work can be infeasible in some cases. For example when object-oriented with encapsulation principles are used, object variables can be private, so observing them can be hard. The other problem is that even though most of the tools on mutation analysis are described that they are capable of weak mutation testing using state difference, the question is if they are capable of doing that with a more fine-grained observation than the one that is used in this work. This is a word of warning for both practitioners and researchers.

We saw a pattern regarding the failure of hypothesis one. All situations were caused by conditions in loops. Either the subsuming mutant was detected because it caused an exception or the subsumed mutant caused an error that was masked before the end of the loop.

The second hypothesis is rejected. The number of cases where it does not hold is low, however a simple program containing two operators is enough to show a case where more than one top level mutants is detected by the same test. We found a result where a single test case could detect all three top level mutants, and in one case, a single test case was ROR mutation adequate on its own.

It is interesting to note that CountNatural is a simple program containing only two operators, and still the hypotheses did not hold. To make things worse, the logic of CountNatural is identical in logic to the commonly used buildstone in bigger programs which goes through an array and then returns how many of the elements have a given attribute.

When it comes to the tool software created for the experiment, no open source release is planned, partially because the somewhat problematic integration of the mutants and partially because it is assumed that the program has little use when it comes to do other things than validating the given hypotheses. On the other hand, the examples rejecting the hypotheses are presented in this document and can be validated by hand.
8 Discussion

8.1 Achievment of goals

The following objectives were achieved for the aim:

- Objective 1: For the first hypothesis, the relations for the mutants based on what is the original operator are identified. For the second hypothesis the top level mutants are identified. The set of literature containing all publications relevant to this study is created using the method proposed, thus the literature study is performed as described until the halting criteria is reached, also no previously unknown and relevant literature could be found. Subsumption is described using the literature and the necessary patterns are identified to assert on which data that is needed to investigate the hypotheses. As the approach on the literature study is followed, and the necessary patterns are identified, the objective is regarded achieved.

- Objective 2: The mutation analysis process is described and used to identify what automation is needed for the experiment. The programming language is described as a factor. Tool support is evaluated and MuJava is selected for mutant generation with MuClipse used as a control tool. For the rest of the data gathering process, it is identified that a tool is needed to be built as the data needed for this experiment is different from the data provided by existing tools. The programs alongside with their structure and the number of operators they hold are identified as a variable, with the programmers’ coding style being a factor. Therefore four programs from external sources are selected: Three widely used and a fourth to cover all operators. A fifth, simple program, which program has all relation operators, is written by the author. The fifth program was written with the purpose to verify the external mutation tools, but also to be used as a control variable. Test suites are identified as a variable, with the need of mutation adequateness. Mutation adequate test suites are created and all equivalent mutants are analysed so that they are marked equivalent accordingly. The set of literature created according to the method description of objective one is used to determine all the experimental variables. Therefore, with taking into consideration that the experimental setup could be built, the objective is considered reached.

- Objective 3: The experimental setup using the requirements and variables identified in objective 1 and 2 is built. The data was gathered in a way it could be analysed. The data gathered is regarded sufficient to investigate the two hypotheses, therefore the objective is considered fulfilled.

- Objective 4: The data was analysed and both hypotheses can be rejected due to the existence of contradictory examples. Therefore objective 4 is considered completed.

The aim is considered fulfilled as all the four objectives are achieved. All objectives are fulfilled as described above. The first hypothesis is rejected, as the hypothesis is not limited to weak mutation, and some subsumption relations do not hold for strong mutation.

When infection can be detected at instruction level for the mutated software the hypothesis cannot be rejected with the gathered data, however detecting such small changes can be problematic. In case weak mutation is used on component level or on return values of functions, all subsumption relation does not hold as showed by the results of this report.
The second hypothesis is also rejected, as it could be shown with multiple programs with multiple test cases that a test case can detect more than one top level mutant with a single test. It can be noted that the second hypothesis is rejected regardless if strong or weak detection is used. In case a mutant is detected strongly, it should also be detected weakly. Therefore the two example cases which are rejecting strongly are rejecting the hypothesis weakly too.

Therefore, the aim is considered fulfilled as both hypotheses are rejected.

8.2 Validity and limitations of the study

First, the quality can be asserted with the combined quality factors defined by Oates (2006, pp. 292) as objectivity, reliability, internal- and external validity and by Wohlin, Runeson, Höst, Ohlsson, Regnell & Wesselin (2012, pp. 104 & 108) as conclusion and construct validity.

Objectivity is achieved by using the variables for the experiments based on other research performed by other people, based on a sterling literature survey. The programs besides the program used as a reference variable are written by others and changes to them are only made so that they could be integrated with the tool to assert the hypothesis. The changes do not affect their behaviour when it comes to mutation testing. The test suites are mutation adequate. In case there is a test that rejects the hypothesis, it is unimportant how it was chosen, since the hypotheses can be rejected based on the existence of any test that contradicts the hypothesis.

Reliability is achieved by the documentation of the research; also the research is described well enough so that it could be performed again by another person to get similar results. The survey parameters are described alongside with the selection criteria to get the variables for the experiment. The experiment is then described. The results can be validated without the tool, as its purpose was to automatize the process. The reader of this report can validate the results of this report by hand as all necessary information is discussed in the report. That is, the programs, the mutants, the tests and the results are presented. That is all that is needed to replicate the study. This manual validation was done by the author resulting in the same results. However, the author can send the tool for examination to those who want it. The size of the tool (around 3000 rows in Java) prevents it for being attached as an appendix to this report.

Internal validity is achieved by the fact that the program gathers the data it should and is built in a way that the trace of information can be asserted manually. Therefore, if there would be some bug resulting in data which would not be in line with how the data really should be, it can be tracked. The external tool to create the ROR mutants was tested with a small program which had all the relation operators to see, if they produced the right results. The ROR mutants for this simple program are also created with a second program (MuClipse) to see if there were any differences, also the results are compared to the literature on which mutants should be created. The tool can only detect mutants on the output return values, but as every tested program consists of one function for the program logic, this is the same in this case as detecting mutants by comparing function return values. This is a limitation in state difference detection ability. However where it was needed, additional data is gathered by hand and displayed in the appendix, so that it can be asserted. The tool to gather the empirical data is built in such a way that it should detect unexpected results and
fail spectacularly in such cases (giving also trace on which point the problem was introduced for further evaluation). A minimal program containing nothing but the relation operators is also mutated and evaluated with the tool to show that the tool handles the possible ROR mutants. The results are then compared to manual computations to show that the tool produces the right results. The program is used as a control variable through the process with other words. This is however not sufficient to show the tool is correct (which is impossible because of the halting problem), but shows some kind of quality. In case unexpected results are produced, all unexpected results tracing back to a given test on a given program mutant are computed manually, to decide if there is a bug in the hypothesis testing tool. A statistical significant number of expected results are also manually checked randomly after every bigger feature update. A dedicated suite for unit testing on the tool is considered, but such an addition would double the development time, which is considered infeasible for this research project and unneeded to as everything it generated could be doublechecked manually. Finally, all results showing a contradiction to the hypotheses are analyzed by hand and presented in this work. It is evident from these examples that the data is correct and not results of any mistake or misinterpretation of the data.

**External validity** is achieved by presenting results in a way they will be generalizable. The experimental setting and treatment (Wohlin, Runeson, Höst, Ohlsson, Regnell & Wesselén, 2012, pp.110) is achieved by the use of a language -Java- which is commonly used by practitioners. The ROR mutants are created with a program actively in development. The programs used are common Java programs, without any unusual construction. Therefore it expected that the results presented in this study occur in other Java software too, however it is not known from the experimental data how often it would happen. The example programs are small, but some of them are extensively used for research on mutants to the present day, with results relevant in general. It should be noted that there are different implementations running under the same name, which can have different amount of mutants, equivalent mutants and needed test cases. It was also considered important to have a mutant adequate coverage, which would be much harder with larger programs, as detecting and marking all equal mutants on larger programs can be difficult (analysis needs to be done).

**Construct validity** is achieved by an appropriate underlying survey. The constructs are defined well enough to build the experiment on them. The literature is mostly very well defined. There were some minor uncertainties in some cases (like in case of weak mutation, O cost of mutation), but was followed up accordingly. However, these factors are only partially interesting for the experiment itself, and do not affect the study or the results.

**Conclusion validity** is achieved by holding the assumptions, having reliable measurements (the outcome of the tests is deterministic), having a reliable implementation and having a fixed experimental setting (like running everything on a dedicated virtual machine). The survey is based on all the found data, as well as the experiment was set up to display every result gathered: No fishing for results is performed and no data is omitted because it is not in line with the research. These factors are also the relevant checks described by Wohlin, Runeson, Höst, Ohlsson, Regnell & Wesselén (2012, pp. 104-106). The statistical power is not so relevant, as the hypotheses are using all quantors, and the research did find the cases rejecting them. There is considerable attention given to the traceability of data, so that the implementation is reliable and the environment is protected from variables which should not interfere with the test milieu. The tool tests the first hypothesis from both ends of a subsumption relation, both from the subsumed mutant and the subsuming mutant, using
different code for the actual pattern checks. Results not being symmetrical can be used to
detect bugs in the tool as a result. All results suggesting that the hypotheses do not hold are
properly analysed by hand.

It can however not be ensured that something is overlooked, misunderstood or taken for
granted. Incompetence of the author and knowledge maturation can be a factor too. Faults
can be made during the research like omitting important papers or making coding errors, but
such errors are unintentional.

Juristo, Moreno and Vegas (2004) describe four maturity levels when it comes to a maturity
of a technique: Laboratory study, formal analysis, laboratory replication and field study. The
report focuses on a laboratory study on a set of small programs. The experiment
performed is based on a set of variables described by previous studies, with programs and
criteria created by other researchers presented in other papers. Wohlin, Runeson, Höst,
Ohlsson, Regnell & Wesselén (2012, pp. 106) describes that subjective measurements are less
reliable than objective measurements. To lower uncertainty, the literature survey was started
on three large surveys done on the target domain in connection with the original article
containing the researched hypotheses (with citation count at the time of writing): “Mutation
2000: Uniting the Orthogonal” with 241 citations by Offutt & Untch (2000), the relevant
part of “Introduction to Software Testing” by Amman & Offutt (2008, pp. 210-212) with 584
citations and “An Analysis and Survey of the Development of Mutation Testing” by Jia &
Harman (2011) with 339 citations. “Mutation 2000” is a bit older, however there is a
publication from one of the authors, “A Mutation Carol: Past, Present and Future” (Offutt,
2011) with 13 citations. The three large surveys on the domain give a good overview of the
390 research papers, 16 PhD and 7 master works identified on the topic of mutation testing
until 2010 (Jia & Harman, 2011), which is also extended by a search on the topic on articles
released after 2010. Therefore, even though the information selection for variables is
subjective, the information can be considered to be selected according to other research
publications. However, it cannot be stated for sure that something was not overlooked, even
though the literature was mined via different routes. The resulting data can be gathered
again, the different results on the different mutants for the given programs can be computed
without the tool built for the research. The objectivity of the resulting data is checked
manually using a standard Java compiler, so as long as it can be assumed that the Java
compiler computes the right results, the data should be right. Therefore, the data gathered is
considered objective, meaning that the laboratory study has been conducted.

**Formal analysis** can be discussed. The statistical significance of the findings is limited, as
only smaller programs are used with a limited sized test suite. However, both of the proposed
hypotheses are defined with the all quantifier, meaning that a single instance, which is not in
line with the hypothesis, can actually falsify it. This is also in line with the fact that he
hypothetico-deductive reasoning is used. Even though black swans are really rare in nature,
their existence falsifies the hypothesis “every swan is white”. Also, the fact that an exception
can be shown on a small, laboratory program means that a bigger, real world program has a
higher chance to contain something which does not conforms to the second hypothesis.
**Laboratory replication** and **field studies** are not part of this research, thus the maturity
level of the findings is still low.
8.3 Discussion of ethics

When it comes to general social aspects, people are depending on their devices, so having devices which fail less because there are better means to control quality is beneficial. Mutation testing / mutation adequate criterion is one of the strongest testing techniques to assert quality, thus being able to use it on commercial products can be considered beneficial to the general public. Having cheaper mutation testing is one way to make more companies affording mutation testing as a method to achieve and control quality.

Research ethics are considered. Findings of others are marked and referenced accordingly. No previously done research by the author is reused in this report. Research quality is taken into consideration, the methods are followed and the research process is done accordingly. All relevant results which are found are displayed in this report, without cosmetics or fishing for results. No results are omitted because they showed different results from the findings in this report. The document can have unintentional defects and improper descriptions, but none of them were placed intentionally. Knowledge maturation of the author, domain knowledge maturation, omitted findings under the literature and faults in the experiment are factors which could affect the outcomes of this research, even though the author did his best to produce work with quality.

8.4 Contributions

This master dissertation presents three main contributions. The first contribution is a comprehensive overview on the domain of mutation testing, which is the result of the literature review. As part of the domain description, practitioners are warned that there exist different interpretations on what weak mutation is and how it should be performed.

The second contribution is the display of a masking behaviour in the domain of mutation testing, which was previously unknown. That means that depending on the program mutated, some errors may not propagate, which can affect the outcome of the mutation analysis.

The third contribution is that it is shown that the granularity of where the mutants are detected under execution is important when performing a reduced form of mutation. That is, the proposed subsumption graphs are rejected for strong mutation, but no evidence is found on weak rejection.

This is the first time an empirical assertion on the ROR subsuption graphs was performed, giving new empirical support on the area. This is a reason too why the research can be considered novel and original (Sandberg & Alvesson, 2012). The results connected to constrained mutation are important, as the topic is “hot”, with active research going on using the findings described in the paper "Better Prediction Testing" and further in the article “Improving logic-based testing” by Kaminski, Ammann & Offutt (2011 & 2013). Minimizing mutation sets is seemingly a strong domain with currently active development and findings which are connected to this article via the ROR operators (Ammann, Delamaro & Offutt, 2014; Kurtz, Ammann, Delamaro, Offutt & Deng, 2014 & Just, Kapfhammer, & Schweiggert, 2012b). As they promise to make mutation analysis much cheaper, the findings in this research about the reduced ROR operator as well as the domain description, the masking behaviour and the importance of detection granularity may be interesting for future research.
The results can be connected to the analysis performed on mutants with regards to how easily they could be detected, so the results are in line with the findings of Yao, Harman & Jia (2014) and giving example that in some cases depending on the construct and the operator, faults are really easy to find. René, Kapfhammer & Schweiggert (2012a) used Kaminski, Ammann & Offutt’s reduced ROR mutation for an empirical study alongside with another reduced mutation operator and noted that the mutation score got worse, which they interpreted as a overestimation on how effective mutation testing really is. In the light of this study, taking into consideration that the used tool MAJOR presumably uses weak mutation as defined by Howden (1982) based on the couple magnitude faster weak mutation detects, it can happen that with the larger and more complex programs they used (around 3-4000 LoC), the effect that some high level mutants are actually not propagating can be considered a factor. However as they eliminate mutants which are easier to be detected, a lower mutation adequacy score is expected. It is proposed that the reduced ROR operator should be used in the tools, but the findings in this document should be also considered when doing so (Kaminski, Ammann & Offutt, 2013).

8.5 Future research

For the future, the main question is how the first hypothesis would hold weakly, even against larger, more complex problems, as there was no data found which could reject the first hypothesis weakly. Lab test replication would help the maturity of the finding, as well as field studies performed on larger programs not considered “toy examples”, maybe even using different programming languages. It could also be interesting to investigate with larger programs, if it would be possible that two of the highest level mutants would propagate with a subsumed mutant, which would not. The most important question would be in such cases what happens with larger systems, is it really safe to only use the highest level ROR mutants instead of all seven. It could be interesting to check on how the program structure could affect mutation analysis using the ROR operator, as it seems like using loops can have masking effect which cannot be detected on function return level. Exceptions were only seen in loops in the case of this research, it could be interesting to see if they could affect subsumption outside loops too.

Statistical analysis could be used to study how small the difference is between the original ROR operator and the ROR operator only creating the three top level mutants. The study could also focus on a similarly fine grained weak mutation technique, to assert if the subsumption claim holds in practice if the internal state difference could be observed within the functions for the mutant. It can also interesting to research on larger programs of the effect that how many times the subsumption fails because the subsumed mutant only infects, but the subsuming mutant propagates too. In the article holding the claims (Kaminski, Ammann & Offutt, 2011), the two claims are related, thus the research could focus on the second claim, as it is easier to be asserted with tools built for mutation analysis.

Considering mutation testing as a whole, looking at the other four important operators to make mutation more cost effective could also be interesting, like defining subsumption on ABS, AOR, LCR and UOI.

The findings of René, Kapfhammer & Schweiggert (2012) indicate that the mutation analysis scores are too high compared to the effectiveness of the actual test suites, as looking at the results, it can be noted that some mutants are detected by (nearly) all tests, while some
require special attention. Their study could be checked in light of the finding of this
document to assert if mutant detection granularity played a role in their results.

The dynamical mutant subsumption graphs (DMSG) proposed by Kurtz, Ammann,
Delamaro, Offutt & Deng (2014) report that they needed 6 tests for Cal to reach 100% mutation coverage. The DMSG approach needs only one more test case than this thesis which could reach mutation coverage with a minimal set of tests with five tests on ROR. DMSG is promising and it uses the reduced ROR. The fact that the subsumption graphs do not necessarily hold should be considered here too.

Moving to the second hypothesis, this study shows evidence that there are tests that detect three top level mutants. The findings indicate that not only mutants can be trivial, there can also be some original operators (or constructs) which if mutated are very easy to be detected. This can be interesting as this would mean that the given fault might mask other faults when detected, which is maybe unwanted in case of higher order mutants are used. This is however only a speculation and the number of operators are far too low in this study to draw conclusions.
References


57


import java.io.*;

class cal {

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

        int numDays;
        if (month2 == month1) // in the same month
            numDays = day2 - day1;
        else {

            // Skip month 0.
            int daysIn[] = {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;

        System.out.println("Enter month1: ");
        month1 = getN();
        System.out.println("Enter day1: ");
day1 = getN();
System.out.println ("Enter month2: ");
month2 = getN();
System.out.println ("Enter day2: ");
day2 = getN();
System.out.println ("Enter year: ");
year = getN();
// 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;
    System.out.println ("invalid month1, choosing 1.");
}
if ( (month2 < 1) || (month2 > 12) )
{
    month2 = 1;
    System.out.println ("invalid month2, choosing 1.");
}
if ( (day1 < 1) || (day1 > 31) )
{
    day1 = 1;
    System.out.println ("invalid day1, choosing 1.");
}
if ( (day2 < 1) || (day2 > 31) )
{
    day2 = 1;
    System.out.println ("invalid day2, choosing 1.");
}
while ( month1 > month2 )
{
    System.out.println ("month1 must be prior or equals to month2");
    System.out.println ("Enter month1: ");
    month1 = getN();
    System.out.println ("Enter month2: ");
    month2 = getN();
}
if ( (year < 1) || (year > 10000) )
{
    year = 1;
    System.out.println ("invalid year, choosing 1.");
}
T = cal (month1, day1, month2, day2, year);
System.out.println ("Result is: " + T);

// Read (or choose) an integer
private static int getN ()
{
    int inputInt = 1;
    BufferedReader in = new BufferedReader (new InputStreamReader (System.in));
    String inStr;
try
{
    inStr    = in.readLine ();
    inputInt = Integer.parseInt(inStr);
}
catch (IOException e)
{
    System.out.println ("Could not read input, choosing 1.");
}
catch (NumberFormatException e)
{
    System.out.println ("Entry must be a number, choosing 1.");
}

return (inputInt);
} // end getN
class countPositive
{

public static int countPositive (int[] x)
{  //Effects: If x==null throw NullPointerException
   // else return the number of
   // positive elements in x.
   int count = 0;

   for (int i=0; i < x.length; i++)
   {
      if (x[i] >= 0)
      {
         count++;
      }
   }
   return count;
}

// test: x=[-4, 2, 0, 2]
//       Expected = 2

public static void main (String []argv)
{  // Driver method for countPositive
   // Read an array from standard input, call countPositive()
   int []inArr = new int [argv.length];
   if (argv.length == 0)
   {
      System.out.println ("Usage: java countPositive v1 [v2] [v3] ... ");
      return;
   }

   for (int i = 0; i< argv.length; i++)
   {
      try
      {
         inArr [i] = Integer.parseInt (argv[i]);
      }
      catch (NumberFormatException e)
      {
         System.out.println ("Entry must be a integer, using 1.");
         inArr [i] = 1;
      }
   }

   System.out.println ("Number of positive numbers is: " + countPositive (inArr));
}
Appendix C - TestPat

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

class TestPat
{

public static void main (String []argv)
{
    final int MAX = 100;
    char subject[] = new char[MAX];
    char pattern[] = new char[MAX];
    if (argv.length != 2)
    {
        System.out.println
            ("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)
    {
        System.out.println
            ("Pattern string is not a substring of the subject string");
    }
    else
    {
        System.out.println
            ("Pattern string begins at the character " + n);
    }
}

public TestPat()
{
}

public 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

    final int NOTFOUND = -1;
    int iSub = 0, rtnIndex = NOTFOUND;
    boolean 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);
}
Appendix D - TriTyp

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

// Jeff Offutt--Java version Feb 2003
// Classify triangles
import java.io.*;

class trityp
{
    private static String[] triTypes = { "", // Ignore 0.
        "scalene", "isosceles", "equilateral", "not a valid triangle"};
    private static String instructions = "This is the ancient TriTyp program.
Enter three integers that represent the lengths of the sides of a triangle.
The triangle will be categorized as either scalene, isosceles, equilateral or invalid.
";

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

        System.out.println (instructions);
        System.out.println ("Enter side 1: ");
        A = getN();
        System.out.println ("Enter side 2: ");
        B = getN();
        System.out.println ("Enter side 3: ");
        C = getN();
        T = Triang (A, B, C);

        System.out.println ("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 + 1;
        if (Side2 == Side3)
            triOut = triOut + 1;

        return triOut;
    }

    // Additional methods
    private static int getN()
    {
        // Method to get a number from the user
    }

    // Method to check if a triangle is valid
    private static boolean isValidTriangle (int Side1, int Side2, int Side3)
    {
        // Method to check if the input is valid
    }

    // Method to compare sides
    private static boolean areSidesEqual (int Side1, int Side2, int Side3)
    {
        // Method to check if any two sides are equal
    }
}

// Example usage
trityp.main(null);
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 getN ()
{
    int inputInt = 1;
    BufferedReader in = new BufferedReader (new InputStreamReader
    (System.in));
    String inStr;
    try
    {
        inStr = in.readLine ();
        inputInt = Integer.parseInt(inStr);
    }
    catch (IOException e)
    {
        System.out.println ("Could not read input, choosing 1.");
    }
    catch (NumberFormatException e)
    {
        System.out.println ("Entry must be a number, choosing 1.");
    }
    return (inputInt);
}  // end getN

}  // end trityp class
Appendix E - Relation

/**
 * @author Andras Marki
 */
public class Relation {
    // <
    public boolean less(int a, int b) {
        return a < b;
    }

    // >
    public boolean greater(int a, int b) {
        return a > b;
    }

    // <=
    public boolean lessEven(int a, int b) {
        return a <= b;
    }

    // >=
    public boolean greaterEven(int a, int b) {
        return a >= b;
    }

    // ==
    public boolean even(int a, int b) {
        return a == b;
    }

    // !=
    public boolean notEven(int a, int b) {
        return a != b;
    }

    // true
    public boolean T(int a, int b) {
        return true;
    }

    // false
    public boolean F(int a, int b) {
        return false;
    }
}
## Appendix F - Mutation testing results for Cal

<table>
<thead>
<tr>
<th>Test case number</th>
<th>Input</th>
<th>Interpretation of input</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1,1,1,1,1</td>
<td>Jan 1, Jan 1, Year 1</td>
</tr>
<tr>
<td>1</td>
<td>2,1,1,1,1</td>
<td>Feb 1, Jan 1, Year 1</td>
</tr>
<tr>
<td>2</td>
<td>1,1,3,1,400</td>
<td>Jan 1, Mar 1, Year 400</td>
</tr>
<tr>
<td>3</td>
<td>1,1,12,30,100</td>
<td>Jan 1, Dec 30, Year 100</td>
</tr>
<tr>
<td>4</td>
<td>1,1,3,29,4</td>
<td>Jan 1, Mar 29, Year 4</td>
</tr>
</tbody>
</table>

Total
Cal_1 is killed by tests 0 2 3 4
Cal_2 is killed by tests 2 3 4
Cal_3 is killed by tests 0 1
Cal_4 is killed by tests 1
Cal_5 is killed by tests 0 1 2 3 4
Cal_6 is killed by tests 1 2 3 4
Cal_7 is killed by tests 0
Cal_8 is alive
Cal_9 is killed by tests 2 4
Cal_10 is killed by tests 1
Cal_11 is killed by tests 1 2 4
Cal_12 is killed by tests 1 2 4
Cal_13 is killed by tests 2 4
Cal_14 is killed by tests 1
Cal_15 is killed by tests 3 4
Cal_16 is killed by tests 4
Cal_17 is killed by tests 3
Cal_18 is alive
Cal_19 is killed by tests 3 4
Cal_20 is killed by tests 4
Cal_21 is killed by tests 3
Cal_22 is alive
Cal_23 is killed by tests 2
Cal_24 is killed by tests 3
Cal_25 is killed by tests 2 3
Cal_26 is killed by tests 2 3
Cal_27 is killed by tests 2
Cal_28 is killed by tests 3
Cal_29 is killed by tests 1 2 3 4
Cal_30 is killed by tests 1 2 3 4
Cal_31 is killed by tests 2 3 4
Cal_32 is killed by tests 3
Cal_33 is killed by tests 1 2 3 4
Cal_34 is killed by tests 1 2 3 4

Total number of tests: 5
Total number of mutants: 34
Total killed: 31
Kill ratio: 91%
BUILD SUCCESSFUL (total time: 5 seconds)

Analysis shows that Cal_8, Cal_18 and Cal_22 are equivalent with the original software and cannot be killed, as it is shown in Appendix K.
Appendix G - Mutation testing results for CountNatural

<table>
<thead>
<tr>
<th>Test case number</th>
<th>Input (the number of natural number is counted by the program)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[1,2,3]</td>
</tr>
<tr>
<td>1</td>
<td>[1,1]</td>
</tr>
<tr>
<td>2</td>
<td>[-1,1,-1,-1]</td>
</tr>
<tr>
<td>3</td>
<td>[0]</td>
</tr>
</tbody>
</table>

CountNatural_1 is killed by tests 0 1 2 3
CountNatural_2 is killed by tests 0 1 2 3
CountNatural_3 is killed by tests 0 1 2 3
CountNatural_4 is killed by tests 0 1 2 3
CountNatural_5 is alive
CountNatural_6 is killed by tests 3
CountNatural_7 is killed by tests 0 1 2 3
CountNatural_8 is killed by tests 0 1 2 3
CountNatural_9 is killed by tests 0 1 2 3
CountNatural_10 is killed by tests 2 3
CountNatural_11 is killed by tests 2
CountNatural_12 is killed by tests 0 1 2 3

Total number of tests: 4
Total number of mutants: 12
Total killed: 11
Kill ratio: 91%
BUILD SUCCESSFUL (total time: 0 seconds)

Analysis shows that CountNatural_5 is equivalent with the original software and cannot be killed, as it is shown in Appendix L.
## Appendix H - Mutation testing results for Relation

<table>
<thead>
<tr>
<th>Test case number</th>
<th>Input (the two number are compared)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0,1</td>
</tr>
<tr>
<td>1</td>
<td>1,0</td>
</tr>
<tr>
<td>2</td>
<td>0,0</td>
</tr>
</tbody>
</table>

Total
- Relation_1 is killed by tests 0 1
- Relation_2 is killed by tests 0 1 2
- Relation_3 is killed by tests 2
- Relation_4 is killed by tests 0 2
- Relation_5 is killed by tests 1
- Relation_6 is killed by tests 1 2
- Relation_7 is killed by tests 0
- Relation_8 is killed by tests 2
- Relation_9 is killed by tests 0 1
- Relation_10 is killed by tests 0 1 2
- Relation_11 is killed by tests 1 2
- Relation_12 is killed by tests 0
- Relation_13 is killed by tests 0 2
- Relation_14 is killed by tests 1
- Relation_15 is killed by tests 0 1 2
- Relation_16 is killed by tests 0 1
- Relation_17 is killed by tests 2
- Relation_18 is killed by tests 0
- Relation_19 is killed by tests 1 2
- Relation_20 is killed by tests 1
- Relation_21 is killed by tests 0 2
- Relation_22 is killed by tests 2
- Relation_23 is killed by tests 0 1 2
- Relation_24 is killed by tests 0 1
- Relation_25 is killed by tests 1
- Relation_26 is killed by tests 0 2
- Relation_27 is killed by tests 0
- Relation_28 is killed by tests 1 2
- Relation_29 is killed by tests 1 2
- Relation_30 is killed by tests 1
- Relation_31 is killed by tests 0 2
- Relation_32 is killed by tests 0
- Relation_33 is killed by tests 0 1 2
- Relation_34 is killed by tests 0 1
- Relation_35 is killed by tests 2
- Relation_36 is killed by tests 0
- Relation_37 is killed by tests 0 2
- Relation_38 is killed by tests 1
- Relation_39 is killed by tests 1 2
- Relation_40 is killed by tests 0 1 2
- Relation_41 is killed by tests 2
- Relation_42 is killed by tests 0 1

Total number of tests: 3
Total number of mutants: 42
Total killed: 42
Kill ratio : 100%
BUILD SUCCESSFUL (total time: 1 second)
Appendix I - Mutation testing results for TestPat

<table>
<thead>
<tr>
<th>Test case number</th>
<th>Input (Controls if the second string is part of the first string)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>“aaa”, “a”</td>
</tr>
<tr>
<td>1</td>
<td>“aaa”, “b”</td>
</tr>
<tr>
<td>2</td>
<td>“a”, “aa”</td>
</tr>
<tr>
<td>3</td>
<td>“aaa”, “ab”</td>
</tr>
<tr>
<td>4</td>
<td>“aabaa”, “abaa”</td>
</tr>
<tr>
<td>5</td>
<td>“baa”, “aaaa”</td>
</tr>
<tr>
<td>6</td>
<td>“b”, “a”</td>
</tr>
</tbody>
</table>

Total:
TestPat_1 is killed by tests 0 5
TestPat_2 is killed by tests 0 5
TestPat_3 is killed by tests 0 3 5 6
TestPat_4 is killed by tests 1 2 3 4 7
TestPat_5 is killed by tests 0 3 5
TestPat_6 is alive
TestPat_7 is killed by tests 1 2 3 4 6 7
TestPat_8 is killed by tests 0 5
TestPat_9 is killed by tests 0 5 7
TestPat_10 is killed by tests 7
TestPat_11 is killed by tests 0 1 2 5
TestPat_12 is killed by tests 1 2
TestPat_13 is killed by tests 0 1 2 5 7
TestPat_14 is killed by tests 1 2 7
TestPat_15 is killed by tests 0 5
TestPat_16 is killed by tests 4 5
TestPat_17 is killed by tests 0 4 5
TestPat_18 is killed by tests 0 5
TestPat_19 is killed by tests 0 4 5
TestPat_20 is alive
TestPat_21 is killed by tests 0 5
TestPat_22 is killed by tests 0 5
TestPat_23 is killed by tests 4
TestPat_24 is killed by tests 4 5
TestPat_25 is alive
TestPat_26 is killed by tests 5
TestPat_27 is killed by tests 4 5
TestPat_28 is killed by tests 5
TestPat_29 is killed by tests 4 5

Total number of tests: 8
Total number of mutants: 28
Total killed: 25
Kill ratio : 89%
BUILD SUCCESSFUL (total time: 2 seconds)

Analysis shows that TestPat_6, TestPat_20 and TestPat_25 are equivalent with the original software and cannot be killed, as it is shown in Appendix M.
## Appendix J - Mutation testing results for TriTyp

<table>
<thead>
<tr>
<th>Test case number</th>
<th>Input (as length of triangle side 1, side 2 and side 3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2,3,4</td>
</tr>
<tr>
<td>1</td>
<td>2,4,3</td>
</tr>
<tr>
<td>2</td>
<td>3,2,4</td>
</tr>
<tr>
<td>3</td>
<td>3,4,2</td>
</tr>
<tr>
<td>4</td>
<td>4,2,3</td>
</tr>
<tr>
<td>5</td>
<td>4,3,2</td>
</tr>
<tr>
<td>6</td>
<td>2,3,5</td>
</tr>
<tr>
<td>7</td>
<td>2,5,3</td>
</tr>
<tr>
<td>8</td>
<td>3,2,5</td>
</tr>
<tr>
<td>9</td>
<td>3,5,2</td>
</tr>
<tr>
<td>10</td>
<td>5,2,3</td>
</tr>
<tr>
<td>11</td>
<td>5,3,2</td>
</tr>
<tr>
<td>12</td>
<td>2,3,6</td>
</tr>
<tr>
<td>13</td>
<td>6,3,2</td>
</tr>
<tr>
<td>14</td>
<td>2,6,3</td>
</tr>
<tr>
<td>15</td>
<td>3,2,2</td>
</tr>
<tr>
<td>16</td>
<td>2,3,2</td>
</tr>
<tr>
<td>17</td>
<td>2,2,3</td>
</tr>
<tr>
<td>18</td>
<td>3,3,2</td>
</tr>
<tr>
<td>19</td>
<td>2,3,3</td>
</tr>
<tr>
<td>20</td>
<td>3,2,3</td>
</tr>
<tr>
<td>21</td>
<td>4,2,2</td>
</tr>
<tr>
<td>22</td>
<td>2,4,2</td>
</tr>
<tr>
<td>23</td>
<td>2,2,4</td>
</tr>
<tr>
<td>24</td>
<td>5,2,2</td>
</tr>
<tr>
<td>25</td>
<td>2,5,2</td>
</tr>
<tr>
<td>26</td>
<td>2,2,5</td>
</tr>
<tr>
<td>27</td>
<td>2,2,2</td>
</tr>
<tr>
<td>28</td>
<td>0,2,2</td>
</tr>
<tr>
<td>29</td>
<td>-2,2,2</td>
</tr>
<tr>
<td>30</td>
<td>2,0,2</td>
</tr>
<tr>
<td>31</td>
<td>2,2,0</td>
</tr>
<tr>
<td>32</td>
<td>2,2,2</td>
</tr>
<tr>
<td>33</td>
<td>2,2,-2</td>
</tr>
</tbody>
</table>

Total

- TriTyp_1 is killed by tests 0 1 2 3 4 5 15 16 17 18 19 20 27 28 29
- TriTyp_2 is killed by tests 0 1 2 3 4 5 15 16 17 18 19 20 27 29
- TriTyp_3 is killed by tests 28
- TriTyp_4 is killed by tests 29
- TriTyp_5 is killed by tests 0 1 2 3 4 5 15 16 17 18 19 20 27 28
- TriTyp_6 is killed by tests 0 1 2 3 4 5 15 16 17 18 19 20 27
- TriTyp_7 is killed by tests 28 29
- TriTyp_8 is killed by tests 0 1 2 3 4 5 15 16 17 18 19 20 27 30 32
- TriTyp_9 is killed by tests 0 1 2 3 4 5 15 16 17 18 19 20 27 32
TriTyp_10 is killed by tests 30
TriTyp_11 is killed by tests 32
TriTyp_12 is killed by tests 0 1 2 3 4 5 15 16 17 18 19 20 27 30
TriTyp_13 is killed by tests 0 1 2 3 4 5 15 16 17 18 19 20 27
TriTyp_14 is killed by tests 30 32
TriTyp_15 is killed by tests 0 1 2 3 4 5 15 16 17 18 19 20 27 31 33
TriTyp_16 is killed by tests 0 1 2 3 4 5 15 16 17 18 19 20 27 33
TriTyp_17 is killed by tests 31
TriTyp_18 is killed by tests 33
TriTyp_19 is killed by tests 0 1 2 3 4 5 15 16 17 18 19 20 27 31
TriTyp_20 is killed by tests 0 1 2 3 4 5 15 16 17 18 19 20 27
TriTyp_21 is killed by tests 31 33
TriTyp_22 is killed by tests 2 4 5 10 11 13 15 17 18 21 24
TriTyp_23 is killed by tests 2 4 5 10 11 13 15 21 24
TriTyp_24 is killed by tests 0 1 3 7 9 14 17 18 19 22 25
TriTyp_25 is killed by tests 0 1 3 7 9 14 19 22 25
TriTyp_26 is killed by tests 0 1 2 3 4 5 7 9 10 11 13 14 15 17 18 19 21 22 24 25
TriTyp_27 is killed by tests 0 1 2 3 4 5 7 9 10 11 13 14 15 19 21 22 24 25
TriTyp_28 is killed by tests 17 18
TriTyp_29 is killed by tests 3 4 5 10 11 13 15 16 20 21 24
TriTyp_30 is killed by tests 3 4 5 10 11 13 15 21 24
TriTyp_31 is killed by tests 0 1 2 6 8 12 16 19 20 23 26
TriTyp_32 is killed by tests 0 1 2 6 8 12 19 23 26
TriTyp_33 is killed by tests 0 1 2 3 4 5 6 8 10 11 12 13 15 16 19 20 21 23 24 26
TriTyp_34 is killed by tests 0 1 2 3 4 5 6 8 10 11 12 13 15 19 21 23 24 26
TriTyp_35 is killed by tests 16 20
TriTyp_36 is killed by tests 1 3 5 7 9 14 15 16 18 19 22 25 27
TriTyp_37 is killed by tests 1 3 5 7 9 14 16 18 22 25
TriTyp_38 is killed by tests 0 2 4 6 8 12 15 17 19 20 23 26 27
TriTyp_39 is killed by tests 0 2 4 6 8 12 17 20 23 26
TriTyp_40 is killed by tests 0 1 2 3 4 5 6 7 8 9 12 14 15 16 17 18 19 20 21 22 23 25 26
TriTyp_41 is killed by tests 0 1 2 3 4 5 6 7 8 9 12 14 16 17 18 20 22 23 25 26
TriTyp_42 is killed by tests 15 19 27
TriTyp_43 is killed by tests 0 1 2 3 4 5 15 16 17 18 19 20 27
TriTyp_44 is killed by tests 15 16 17 18 19 20 27
TriTyp_45 is killed by tests 0 1 2 3 4 5
TriTyp_46 is alive
TriTyp_47 is killed by tests 0 1 2 3 4 5 15 16 17 18 19 20 27
TriTyp_48 is killed by tests 15 16 17 18 19 20 27
TriTyp_49 is killed by tests 0 1 2 3 4 5
TriTyp_50 is killed by tests 0 1 2 3 4 5 6 8 12
TriTyp_51 is killed by tests 0 1 2 3 4 5 12
TriTyp_52 is killed by tests 6 8
TriTyp_53 is killed by tests 12
TriTyp_54 is killed by tests 0 1 2 3 4 5 6 8
TriTyp_55 is killed by tests 0 1 2 3 4 5
TriTyp_56 is killed by tests 6 8 12
TriTyp_57 is killed by tests 0 1 2 3 4 5 10 11 13
TriTyp_58 is killed by tests 0 1 2 3 4 5 13
TriTyp_59 is killed by tests 10 11
TriTyp_60 is killed by tests 13
TriTyp_61 is killed by tests 0 1 2 3 4 5 10 11
TriTyp_62 is killed by tests 0 1 2 3 4 5
TriTyp_63 is killed by tests 10 11 13
TriTyp_64 is killed by tests 0 1 2 3 4 5 7 9 14
TriTyp_65 is killed by tests 0 1 2 3 4 5 14
TriTyp_66 is killed by tests 7 9

76
TriTyp_67 is killed by tests 14
TriTyp_68 is killed by tests 0 1 2 3 4 5 7 9
TriTyp_69 is killed by tests 0 1 2 3 4 5
TriTyp_70 is killed by tests 7 9 14
TriTyp_71 is killed by tests 15 19 21 24
TriTyp_72 is killed by tests 16 17 18 20 22 23 25 26 27
TriTyp_73 is killed by tests 15 16 17 18 19 20 21 22 23 24 25 26 27
TriTyp_74 is killed by tests 15 19 21 24 27
TriTyp_75 is killed by tests 16 17 18 20 22 23 25 26
TriTyp_76 is killed by tests 15 16 17 18 19 20 21 22 23 24 25 26 27
TriTyp_77 is killed by tests 27
TriTyp_78 is killed by tests 17 18 21 22 24 25
TriTyp_79 is killed by tests 21 22 24 25
TriTyp_80 is killed by tests 17 18
TriTyp_81 is alive
TriTyp_82 is killed by tests 17 18 21 22 24 25
TriTyp_83 is killed by tests 21 22 24 25
TriTyp_84 is killed by tests 17 18
TriTyp_85 is killed by tests 23
TriTyp_86 is killed by tests 17 18 26
TriTyp_87 is killed by tests 17 18 23 26
TriTyp_88 is killed by tests 17 18 23
TriTyp_89 is killed by tests 26
TriTyp_90 is killed by tests 23 26
TriTyp_91 is killed by tests 17 18
TriTyp_92 is killed by tests 16 20 21 24
TriTyp_93 is killed by tests 21 24
TriTyp_94 is killed by tests 16 20 23 26
TriTyp_95 is killed by tests 23 26
TriTyp_96 is killed by tests 16 20 21 23 24 26
TriTyp_97 is killed by tests 21 23 24 26
TriTyp_98 is killed by tests 16 20
TriTyp_99 is killed by tests 22
TriTyp_100 is killed by tests 16 20 25
TriTyp_101 is killed by tests 16 20 22 25
TriTyp_102 is killed by tests 16 20 22
TriTyp_103 is killed by tests 25
TriTyp_104 is killed by tests 22 25
TriTyp_105 is killed by tests 16 20
TriTyp_106 is killed by tests 15 19
TriTyp_107 is alive
TriTyp_108 is killed by tests 15 19 22 23 25 26
TriTyp_109 is killed by tests 22 23 25 26
TriTyp_110 is killed by tests 15 19 22 23 25 26
TriTyp_111 is killed by tests 22 23 25 26
TriTyp_112 is killed by tests 15 19
TriTyp_113 is killed by tests 21
TriTyp_114 is killed by tests 15 19 24
TriTyp_115 is killed by tests 15 19 21 24
TriTyp_116 is killed by tests 15 19 21
TriTyp_117 is killed by tests 24
TriTyp_118 is killed by tests 21 24
TriTyp_119 is killed by tests 15 19

Total number of tests: 34
Total number of mutants: 119
Total killed: 116
Kill ratio : 97%
BUILD SUCCESSFUL (total time: 8 seconds)
Analysis shows that TriTyp_46 Trityp_81 and TriTyp_107 are equivalent with the original software and cannot be killed, as it is shown in Appendix N.
Appendix K - Analysis on the mutants alive: Cal

This appendix presents an analysis on the mutants alive on the program Cal. Below is the original program presented, with the original row which has been mutated marked with "ORIGINAL", the mutated rows marked with ROR_8, ROR_18 respective ROR_22.

It should be noted that the value m4 comes from a year % 4 operation, so m4 can have the values \{0,1,2,3\}. ROR_8 replaces the condition m4!=0 with m4>0. It should be noted that if m4 has a value of 0, both m4 != 0 and m4 > 0 evaluate to false, and for the values \{1,2,3\}, both expressions evaluate to true. Therefore, ROR_8 is an equivalent mutant.

The variable m100 gets its value from a year % 100 operation, thus it can have the values \{0, 1, 2, ..., 98, 99\}. ROR_18 replaces m100 == 0 with m100 <= 0. As m100 cannot have a negative value, the expressions m100 <= 0 and m100 == 0 are equivalent: Therefore, ROR_18 is an equivalent mutant.

The variable m400 comes from year % 400, and can therefore have the values of \{0, 1, ..., 399\}. ROR_22 replaces m400 != 0 with m400 > 0. Similarly to the case of m4, m400 is also equivalent therefore.

Thus all the three mutants that could not be killed with the test suite are equivalent mutants.

```java
public int cal( int month1, int day1, int month2, int day2, int year )
{
    int numDays;
    if (month2 == month1) {
        numDays = day2 - day1;
    } else {
        int[] daysIn = { 0, 31, 0, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31, 31, 30, 31, 31, 30, 31 };
        int m4 = year % 4;
        int m100 = year % 100;
        int m400 = year % 400;
        ORIGINAL if (m4 != 0 || m100 == 0 && m400 != 0) {
            daysIn[2] = 28;
        } else {
            daysIn[2] = 29;
        }
    }
    numDays = day2 + (daysIn[month1] - day1);
    for (int i = month1 + 1; i <= month2 - 1; i++) {
        numDays = daysIn[i] + numDays;
    }
    return numDays;
}
```
public int getN()
{
    int inputInt = 1;
    java.io.BufferedReader in = new java.io.BufferedReader( new
    java.io.InputStreamReader( System.in ) );
    java.lang.String inStr;
    try {
        inStr = in.readLine();
        inputInt = Integer.parseInt( inStr );
    } catch ( java.io.IOException e ){
        System.out.println( "Could not read input, choosing 1." );
    } catch ( java.lang.NumberFormatException e ){
        System.out.println( "Entry must be a number, choosing 1." );
    }
    return inputInt;
}
Appendix L - Analysis on the mutants alive: CountNatural

This appendix presents an analysis on the mutants of the program CountNatural alive after running the test suite. Below is the original program presented, with the original rows which contained the mutant operator marked with the prefix ORIG and a number indicating which mutant it originated. The mutants are marked with the prefix ROR_ and italic font. The faults introduced to the given row by the mutation operator are marked with bold.

The mutation operator changes the operator < to != in this case in a for loop: A difference could be detected if i could be larger than x.length, however this can't happen as the loop terminates at the same time in both cases. != is equivalent to < in this case, therefore this mutant is equivalent.

TODO
// This is a mutant program.
// Author : ysmal

package simpletest;

class CountPositive implements simpletest.Mutant {

    public int countPositive(int[] x) {
        int count = 0;
        ORIG for (int i = 0; i < x.length; i++) {
            ROR_5 for (int i = 0; i != x.length; i++) {
                if (x[i] >= 0) {
                    count++;
                }
            }
            return count;
        }
    }
}
Appendix M - Analysis on the mutants alive:
TestPat

This appendix presents an analysis on the mutants alive after running the tests set. Below is the original program presented, with the original rows which contained the mutant operator marked with the prefix ORIG and a number indicating which mutant it originated. The mutants are marked with the prefix ROR_ and italic font. The faults introduced to the given row by the mutation operator are marked with bold.

It can be noted that the “iSub + patternLen - 1 < subjectLen” and the “iSub + patternLen - 1 != subjectLen” expressions are true from the beginning in the while loop and become false at the same time. The original expression would be different if (iSub + patternLen - 1) could become bigger than subjectLen, but this cannot happen. Therefore ROR_6 is equivalent. ROR_20 and ROR_25 relate in a same way to their originals (where ROR_25 changes != to < and not the other way around). Therefore, each of these three mutants are equivalent to the original.

// This is a mutant program.
// Author : ysm

package testpat;

class TestPat implements testpat.Mutant
{
    public int pat( char[] subject, char[] pattern )
    {
        final int NOTFOUND = -1;
        int iSub = 0;
        int rtnIndex = NOTFOUND;
        boolean isPat = false;
        int subjectLen = subject.length;
        int patternLen = pattern.length;

        ORIG6   while (isPat == false && iSub + patternLen - 1 < subjectLen)
        ROR_6   while (isPat == false && iSub + patternLen - 1 != subjectLen)
        {
            if (subject[iSub] == pattern[0])
            {
                rtnIndex = iSub;
                isPat = true;
            }
            ORIG20   for (int iP = 1; iP < patternLen; iP++)
            ROR_20   for (int iP = 1; iP != patternLen; iP++)
            ORIG25   if (subject[iSub + iP] != pattern[iP])
            ROR_25   if (subject[iSub + iP] < pattern[iP])
            {
                rtnIndex = NOTFOUND;
                isPat = false;
                break;
            }
        }
        isSub++;
    }
    return rtnIndex;
}
Appendix N - Analysis on the mutants alive: TriTyp

This appendix presents an analysis on the mutants of the program TriTyp alive after running the test suite. Below is the original program presented, with the original rows which contained the mutant operator marked with the prefix ORIG and a number indicating which mutant it originated. The mutants are marked with the prefix ROR_ and italic font. The faults introduced to the given row by the mutation operator are marked with bold.

ROR_46 needs triOut to be less than 0 to infect, however triOut is set to 0 and all assignments after that are incremental so triOut is at least 0 at that point. Thus, ROR_46 is equivalent.

ROR_81 needs the variable triOut to be less than 1 to give a different execution path for the program; however that can’t happen as triOut is at least 1 at that point. ROR_81 is therefore equivalent. ROR_107 needs triOut to be greater than 3 at that point to infect, which cannot happen as a previous if-statement sets TriOut ot 3 if it is larger than 3. ROR_107 is also equivalent.

Thus all three remaining mutants alive are marked as equal.

```
// This is a mutant program.
// Author: ysma
package trityp;

import java.io.*;

class TriTyp implements trityp.Mutant {
    public int Triang(int Side1, int Side2, int Side3) {
        int triOut;
        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) {
            if (triOut <= 0) {
                if (Side1 + Side2 <= Side3 || Side2 + Side3 <= Side1 || Side1 + Side3 <= Side2) {
                    triOut = 4;
                } else {
                    triOut = 1;
                }
            }
            return triOut;
        }
        if (triOut > 3) {
            triOut = 3;
        }
        if (triOut == 1 && Side1 + Side2 > Side3) {
            triOut = 2;
        }
    }
```
if (triOut == 2 && Side1 + Side3 > Side2) {
    triOut = 2;
} else {
    if (triOut == 3 && Side2 + Side3 > Side1) {
        if (triOut >= 3 && Side2 + Side3 > Side1) {
            triOut = 2;
        } else {
            triOut = 4;
        }
    }
}
return triOut;

public int getN() {
    int inputInt = 1;
    java.io.BufferedReader in = new java.io.BufferedReader( new java.io.InputStreamReader(System.in) );
    java.lang.String inStr;
    try {
        inStr = in.readLine();
        inputInt = Integer.parseInt( inStr );
    } catch (java.io.IOException e) {
        System.out.println( "Could not read input, choosing 1." );
    } catch (java.lang.NumberFormatException e) {
        System.out.println( "Entry must be a number, choosing 1." );
    }
    return inputInt;
}
Appendix O  -  Cal: compatible with test tool

Cal.java

// This is a mutant program.
// Author : ysma

class Cal implements cal.Mutant {

    public int cal(int month1, int day1, int month2, int day2, int year) {
        int numDays;
        if (month2 == month1) {
            numDays = day2 - day1;
        } else {
            int[] daysIn = {0, 31, 0, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
            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;
            }
            numDays = day2 + (daysIn[month1] - day1);
            for (int i = month1 + 1; i <= month2 - 1; i++) {
                numDays = daysIn[i] + numDays;
            }
        }
        return numDays;
    }

    public int getN() {
        int inputInt = 1;
        java.io.BufferedReader in = new java.io.BufferedReader( new java.io.InputStreamReader( System.in ) );
        java.lang.String inStr;
        try {
            inStr = in.readLine();
            inputInt = Integer.parseInt( inStr );
        } catch (java.io.IOException e) {
            System.out.println( "Could not read input, choosing 1." );
        } catch (java.lang.NumberFormatException e) {
            System.out.println( "Entry must be a number, choosing 1." );
        }
        return inputInt;
    }

}
Mutant.java

/*
 * Needed for the test driver
 */

package cal;

/**
 * @author mard
 */

public interface Mutant {
    abstract int cal( int month1, int day1, int month2, int day2, int year );
    abstract int getN();
}
Appendix P  -  CountNatural: compatible with test tool

CountPositive.java
// This is a mutant program.
// Author : ysma

package simpletest;

class CountPositive implements simpletest.Mutant
{
    public int countPositive( int[] x )
    {
        int count = 0;
        for (int i = 0; i < x.length; i++) {
            if (x[i] >= 0) {
                count++;
            }
        }
        return count;
    }
}

Mutant.java
package simpletest;

/**
 * @author mard
 */
public interface Mutant {
    int countPositive(int[] x);
}
Appendix Q - Relation: compatible with test tool

Relation.java
// This is a mutant program.
// Author : ysma

class simpletest;

public class Relation implements Mutant {
    public boolean less( int a, int b ) {
        return a < b;
    }
    public boolean greater( int a, int b ) {
        return a > b;
    }
    public boolean lessEven( int a, int b ) {
        return a <= b;
    }
    public boolean greaterEven( int a, int b ) {
        return a >= b;
    }
    public boolean even( int a, int b ) {
        return a == b;
    }
    public boolean notEven( int a, int b ) {
        return a != b;
    }
    public boolean T( int a, int b ) {
        return true;
    }
    public boolean F( int a, int b ) {
        return false;
    }
}
package simpletest;

public interface Mutant {
    public boolean less(int a, int b);
    // >
    public boolean greater(int a, int b);
    // <=
    public boolean lessEven(int a, int b);
    // >=
    public boolean greaterEven(int a, int b);
    // ==
    public boolean even(int a, int b);
    // !=
    public boolean notEven(int a, int b);
    // true
    public boolean T(int a, int b);
    // false
    public boolean F(int a, int b);
}
Appendix R  -  TestPat: compatible with test tool

TestPat.java

// This is a mutant program.
// Author: ysma

package testpat;

class TestPat implements testpat.Mutant
{
    public int pat(char[] subject, char[] pattern)
    {
        final int NOTFOUND = -1;
        int iSub = 0;
        int rtnIndex = NOTFOUND;
        boolean isPat = false;
        int subjectLen = subject.length;
        int patternLen = pattern.length;
        while (isPat == false && iSub + patternLen - 1 < subjectLen) {
            if (subject[iSub] == pattern[0]) {
                rtnIndex = iSub;
                isPat = true;
                for (int iPat = 1; iPat < patternLen; iPat++) {
                    if (subject[iSub + iPat] != pattern[iPat]) {
                        rtnIndex = NOTFOUND;
                        isPat = false;
                        break;
                    }
                }
            }
            iSub++;
        }
        return rtnIndex;
    }
}

Mutant.java

/**
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */

package testpat;

/**
 * @author mard
 */
public interface Mutant {
    public int pat(char[] subject, char[] pattern);
}
Appendix S - TriTyp: Compatible with test tool

Trityp.java
// This is a mutant program.
// Author : ysma

package trityp;

import java.io.*;

class TriTyp implements trityp.Mutant {
    public int Triang(int Side1, int Side2, int Side3) {
        int triOut;
        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) {
            if (Side1 + Side2 <= Side3 || Side2 + Side3 <= Side1 || Side1 + Side3 <= Side2) {
                triOut = 4;
            } else {
                triOut = 1;
            }
            return triOut;
        }
        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;
}

public int getN()
{
    int inputInt = 1;
    java.io.BufferedReader in = new java.io.BufferedReader( new
    java.io.InputStreamReader( System.in ) );
    java.lang.String inStr;
    try {
        inStr = in.readLine();
        inputInt = Integer.parseInt( inStr );
    } catch ( java.io.IOException e ) {
        System.out.println( "Could not read input, choosing 1." );
    } catch ( java.lang.NumberFormatException e ) {
        System.out.println( "Entry must be a number, choosing 1." );
    }
    return inputInt;
}

Mutant.java

/*
 * Needed for the test driver
 */

package trityp;

/**
 *  @author mard
 */

public interface Mutant {
    abstract int Triang (int Side1, int Side2, int Side3);

    abstract int getN ();
}
Appendix T - TestPat: trace aaa b

The results after the tests with the input aaa b are below, which means that the program calculates if the string “b” is part of the string “aaa”. An empty result means crash, which also triggers the "Dead mutant with exception" printout.

<table>
<thead>
<tr>
<th>Testing with the input aaa b</th>
<th>result for class testpat.TestPat_original</th>
<th>result: -1</th>
</tr>
</thead>
<tbody>
<tr>
<td>result for class testpat.TestPat_1</td>
<td>result: -1</td>
<td></td>
</tr>
<tr>
<td>result for class testpat.TestPat_2</td>
<td>result: -1</td>
<td></td>
</tr>
<tr>
<td>result for class testpat.TestPat_3</td>
<td>result: -1</td>
<td></td>
</tr>
<tr>
<td>result for class testpat.TestPat_4</td>
<td>result:</td>
<td></td>
</tr>
<tr>
<td>result for class testpat.TestPat_5</td>
<td>result: -1</td>
<td></td>
</tr>
<tr>
<td>result for class testpat.TestPat_6</td>
<td>result: -1</td>
<td></td>
</tr>
<tr>
<td>result for class testpat.TestPat_7</td>
<td>result:</td>
<td></td>
</tr>
<tr>
<td>result for class testpat.TestPat_8</td>
<td>result: -1</td>
<td></td>
</tr>
<tr>
<td>result for class testpat.TestPat_9</td>
<td>result: -1</td>
<td></td>
</tr>
<tr>
<td>result for class testpat.TestPat_10</td>
<td>result: -1</td>
<td></td>
</tr>
<tr>
<td>result for class testpat.TestPat_11</td>
<td>result: 0</td>
<td></td>
</tr>
<tr>
<td>result for class testpat.TestPat_12</td>
<td>result: 0</td>
<td></td>
</tr>
<tr>
<td>result for class testpat.TestPat_13</td>
<td>result:</td>
<td></td>
</tr>
<tr>
<td>result for class testpat.TestPat_14</td>
<td>result: 0</td>
<td></td>
</tr>
<tr>
<td>result for class testpat.TestPat_15</td>
<td>result: -1</td>
<td></td>
</tr>
<tr>
<td>result for class testpat.TestPat_16</td>
<td>result: -1</td>
<td></td>
</tr>
<tr>
<td>result for class testpat.TestPat_17</td>
<td>result: -1</td>
<td></td>
</tr>
<tr>
<td>result for class testpat.TestPat_18</td>
<td>result: -1</td>
<td></td>
</tr>
<tr>
<td>result for class testpat.TestPat_19</td>
<td>result: -1</td>
<td></td>
</tr>
<tr>
<td>result for class testpat.TestPat_20</td>
<td>result: -1</td>
<td></td>
</tr>
<tr>
<td>result for class testpat.TestPat_21</td>
<td>result: -1</td>
<td></td>
</tr>
<tr>
<td>result for class testpat.TestPat_22</td>
<td>result:</td>
<td></td>
</tr>
<tr>
<td>result for class testpat.TestPat_23</td>
<td>result: -1</td>
<td></td>
</tr>
<tr>
<td>result for class testpat.TestPat_24</td>
<td>result: -1</td>
<td></td>
</tr>
<tr>
<td>result for class testpat.TestPat_25</td>
<td>result: -1</td>
<td></td>
</tr>
<tr>
<td>result for class testpat.TestPat_26</td>
<td>result: -1</td>
<td></td>
</tr>
<tr>
<td>result for class testpat.TestPat_27</td>
<td>result: -1</td>
<td></td>
</tr>
<tr>
<td>result for class testpat.TestPat_28</td>
<td>result: -1</td>
<td></td>
</tr>
<tr>
<td>result for class testpat.TestPat_29</td>
<td>result: -1</td>
<td></td>
</tr>
</tbody>
</table>
These are the results for the kill:

<table>
<thead>
<tr>
<th>TEST RESULT</th>
<th>row</th>
<th>original</th>
<th>mutant</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>TestPat_1</td>
<td>18</td>
<td>==</td>
<td>!=</td>
<td>Alive</td>
</tr>
<tr>
<td>TestPat_2</td>
<td>18</td>
<td>&lt;</td>
<td>&gt;</td>
<td>Alive</td>
</tr>
<tr>
<td>TestPat_3</td>
<td>18</td>
<td>&lt;</td>
<td>&gt;=</td>
<td>Alive</td>
</tr>
<tr>
<td>TestPat_4</td>
<td>18</td>
<td>&lt;</td>
<td>&lt;=</td>
<td>Dead</td>
</tr>
<tr>
<td>TestPat_5</td>
<td>18</td>
<td>&lt;</td>
<td>==</td>
<td>Alive</td>
</tr>
<tr>
<td>TestPat_6</td>
<td>18</td>
<td>&lt;</td>
<td>!=</td>
<td>Alive</td>
</tr>
<tr>
<td>TestPat_7</td>
<td>18</td>
<td>&lt;</td>
<td>TRUE</td>
<td>Dead</td>
</tr>
<tr>
<td>TestPat_8</td>
<td>18</td>
<td>&lt;</td>
<td>FALSE</td>
<td>Alive</td>
</tr>
<tr>
<td>TestPat_9</td>
<td>19</td>
<td>==</td>
<td>&gt;</td>
<td>Alive</td>
</tr>
<tr>
<td>TestPat_10</td>
<td>19</td>
<td>==</td>
<td>&gt;=</td>
<td>Alive</td>
</tr>
<tr>
<td>TestPat_11</td>
<td>19</td>
<td>==</td>
<td>&lt;</td>
<td>Dead</td>
</tr>
<tr>
<td>TestPat_12</td>
<td>19</td>
<td>==</td>
<td>&lt;=</td>
<td>Dead</td>
</tr>
<tr>
<td>TestPat_13</td>
<td>19</td>
<td>==</td>
<td>!=</td>
<td>Dead</td>
</tr>
<tr>
<td>TestPat_14</td>
<td>19</td>
<td>==</td>
<td>TRUE</td>
<td>Dead</td>
</tr>
<tr>
<td>TestPat_15</td>
<td>19</td>
<td>==</td>
<td>FALSE</td>
<td>Alive</td>
</tr>
<tr>
<td>TestPat_16</td>
<td>22</td>
<td>&lt;</td>
<td>&gt;</td>
<td>Alive</td>
</tr>
<tr>
<td>TestPat_17</td>
<td>22</td>
<td>&lt;</td>
<td>&gt;=</td>
<td>Alive</td>
</tr>
<tr>
<td>TestPat_18</td>
<td>22</td>
<td>&lt;</td>
<td>&lt;=</td>
<td>Alive</td>
</tr>
<tr>
<td>TestPat_19</td>
<td>22</td>
<td>&lt;</td>
<td>==</td>
<td>Alive</td>
</tr>
<tr>
<td>TestPat_20</td>
<td>22</td>
<td>&lt;</td>
<td>!=</td>
<td>Alive</td>
</tr>
<tr>
<td>TestPat_21</td>
<td>22</td>
<td>&lt;</td>
<td>TRUE</td>
<td>Alive</td>
</tr>
<tr>
<td>TestPat_23</td>
<td>23</td>
<td>!=</td>
<td>&gt;</td>
<td>Alive</td>
</tr>
<tr>
<td>TestPat_24</td>
<td>23</td>
<td>!=</td>
<td>&gt;=</td>
<td>Alive</td>
</tr>
<tr>
<td>TestPat_25</td>
<td>23</td>
<td>!=</td>
<td>&lt;</td>
<td>Alive</td>
</tr>
<tr>
<td>TestPat_26</td>
<td>23</td>
<td>!=</td>
<td>&lt;=</td>
<td>Alive</td>
</tr>
<tr>
<td>TestPat_27</td>
<td>23</td>
<td>!=</td>
<td>==</td>
<td>Alive</td>
</tr>
<tr>
<td>TestPat_28</td>
<td>23</td>
<td>!=</td>
<td>TRUE</td>
<td>Alive</td>
</tr>
<tr>
<td>TestPat_29</td>
<td>23</td>
<td>!=</td>
<td>FALSE</td>
<td>Alive</td>
</tr>
</tbody>
</table>
The results below show how the validation on the first claim fails. The results can be controlled by hand (and this was done and also with the original program using the inputs as well as mutating it manually to gather the results).

All operators are checked against three other operators accordingly to the proposed subsumption graphs. The lowest level operator is checked against all middle level (all which are in “subsumes” towards it). The middle level operators are checked against the lowest level operator (which they have “subsumed” relation towards) and the two top level ones which are “subsumes” towards the operators. The top level mutants are checked against the two subsumed middle level mutants as well as the lowest level mutants as an extra check, and have relations “subsumed” against them. So for example TRUE is dead but >= is alive, which is against the first claim. Such results which are not in line with the claim are marked.

<table>
<thead>
<tr>
<th>mutantID</th>
<th>Original operator</th>
<th>Mutant Operator 1</th>
<th>result</th>
<th>row</th>
<th>position</th>
<th>Related operator 1</th>
<th>op1 result</th>
<th>Relation to op1</th>
<th>Related operator 2</th>
<th>op2 result</th>
<th>Relation to operator 2</th>
<th>Relation to operator 3</th>
<th>Op3 result</th>
<th>Relation to operator 3</th>
<th>Subsumption</th>
</tr>
</thead>
<tbody>
<tr>
<td>TestPat_1</td>
<td>==</td>
<td>!=</td>
<td>live</td>
<td>18</td>
<td>21</td>
<td>TRUE</td>
<td>unknown subsumes</td>
<td>&lt;</td>
<td>unknown subsumes</td>
<td>&gt;</td>
<td>unknown subsumes</td>
<td>OK</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TestPat_2</td>
<td>&lt;</td>
<td>&gt;</td>
<td>live</td>
<td>18</td>
<td>55</td>
<td>FALSE</td>
<td>live subsumes</td>
<td>!&gt;</td>
<td>live subsumes</td>
<td>&gt;=</td>
<td>live subsumed</td>
<td>OK</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TestPat_3</td>
<td>&lt;</td>
<td>&gt;&gt;</td>
<td>live</td>
<td>18</td>
<td>55</td>
<td>&gt;</td>
<td>live subsumes</td>
<td>==</td>
<td>live subsumes</td>
<td>TRUE</td>
<td>dead subsumed</td>
<td>OK</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TestPat_4</td>
<td>&lt;</td>
<td>&lt;=</td>
<td>dead</td>
<td>18</td>
<td>55</td>
<td>&lt;=</td>
<td>live subsumed</td>
<td>TRUE</td>
<td>dead subsumed</td>
<td>&gt;=</td>
<td>live subsumed</td>
<td>not OK</td>
<td>OK</td>
<td>not OK</td>
<td>OK</td>
</tr>
<tr>
<td>TestPat_5</td>
<td>&lt;</td>
<td>&gt;=</td>
<td>live</td>
<td>18</td>
<td>55</td>
<td>FALSE</td>
<td>live subsumes</td>
<td>&lt;=</td>
<td>dead subsumes</td>
<td>&gt;=</td>
<td>live subsumed</td>
<td>OK</td>
<td>OK</td>
<td>not OK</td>
<td>OK</td>
</tr>
<tr>
<td>TestPat_6</td>
<td>&lt;</td>
<td>!=</td>
<td>live</td>
<td>18</td>
<td>55</td>
<td>&gt;</td>
<td>live subsumed</td>
<td>TRUE</td>
<td>dead subsumed</td>
<td>&gt;=</td>
<td>live subsumed</td>
<td>OK</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TestPat_7</td>
<td>&lt;</td>
<td>TRUE</td>
<td>dead</td>
<td>18</td>
<td>55</td>
<td>!=</td>
<td>live subsumes</td>
<td>&lt;=</td>
<td>dead subsumes</td>
<td>&gt;=</td>
<td>live subsumed</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>TestPat_8</td>
<td>&lt;</td>
<td>FALSE</td>
<td>live</td>
<td>18</td>
<td>55</td>
<td>&lt;</td>
<td>live subsumed</td>
<td>==</td>
<td>live subsumed</td>
<td>&gt;=</td>
<td>live subsumed</td>
<td>OK</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TestPat_9</td>
<td>==</td>
<td>&gt;</td>
<td>live</td>
<td>19</td>
<td>30</td>
<td>==</td>
<td>live subsumes</td>
<td>FALSE</td>
<td>live subsumes</td>
<td>!=</td>
<td>dead subsumed</td>
<td>OK</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TestPat_10</td>
<td>==</td>
<td>&gt;=</td>
<td>live</td>
<td>19</td>
<td>30</td>
<td>TRUE</td>
<td>dead subsumed</td>
<td>&gt;</td>
<td>live subsumed</td>
<td>!=</td>
<td>dead subsumed</td>
<td>OK</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TestPat_11</td>
<td>==</td>
<td>&lt;</td>
<td>dead</td>
<td>19</td>
<td>30</td>
<td>&lt;=</td>
<td>dead subsumes</td>
<td>FALSE</td>
<td>live subsumes</td>
<td>!=</td>
<td>dead subsumed</td>
<td>OK</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TestPat_12</td>
<td>==</td>
<td>&lt;=</td>
<td>dead</td>
<td>19</td>
<td>30</td>
<td>TRUE</td>
<td>dead subsumed</td>
<td>&lt;</td>
<td>dead subsumed</td>
<td>!=</td>
<td>dead subsumed</td>
<td>OK</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TestPat_13</td>
<td>==</td>
<td>!=</td>
<td>dead</td>
<td>19</td>
<td>30</td>
<td>TRUE</td>
<td>dead subsumed</td>
<td>&lt;</td>
<td>dead subsumed</td>
<td>!=</td>
<td>dead subsumed</td>
<td>OK</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TestPat_14</td>
<td>==</td>
<td>TRUE</td>
<td>dead</td>
<td>19</td>
<td>30</td>
<td>&lt;</td>
<td>dead subsumes</td>
<td>&gt;</td>
<td>live subsumed</td>
<td>!=</td>
<td>dead subsumed</td>
<td>OK</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TestPat_15</td>
<td>==</td>
<td>FALSE</td>
<td>live</td>
<td>19</td>
<td>30</td>
<td>&lt;</td>
<td>dead subsumed</td>
<td>&gt;</td>
<td>live subsumed</td>
<td>!=</td>
<td>dead subsumed</td>
<td>OK</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TestPat_16</td>
<td>&lt;</td>
<td>&gt;</td>
<td>live</td>
<td>22</td>
<td>40</td>
<td>FALSE</td>
<td>unknow subsumes</td>
<td>!=</td>
<td>live subsumes</td>
<td>&gt;=</td>
<td>live subsumed</td>
<td>OK</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TestPat_17</td>
<td>&lt;</td>
<td>&gt;&gt;</td>
<td>live</td>
<td>22</td>
<td>40</td>
<td>&gt;</td>
<td>live subsumes</td>
<td>==</td>
<td>live subsumes</td>
<td>TRUE</td>
<td>live subsumed</td>
<td>OK</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TestPat_18</td>
<td>&lt;</td>
<td>&lt;=</td>
<td>live</td>
<td>22</td>
<td>40</td>
<td>&lt;=</td>
<td>live subsumed</td>
<td>TRUE</td>
<td>live subsumed</td>
<td>..</td>
<td>live subsumed</td>
<td>OK</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TestPat_19</td>
<td>&lt;</td>
<td>==</td>
<td>live</td>
<td>22</td>
<td>40</td>
<td>FALSE</td>
<td>unknow subsumes</td>
<td>&lt;=</td>
<td>live subsumes</td>
<td>&gt;=</td>
<td>live subsumed</td>
<td>OK</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TestPat_20</td>
<td>&lt;</td>
<td>!=</td>
<td>live</td>
<td>22</td>
<td>40</td>
<td>&gt;</td>
<td>live subsumed</td>
<td>TRUE</td>
<td>live subsumed</td>
<td>&gt;=</td>
<td>live subsumed</td>
<td>OK</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TestPat_21</td>
<td>&lt;</td>
<td>TRUE</td>
<td>live</td>
<td>22</td>
<td>40</td>
<td>!=</td>
<td>live subsumes</td>
<td>&lt;=</td>
<td>live subsumes</td>
<td>&gt;=</td>
<td>live subsumed</td>
<td>OK</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TestPat_22</td>
<td>!=</td>
<td>&gt;</td>
<td>live</td>
<td>23</td>
<td>45</td>
<td>FALSE</td>
<td>live subsumed</td>
<td>==</td>
<td>live subsumed</td>
<td>&gt;=</td>
<td>live subsumed</td>
<td>OK</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TestPat_23</td>
<td>!=</td>
<td>&gt;&gt;</td>
<td>live</td>
<td>23</td>
<td>45</td>
<td>TRUE</td>
<td>live subsumed</td>
<td>==</td>
<td>live subsumed</td>
<td>&gt;=</td>
<td>live subsumed</td>
<td>OK</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TestPat_24</td>
<td>!=</td>
<td>&lt;=</td>
<td>live</td>
<td>23</td>
<td>45</td>
<td>&lt;=</td>
<td>live subsumes</td>
<td>TRUE</td>
<td>live subsumed</td>
<td>==</td>
<td>live subsumed</td>
<td>OK</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TestPat_25</td>
<td>!=</td>
<td>==</td>
<td>live</td>
<td>23</td>
<td>45</td>
<td>FALSE</td>
<td>live subsumed</td>
<td>&gt;=</td>
<td>live subsumed</td>
<td>==</td>
<td>live subsumed</td>
<td>OK</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TestPat_26</td>
<td>!=</td>
<td>!=</td>
<td>live</td>
<td>23</td>
<td>45</td>
<td>&lt;</td>
<td>live subsumes</td>
<td>TRUE</td>
<td>live subsumed</td>
<td>==</td>
<td>live subsumed</td>
<td>OK</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TestPat_27</td>
<td>!=</td>
<td>TRUE</td>
<td>live</td>
<td>23</td>
<td>45</td>
<td>&lt;=</td>
<td>live subsumed</td>
<td>==</td>
<td>live subsumed</td>
<td>==</td>
<td>live subsumed</td>
<td>OK</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TestPat_28</td>
<td>!=</td>
<td>FALSE</td>
<td>live</td>
<td>23</td>
<td>45</td>
<td>&lt;</td>
<td>live subsumed</td>
<td>&gt;=</td>
<td>live subsumed</td>
<td>==</td>
<td>live subsumed</td>
<td>OK</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Appendix U displays a slightly modified variant of TestPat, which exposes its inner state by printouts. It is needed to track the inner state of TestPat. Below is the state change of the original program.

<table>
<thead>
<tr>
<th>TestPat (original unmutated file)</th>
</tr>
</thead>
<tbody>
<tr>
<td>run:</td>
</tr>
<tr>
<td>Initial values: NOTFOUND:-1 iSub:0 rtnIndex:-1 isPAT:false subjectLen:3 patternLen:1</td>
</tr>
<tr>
<td>While: NOTFOUND:-1 iSub:0 rtnIndex:-1 isPAT:false subjectLen:3 patternLen:1</td>
</tr>
<tr>
<td>While: NOTFOUND:-1 iSub:1 rtnIndex:-1 isPAT:false subjectLen:3 patternLen:1</td>
</tr>
<tr>
<td>While: NOTFOUND:-1 iSub:2 rtnIndex:-1 isPAT:false subjectLen:3 patternLen:1</td>
</tr>
<tr>
<td>Pattern string is not a substring of the subject string</td>
</tr>
<tr>
<td>BUILD SUCCESSFUL (total time: 0 seconds)</td>
</tr>
</tbody>
</table>

TestPat_3 follows a different path: There is a state change, so the fault becomes an error, but there is no propagation. The tool which can kill strongly cannot kill this.

<table>
<thead>
<tr>
<th>TestPat_3 original &lt; mutant &gt;= state Alive</th>
</tr>
</thead>
<tbody>
<tr>
<td>run:</td>
</tr>
<tr>
<td>Initial values: NOTFOUND:-1 iSub:0 rtnIndex:-1 isPAT:false subjectLen:3 patternLen:1</td>
</tr>
<tr>
<td>Pattern string is not a substring of the subject string</td>
</tr>
<tr>
<td>BUILD SUCCESSFUL (total time: 1 second)</td>
</tr>
</tbody>
</table>

TestPat_3 ("\>=") should be subsumed by TestPat _4 ("\<="). However this is not the case because the mutant breaks with a nullpointer in case of TestPat_4 resulting in propagation. TestPat_4 can therefore be killed strongly.

<table>
<thead>
<tr>
<th>TestPat_4 original &lt; mutant &lt;= state Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>run:</td>
</tr>
<tr>
<td>Initial values: NOTFOUND:-1 iSub:0 rtnIndex:-1 isPAT:false subjectLen:3 patternLen:1</td>
</tr>
<tr>
<td>While: NOTFOUND:-1 iSub:0 rtnIndex:-1 isPAT:false subjectLen:3 patternLen:1</td>
</tr>
<tr>
<td>While: NOTFOUND:-1 iSub:1 rtnIndex:-1 isPAT:false subjectLen:3 patternLen:1</td>
</tr>
<tr>
<td>While: NOTFOUND:-1 iSub:2 rtnIndex:-1 isPAT:false subjectLen:3 patternLen:1</td>
</tr>
<tr>
<td>While: NOTFOUND:-1 iSub:3 rtnIndex:-1 isPAT:false subjectLen:3 patternLen:1</td>
</tr>
<tr>
<td>Exception in thread &quot;main&quot; java.lang.ArrayIndexOutOfBoundsException: 3</td>
</tr>
</tbody>
</table>

Java Result: 1
BUILD SUCCESSFUL (total time: 0 seconds)
TestPat_5 follows different path, but the error does not propagate enough to be a failure. Strong killing is not enough here, either.

<table>
<thead>
<tr>
<th>TestPat_5</th>
<th>original &lt; mutant == state Alive</th>
</tr>
</thead>
<tbody>
<tr>
<td>run:</td>
<td>Initial values: NOTFOUND:-1 iSub:0 rtnIndex:-1 isPAT:false subjectLen:3 patternLen:1</td>
</tr>
<tr>
<td></td>
<td>Pattern string is not a substring of the subject string</td>
</tr>
<tr>
<td></td>
<td>BUILD SUCCESSFUL (total time: 0 seconds)</td>
</tr>
</tbody>
</table>

TestPat_3 (“\(\geq\)”) should be subsumed by TestPat_7 (“TRUE”). However this is not the case because the program breaks with a nullpointer in case of TestPat_7 resulting in propagation. TestPat_7 can be therefore strongly killed.

<table>
<thead>
<tr>
<th>TestPat_7</th>
<th>original &lt; mutant TRUE state Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>run:</td>
<td>Initial values: NOTFOUND:-1 iSub:0 rtnIndex:-1 isPAT:false subjectLen:3 patternLen:1</td>
</tr>
<tr>
<td></td>
<td>While: NOTFOUND:-1 iSub:0 rtnIndex:-1 isPAT:false subjectLen:3 patternLen:1</td>
</tr>
<tr>
<td></td>
<td>While: NOTFOUND:-1 iSub:1 rtnIndex:-1 isPAT:false subjectLen:3 patternLen:1</td>
</tr>
<tr>
<td></td>
<td>Exception in thread &quot;main&quot; java.lang.ArrayIndexOutOfBoundsException: 3</td>
</tr>
<tr>
<td></td>
<td>While: NOTFOUND:-1 iSub:2 rtnIndex:-1 isPAT:false subjectLen:3 patternLen:1</td>
</tr>
<tr>
<td></td>
<td>While: NOTFOUND:-1 iSub:3 rtnIndex:-1 isPAT:false subjectLen:3 patternLen:1</td>
</tr>
<tr>
<td></td>
<td>Java Result: 1</td>
</tr>
<tr>
<td></td>
<td>BUILD SUCCESSFUL (total time: 1 second)</td>
</tr>
</tbody>
</table>

While TestPat_2 (\(>\)), TestPat_6 (!=) and TestPat_8 (FALSE) are matching the pattern, they can be interesting to look on them too.

TestPat_2 follows a different path too. There is a state difference compared to the original, but it does not propagate. TestPat_2 could be killed weakly too.

<table>
<thead>
<tr>
<th>TestPat_2</th>
<th>original &lt; mutant &gt; state Alive</th>
</tr>
</thead>
<tbody>
<tr>
<td>run:</td>
<td>Initial values: NOTFOUND:-1 iSub:0 rtnIndex:-1 isPAT:false subjectLen:3 patternLen:1</td>
</tr>
<tr>
<td></td>
<td>Pattern string is not a substring of the subject string</td>
</tr>
<tr>
<td></td>
<td>BUILD SUCCESSFUL (total time: 1 second)</td>
</tr>
</tbody>
</table>

TestPat_6 is equivalent as it is displayed in Appendix M. The execution is identical to the original when it comes to inner states, which is expected from an equivalent mutant per definition.

<table>
<thead>
<tr>
<th>TestPat_6</th>
<th>original &lt; mutant != state Alive</th>
</tr>
</thead>
<tbody>
<tr>
<td>run:</td>
<td>Initial values: NOTFOUND:-1 iSub:0 rtnIndex:-1 isPAT:false subjectLen:3 patternLen:1</td>
</tr>
<tr>
<td></td>
<td>While: NOTFOUND:-1 iSub:0 rtnIndex:-1 isPAT:false subjectLen:3 patternLen:1</td>
</tr>
<tr>
<td></td>
<td>While: NOTFOUND:-1 iSub:1 rtnIndex:-1 isPAT:false subjectLen:3 patternLen:1</td>
</tr>
<tr>
<td></td>
<td>While: NOTFOUND:-1 iSub:2 rtnIndex:-1 isPAT:false subjectLen:3 patternLen:1</td>
</tr>
<tr>
<td></td>
<td>Pattern string is not a substring of the subject string</td>
</tr>
<tr>
<td></td>
<td>BUILD SUCCESSFUL (total time: 1 second)</td>
</tr>
</tbody>
</table>
TestPat_8 follows a different path too compared to the original program. There is a state difference as it terminates the loop direct, but this difference is not present on the final state. TestPat_8 could be killed weakly.

<table>
<thead>
<tr>
<th>TestPat_8</th>
<th>original&lt; mutant</th>
<th>FALSE state</th>
<th>Alive run:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial values: NOTFOUND:-1 iSub:0 rtnIndex:-1 isPAT:false subjectLen:3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pattern string is not a substring of the subject string</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BUILD SUCCESSFUL (total time: 1 second)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Appendix U - TestPat with exposed inner states

Main.java

package testpat;

class Main{

    public static void main (String []argv)
    {
        final int MAX = 100;
        char subject[] = new char[MAX];
        char pattern[] = new char[MAX];
        /*
        if (argv.length != 2)
        {
            System.out.println
            ("java TestPat String-Subject String-Pattern");
            return;
        }
        subject = argv[0].toCharArray();
        pattern = argv[1].toCharArray();
        */
        subject = "aaa".toCharArray();
        pattern = "aaa".toCharArray();
        TestPat testPat = new TestPat();
        int n = 0;
        if ((n = testPat.pat(subject, pattern)) == -1)
            System.out.println
            ("Pattern string is not a substring of the subject string");
        else
            System.out.println
            ("Pattern string begins at the character " + n);
    }
}

Mutant.java

/*@author mard*/

public interface Mutant {
    public int pat(char[] subject, char[] pattern);
}
class TestPat implements Mutant {

    public 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

        final int NOTFOUND = -1;
        int iSub = 0, rtnIndex = NOTFOUND;
        boolean isPat = false;
        int subjectLen = subject.length;
        int patternLen = pattern.length;

        System.out.println("Initial values: NOTFOUND:" + NOTFOUND + " iSub:" + iSub + " rtnIndex:" + rtnIndex + " isPAT:" + isPat + " subjectLen:" + subjectLen + " patternLen:" + patternLen);

        while (isPat == false && iSub + patternLen - 1 == subjectLen) {
            System.out.println("While: NOTFOUND:" + NOTFOUND + " iSub:" + iSub + " rtnIndex:" + rtnIndex + " isPAT:" + isPat + " subjectLen:" + subjectLen + " patternLen:" + patternLen);
            if (subject [iSub] == pattern [0]) {
                System.out.println("OuterIf: NOTFOUND:" + NOTFOUND + " iSub:" + iSub + " rtnIndex:" + rtnIndex + " isPAT:" + isPat + " subjectLen:" + subjectLen + " patternLen:" + patternLen);
                rtnIndex = iSub; // Starting at zero
                isPat = true;
                for (int iPat = 1; true; iPat++) {
                    System.out.println("For: NOTFOUND:" + NOTFOUND + " iSub:" + iSub + " rtnIndex:" + rtnIndex + " isPAT:" + isPat + " subjectLen:" + subjectLen + " patternLen:" + patternLen + " iPat:" + iPat);
                    if (subject[iSub + iPat] != pattern[iPat]) {
                        System.out.println("InnerIF: NOTFOUND:" + NOTFOUND + " iSub:" + iSub + " rtnIndex:" + rtnIndex + " isPAT:" + isPat + " subjectLen:" + subjectLen + " patternLen:" + patternLen + " iPat:" + iPat);
                        rtnIndex = NOTFOUND;
                        isPat = false;
                        break; // out of for loop
                    }
                }
                System.out.println("Out of for loop");
            }
            iSub ++;
        }
        return (rtnIndex);
    }
}