TOWARDS MINIMAL MUTATION ANALYSIS

Using the approximated dominator set of mutants

RESEARCH PROPOSAL

TOWARDS MINIMAL MUTATION ANALYSIS

Using the approximated dominator set of mutants

ANDRÁS MÁRKI
Informatics

UNIVERSITY
OF SKÖVDE

András Márki, 2019

Research Proposal

Title: Towards minimal mutation analysis

Using the approximated dominator set of mutants

University of Skövde, Sweden
www.his.se

ABSTRACT

In mutation testing, variants (i.e., mutants) of the software under test are created. The mutants are then used to design tests that can detect the difference between the mutants and the original software under test. Empirical studies have shown that test suites that are effective in detecting mutants are also effective in detecting real faults. Mutation analysis is therefore often used to benchmark effectiveness of other testing techniques. The main drawback of mutation testing is that it is computationally expensive because of the large number of mutants to analyze. It is well known that many of these mutants are redundant and recent studies have shown that the redundancy among the mutants can be up to 99%. However, identifying which mutants that are redundant is challenging since this depends on the software under test as well as the specific mutations.

This work aims to combine techniques from areas, such as static analysis and machine learning, in a process for cost-effective mutation analysis. Such techniques are expected to provide partial solutions to the problem of avoiding creation of the redundant mutants. The outcome of this research is two-fold: (i) an evaluation of techniques that can be used to minimize the set of non-redundant mutants that needs to be created, and (ii) a process for mutation analysis combining such minimization techniques. A framework will also be developed to evaluate the minimization techniques and the entire process.

keywords: mutation analysis, dominator mutants, redundant mutants, mutation testing, software testing, mutant minimization

PUBLICATIONS

Publications connected to this work are listed below. Publications 1 and 3 evaluate the limitations of a special form of static subsumption in the domain of strong and weak mutation. Publication 2 evaluates Java tools for mutation analysis with the goal to select a suitable tool to perform the research necessary to achieve the goal of the proposed work.

PUBLICATIONS WITH HIGH RELEVANCE

1. Lindström, Birgitta and Márki, András (2018). "On strong mutation and the theory of subsuming logic-based mutants." In: *Software Testing, Verification and Reliability* 29.1-2. DOI: 10.1002/stvr.1667.
2. Márki, András and Lindström, Birgitta (2017). "Mutation Tools for Java." In: *Proceedings of the Symposium on Applied Computing*. SAC '17. Marrakech, Morocco: ACM, pp. 1364–1415. ISBN: 978-1-4503-4486-9.

PUBLICATIONS WITH LOW RELEVANCE

3. Lindström, Birgitta and Márki, András (2016). "On strong mutation and subsuming mutants." In: *11th International Workshop on Mutation Analysis (MUTATION 2016)*. IEEE, pp. 112–121.

CONTENTS

1 **Introduction** 3

2 **Background** 5

2.1 Software testing 5

2.1.1 Covering the input domain & observing the defects 6

2.1.2 Example of software testing 6

2.2 Mutation analysis 9

2.2.1 Example: mutation analysis 11

2.2.2 Benefits of mutation analysis 15

2.2.3 Challenges of mutation analysis 16

2.2.4 Making mutation analysis cheaper 17

2.2.5 Dominator mutants 18

2.3 Symbolic execution 18

3 **Research Problem** 21

3.1 Goal 22

3.1.1 Research questions 22

3.1.2 Objectives 23

3.2 Preliminary results 24

4 **Research Method and Approach** 27

4.1 The scientific search process by design science 27

4.2 Literature review 28

4.3 Experiment and case study 29

4.4 The combination of the methods 30

5 **Research Plan** 33

6 **Expected Results** 35

7 **Related Work** 37

References 43

CHAPTER 1
INTRODUCTION

Mutation analysis is a strong but computationally expensive testing technique (Papadakis, Kintis, et al., 2019; Ammann and Offutt, 2016; Jia and Harman, 2011; Offutt, 2011; Offutt and Untch, 2001; DeMillo, Lipton, and Sayward, 1978). The idea of mutation analysis is to create variants of the program under test by systematically seeding syntactical changes into the program using mutation operators. Such program variants are called mutants, which mutants can be used both to evaluate and generate test cases. The syntactic changes are either similar to faults that programmers make or implement good testing practices such as branch coverage. The premise of mutation analysis is that if a test suite can reveal different behavior between each mutant and the original program (called killing the mutants), then the test suite is effective in finding real defects in the software under test (Chekam, Papadakis, Le Traon, and Harman, 2017; Offutt and Untch, 2001; Offutt and S. D. Lee, 1991; DeMillo, Lipton, and Sayward, 1978).

Mutation analysis is very effective as long as all the mutation operators are utilized to create the mutants. However 90-99% of the generated mutants are redundant, which makes the technique unnecessarily expensive (Kurtz, 2018; Papadakis, Henard, et al., 2016; Ammann, Delamaro, and Offutt, 2014). Hence, theoretically it is possible to reduce the number of mutants by one or two magnitudes without lowering the effectiveness (Kurtz, Ammann, Delamaro, et al., 2014). To do this, we need to know which mutants that would be redundant if generated and analyzed.

The purpose of the proposed research is to define a methodology and design a process to make mutation analysis more efficient without lowering its fault revealing ability. The process will avoid generating redundant mutants while guarantee that the non-redundant mutants are used in the analysis. The proposed research aims to make test design easier using mutation analysis.

The research proposal is divided into the following chapters. Chapter 2 covers the necessary terminology and grounded theory concerning the research problem at hand. Chapter 3 defines the aim and the objectives for the research project. Chapter 4 presents the methodology defined to achieve the goal for every objective and the overall aim. Chapter 5 contains the time plan of both the planned research and publications. Chapter 6 describes the contributions expected from this work. Chapter 7 puts this work in a broader perspective as well as presenting the related work in this field.

CHAPTER 2
BACKGROUND

This chapter gives a brief overview of software testing in general and mutation analysis in particular.

2.1 SOFTWARE TESTING

Software testing is a de facto standard to reach a desired level of confidence in the quality of the software under test by executing inputs and evaluating the output (Juristo, Moreno, and Vegas, 2004; Ammann and Offutt, 2016). Therefore, testing is an important activity during the software development process. Typically, testing tend to consume more than 50% of the development costs. This percentage is even higher for safety critical software (Myers, Badgett, Thomas, and Sandler, 2004; Gates, 2002; Harrold, 2000). Software testing is labor intensive and requires a high degree of automation (Ammann and Offutt, 2016).

A coverage criterion defines how to test the software in a complete, unambiguous and systematic way (Ammann and Offutt, 2016). The coverage criterion defines which test requirements apply for the software under test. For example the criterion branch coverage gives one test requirement for each branch. Test requirements can be based on source code, design, specification or the software´s input domain (Ammann and Offutt, 2016). Test cases should be created systematically based on test requirements, with the test requirements defining which aspect and to what extent a test case should cover (Ammann and Offutt, 2016). When all test requirements are covered, the quality of the test suite is known to fulfill the given coverage (Morell, 1990).

Coverage criteria can subsume other coverage criteria. In case a criterion c1 subsumes criterion c2, then any test suite which satisfies c1 will always satisfy c2 (Frankl and Weyuker, 1988). In case neither c1 subsumes c2, nor c2 subsumes c1, they are incomparable (Weyuker, Weiss, and Hamlet, 1991).

During software testing, a set of test cases denoted as a test suite is used to assert the quality of the software (Ammann and Offutt, 2016). A test case has input values, may require a program state before it can be executed (prefix values) and may need inputs sent to the program after the tests have executed (postfix values). After executing a test case, the result is compared with the expected result. A test oracle determines if the expected result matches the actual result. If they match, the test case passes, otherwise it fails.

To illustrate the connection between coverage criteria, test requirements, test cases and subsumption, regard the following example using predicate coverage. P is the set of all predicates in the software under test. For each $p \in P$ predicate coverage requires p to evaluate to true and p needs to evaluate to false (Ammann and Offutt, 2016). Therefore there should exist at least one test for each p where p evaluates to true and one test where p evaluates to false, so that all test requirements are fulfilled and 100% predicate coverage is achieved by the test suite. It can be proven that a test suite achieving 100%

predicate coverage will also cover 100% of the lines of code. Therefore predicate coverage subsumes the criterion to cover all lines of code.

2.1.1 COVERING THE INPUT DOMAIN & OBSERVING THE DEFECTS

Software testing has the limitation that only the presence of defects can be shown, but not their absence. The absence of defects cannot be shown for a number of reasons concerning complexity, controllability and observability.

The major reason is that the entire input domain cannot be tested in a general case since the number of test needed would be too many to run within reasonable time (Ammann and Offutt, 2016). A function taking an integer as an input parameter has usually 2^{32} possible input values and testing them all is impossible. While it is impossible to exhaustively cover the entire input domain, coverage criteria provides a solution to systematically cover the input domain using some key features of the program under test. For example such key features are code structures or domain partitions.

A second reason why absence of defects cannot be shown is because of controllability issues which is described by the *Reachability, Infection and Propagation* (RIP) model. Software can contain unreachable parts. Defects in the unreachable part of the software cannot be reached by any test, therefore no test can find such defects (Ammann and Offutt, 2016). The RIP model describes what is required from a test to find a defect in a software (Ammann and Offutt, 2016). Reach denotes that a static *fault* is executed with some input and state combination on the program. Infection requires the fault to cause an *error*, i.e. an internal erroneous state difference. Propagation requires that the internal error state propagates to the output of the program thus causing a *failure*.

A third reason is observability and revealability, i.e. in some systems, some variables are difficult or impossible to monitor, which motivates by the expansion of the RIP model with revealability, resulting in the *Reachability, Infection, Propagation and Revealability* (RIPR) model (Ammann and Offutt, 2016). Revealability requires that the incorrect part of the final program state, also the failure, is observable by the tester. With other words, revealability requires the necessary variables to be monitorable by the tester (Li and Offutt, 2017; Ammann and Offutt, 2016). It is possible that the incorrect portion of the program state is not observable (Ammann and Offutt, 2016).

2.1.2 EXAMPLE OF SOFTWARE TESTING

The following section illustrates the connection between software under test, coverage criterion, test requirement, test suite, test case and defect finding ability.

An example program *countNatural* can be seen in figure 2.1 (Lindström and Márki, 2018; Ammann and Offutt, 2016). The example program takes an array of integers (*inputArray*) as input and returns the number of occurrences of natural numbers in the array.

To keep the example simple, predicate coverage is used as a coverage criterion. Predicate coverage gives two test requirement for each predicate p from the set of predicates P concerning the program under test: p needs to evaluate to true and p needs to evaluate to false (Ammann and Offutt, 2016). countNatural has two predicates:

- p1: "*i* < *inputArray.length*" at line 6
- p2: "*inputArray[i]* >= *0*" at line 7

```
1  package dominator;
2
3  public class CountNatural {
4      public int countNatural(int[] inputArray) {
5          int count  = 0;
6          for(int i = 0; i < inputArray.length; i++) {
7              if(inputArray[i] >= 0) {
8                  count++;
9              }
10         }
11         return count;
12     }
13 }
```

Figure 2.1: Example program countNatural

To fulfill all test requirements needed to achieve predicate coverage, both predicates p1 and p2 should evaluate to true and to false resulting in the following set of requirements

1. tr1: p1 evaluates to true
2. tr2: p1 evaluates to false
3. tr3: p2 evaluates to true
4. tr4: p2 evaluates to false

Requirements covered by test cases		
Test requirement	Test case	Comments
tr1: p1 true	{-1, 1}	p1 true in 1st iteration
tr2: p1 false	{}	p1 false in 1st iteration
tr3: p2 true	{-1, 1}	p2 true in 2nd iteration
tr4: p2 false	{-1, 1}	p2 false in 1st iteration

Table 2.1: Mapping between test requirements and test cases

A JUnit test suite seen in figure 2.2 consist of two tests, t1: {-1,1} and t2: {}. Table 2.1 displays the mapping between the four test requirements and the two test cases required to cover them. The example displays that a single test can cover multiple test requirements at the same time.

A defective countNatural containing a single defect can be seen in figure 2.3. The defect is present in line seven. In the defective version all positive and negative numbers are counted, but no zeroes.

The results for the defective countNatural can be seen in table 2.2. The only test detecting the defect is t1, as t2 does not reach the location of the fault.

```
1  package dominator;
2
3⊕ import static org.junit.jupiter.api.Assertions.*;
6
7  class CountNaturalTest {
8      CountNatural countNatural = new CountNatural();
9
10⊖     @Test
11     void t1() {
12         int[] array = {-1,1};
13         assertEquals(1,countNatural.countNatural(array));
14     }
15⊖     @Test
16     void t2() {
17         int[] array = {};
18         assertEquals(0,countNatural.countNatural(array));
19     }
20 }
```

Figure 2.2: Example test suite

```
1  package dominator;
2
3  public class CountNaturalDefective {
4⊖     public int countNaturalDefective(int[] inputArray) {
5          int count  = 0;
6          for(int i = 0; i < inputArray.length; i++) {
7              if(inputArray[i] != 0) {
8                  count++;
9              }
10         }
11         return count;
12     }
13 }
```

Figure 2.3: Example program countNatural with defect

Test results			
Test	Expected result	Actual result	Test output
t1: {-1,1}:	1	2	fail
t2: {}:	0	0	pass

Table 2.2: Table displaying the results on the defective countNatural with the example test suite

2.2 MUTATION ANALYSIS

Mutation analysis works by creating mutants from the software under test using mutation operators. The mutants are variants of the software under test created by systematically applying well-defined mutation operators to the software. Test suites can be evaluated with respect to their ability to distinguish the original program from the mutant. Mutation analysis can also be used to design test suites that detect all mutants, i.e., mutation testing.

Mutants can have different states during mutation analysis (Offutt and Untch, 2001; DeMillo and Offutt, 1991):

- **Alive / Undetected**: A mutant is alive (undetected), if no test until that point distinguished it from the original program.
- **Dead / Detected**: A mutant is dead (detected) if at least one test has distinguished it from the original program

The mutation score equals the number of mutants killed by the test suite divided with the total number of mutants (Offutt and Untch, 2001; DeMillo and Offutt, 1991). A mutation score of 100% is achieved when the tests can distinguish all mutants from the original program. Mutation score, sometimes also denoted as mutation coverage, can both be used to evaluate the strength of the test suite as well as a guiding the creation of test cases (Papadakis, Chekam, and Le Traon, 2018). The kill matrix, sometimes referred as score function, presents a view on which mutants were killed by which tests (Kurtz, 2018; Ammann, Delamaro, and Offutt, 2014).

The mutation operators are well-defined rules to systematically create mutants from the original program. A mutator is the actual implementation of a mutation operator function implemented in a tool. For example, the relation operator replacement operator (ROR) replaces the operators <, >, <=, >=, == and != in the code to all other operators, as well as replaces the whole relation expression which is mutated with true and false.

Mutation analysis compares the behaviour of the original program and the mutant when executing test cases to kill mutants, but there are multiple approaches to do the comparison. Mutants can be classified as dead based either on their internal or final state. Weak mutation compares the internal state of the mutant with the original and thus only requires that the test causes an infection (DeMillo, Lipton, and Sayward, 1978; Howden, 1982). Strong mutation requires both propagation and revealability as mutants are detected by comparing the final program states (Chekam, Papadakis, Le Traon, and Harman, 2017; Offutt and S. D. Lee, 1991). Strong and weak mutation can be considered as part of a spectrum depending on where the program states are observed to detect mutants (Hierons, Harman, and Danicic, 1999). For example a failure at the function level can become an error at the component level (Hierons, Harman, and Danicic, 1999).

Mutants can be classified based on how hard is it to detect them:

- **Equivalent**: Mutants that are semantically equivalent to the original program, even though they are syntactically different. Tests cannot detect them.
- **Stubborn**: Non-equivalent mutants which are difficult to detect in practice (Yao, Harman, and Jia, 2014; Hierons, Harman, and Danicic, 1999).
- **Ordinary**: Non-equivalent mutants that are neither stubborn, nor trivial.

- **Trivial**: Mutants detected by the majority of the tests (Offutt, Voas, and Payne, 1996).
- **Stillborn**: A mutant that contains a syntactical change that prevents compilation. Such mutants are removed as they are not regarded as useful (Offutt, Voas, and Payne, 1996).

Mutation adequacy means that a test suite can detect all non-equivalent mutants, meaning a mutation score of 100% (Offutt and Untch, 2001; Offutt, A. Lee, et al., 1996). For larger programs mutation adequacy is however impractical because differentiating stubborn and equivalent mutants is difficult as it cannot be done in a fully automated way (Yao, Harman, and Jia, 2014).

Subsumption relation for mutation is defined on both mutant and operator level (Kurtz, Ammann, Delamaro, et al., 2014). A mutant M1 subsumes another mutant M2 if and only if every test that kills M1 also kills M2, given that M1 is at least killed by one test (Kurtz, Ammann, Delamaro, et al., 2014). An operator O1 subsumes O2 if and only if any set of tests that that kills all mutants created from O1 is guaranteed to also kill all mutants created from O2 (Kurtz, Ammann, Delamaro, et al., 2014). If two or more mutants show the same behaviour for all test cases, such mutants are indistinguishable from each other (Kurtz, Ammann, Delamaro, et al., 2014). A mutant subsumption graph (MSG) is a visualization showing the subsumption relations between mutants (Kurtz, Ammann, Offutt, et al., 2016; Kurtz, Ammann, Delamaro, et al., 2014). In case a test suite can kill all subsuming mutants, then it will certainly kill all mutants. Hence the subset of subsumed mutants are redundant.

There are three types of subsumption defined for mutation analysis (Kurtz, Ammann, Delamaro, et al., 2014). True subsumption is related to the entire input domain and would therefore need complete knowledge on the relation of all mutants to each other. True subsumption is however not computationally decidable so it must be approximated (Kurtz, Ammann, Delamaro, et al., 2014). An MSG based on true subsumption is called a true mutant subsumption graph (TMSG). Dynamic subsumption is relative to the test suite used, resulting in a dynamic mutant subsumption graph (DMSG). A DMSG is an approximation of the TMSG, but a full mutation analysis using all mutants is required to create such a DMSG (Kurtz, Ammann, Delamaro, et al., 2014). Static subsumption is based on an analysis of the mutants, resulting in a static mutant subsumption graph (SMSG), which is an approximation of the TMSG that can be created without performing a full mutation analysis (Kurtz, Ammann, and Offutt, 2015; Kurtz, Ammann, Delamaro, et al., 2014).

A small example on the mutant subsumption graph can be seen on figure 2.4 (Kurtz, Ammann, Delamaro, et al., 2014). Nodes represent maximal sets of indistinguishable mutants and edges represent subsumption relations. The mutants in the equivalent / alive node are either equivalent, or depending on how the MSG is created, not yet killed by any tests. Mutants in node 1 and 2 subsume mutants in node 4 and by transitivity mutants in node 5. In this example, mutants in node 1, 2 and 3 subsume all other mutants.

There are also some mutants, which are *duplicated*, i.e. equivalent to other mutants (Papadakis, Henard, et al., 2016; Papadakis, Jia, Harman, and Le Traon, 2015). Duplicated mutants are indistinguishable from each other and thus represented by the same node in the MSG. Such mutants are also redundant (Kurtz, Ammann, Offutt, et al., 2016; Just and Schweiggert, 2015; Ammann, Delamaro, and Offutt, 2014; Kurtz, Ammann, Delamaro, et al., 2014). 90-99% of the mutants show to be redundant (Papadakis, Henard, et

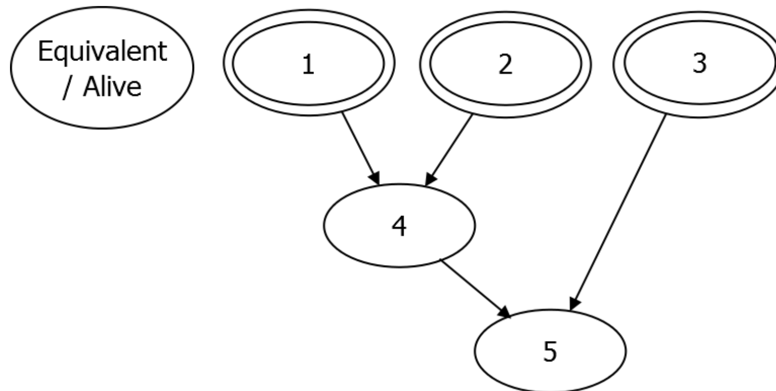

Figure 2.4: Example mutant subsumption graph

al., 2016; Ammann, Delamaro, and Offutt, 2014). Redundant mutants do not contribute to mutation analysis, as they are guaranteed to be detected when the non-redundant mutants are detected. Additionally, the large number of redundant mutants makes it possible for test suites to achieve high mutation score without killing non-redundant mutants. Therefore, a minimal set of dominator mutants, i.e., one mutant from each MSG node with dominator mutants, would be enough for maintaining the full effect of a complete mutation analysis that uses all mutants.

Figure 2.5 visualizes the mutation analysis process (Papadakis, Kintis, et al., 2019; Jia and Harman, 2011; Offutt and Untch, 2001). The steps marked with bold edges are the steps that require manual human decisions. The threshold defining how high the mutation score should be before regarding the test suite as strong enough is defined by a human operator. The verdict of the test as well as the necessary debugging process require human intervention too. The rest of the process is usually automatized either completely or to a high degree.

Mutation analysis is syntax based and therefore different languages require different tools. Mutation operators can also be implemented differently by different tools (Márki and Lindström, 2017). Therefore the mutation score might vary between tools even when the same test suite is used with the same program under test (Márki and Lindström, 2017; Laurent et al., 2017; Delahaye and Du Bousquet, 2013). Mutation tools were created for a number of programming languages, such as Fortran77 (DeMillo, Guindi, et al., 1988), Ada (Offutt, Voas, and Payne, 1996), Java (Just, 2014; Ma, Offutt, and Kwon, 2005; Coles et al., 2016), C (Denisov and Pankevich, 2018; Jia and Harman, 2008b; Delamaro, Maldonado, and Vincenzi, 2001; Phan, Y. Kim, and M. Kim, 2018) and C# (Derezińska and Szustek, 2009).

2.2.1 EXAMPLE: MUTATION ANALYSIS

Figure 2.6 shows an example of consisting of two mutated relational operator with the list of the created mutants. The analysis results observed on the final program states after executing three tests are presented in table 2.3 (Lindström and Márki, 2018; Ammann and Offutt, 2016). Only the relational operator replacement (ROR) mutation operator is applied to the software, however both relational operators are mutated (Ammann and Offutt, 2016).

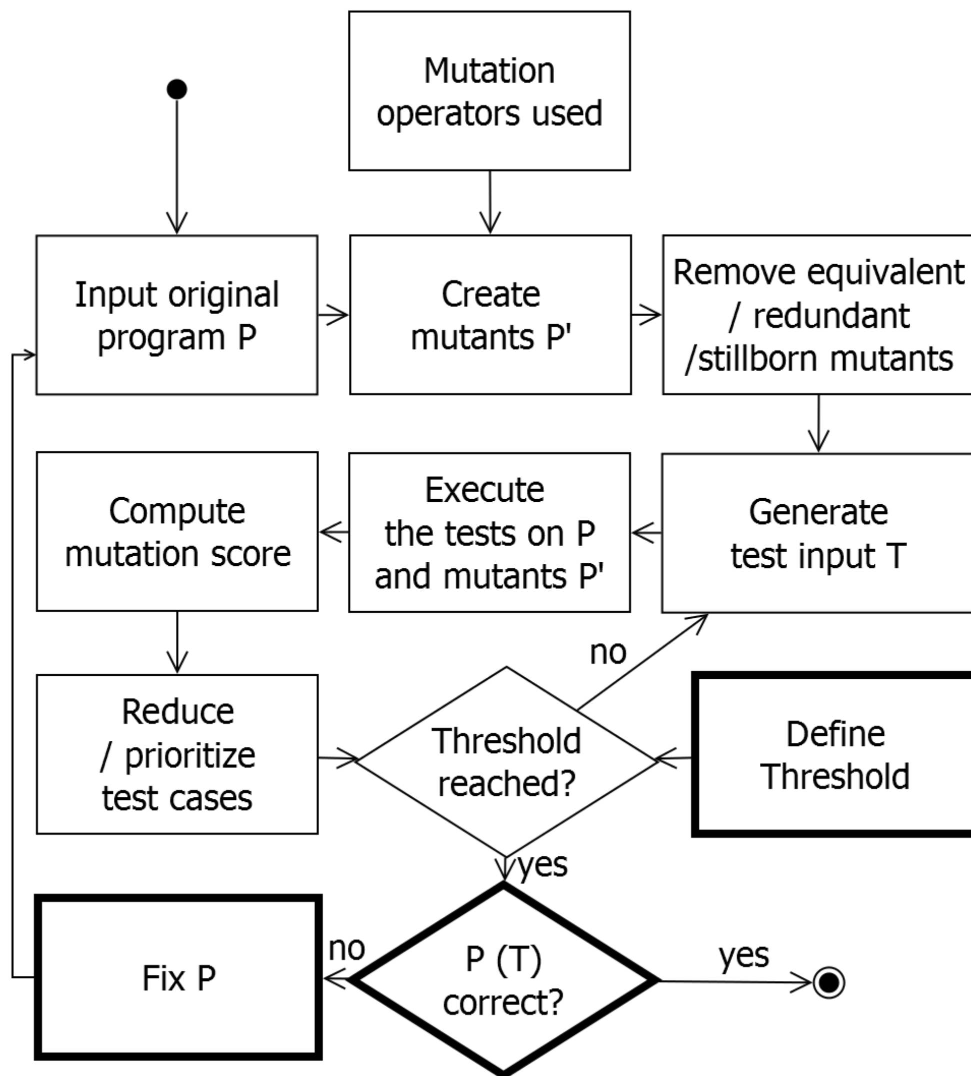

Figure 2.5: The mutation analysis process, based on (Papadakis, Kintis, et al., 2019; Jia and Harman, 2011; Offutt and Untch, 2001).

```
1  package dominator;
2
3  public class CountNatural {
4      public int countNatural(int[] inputArray) {
5          int count  = 0;
6          for(int i = 0; i < inputArray.length; i++) {    // ORIG1
7  //      for(int i = 0; i > inputArray.length; i++) {    // ROR_1
8  //      for(int i = 0; i <= inputArray.length; i++) {   // ROR_2
9  //      for(int i = 0; i >= inputArray.length; i++) {   // ROR_3
10 //      for(int i = 0; i == inputArray.length; i++) {   // ROR_4
11 //      for(int i = 0; i != inputArray.length; i++) {   // ROR_5
12 //      for(int i = 0; true; i++) {                     // ROR_6
13 //      for(int i = 0; false; i++) {                    // ROR_7
14             if(inputArray[i] >= 0) {        // ORIG2
15             //if(inputArray[i] < 0) {       // ROR_8
16             //if(inputArray[i] > 0) {       // ROR_9
17             //if(inputArray[i] <= 0) {      // ROR_10
18             //if(inputArray[i] == 0) {      // ROR_11
19             //if(inputArray[i] != 0) {      // ROR_12
20             //if(true) {                    // ROR_13
21             //if(false) {                   // ROR_14
22                 count++;
23             }
24         }
25         return count;
26     }
27 }
```

Figure 2.6: Example of mutation analysis using the ROR operator applied on two syntactic elements

The function countNatural counts the number of natural numbers in an array of integers. The ROR mutation operator is applied to the relational operator <(ORIG1) and >= (ORIG2). The ROR operator replaces the original operators with all the other relational operator resulting in the mutants ROR1...ROR5 for ORIG1 and ROR8...ROR12 for ORIG2. The whole relational expression is also replaced by true and false, resulting in ROR6...ROR7 for ORIG1 and ROR13...ROR14 for ORIG2.

The mutation for ORG1 affects when the loop is terminated. ROR1 terminates directly. ROR2 tries to access an array element which is outside the boundaries of the array, resulting in an exception. ROR3 has an inverse termination condition compared to the original, making it a trivial mutant. ROR4 only enters the loop with an empty array, at which point an element is evaluated which does not exists, resulting also in a trivial mutant. ROR5 is equivalent to the original operator in this case, because semantically it has the same behaviour as the original. The syntactical change in ROR6 results in an infinite loop which cannot be exited. Because the Java compiler automatically detects that the loop cannot terminate in this case, ROR6 cannot be compiled in Java. ROR7 results in dead code, as the loop cannot be entered. As the Java compiler detects the dead code in this particular case, ROR7 cannot be compiled either. Therefore both ROR6 and ROR7 are stillborn mutants when using the Java compiler, however both mutants may results in compilable mutants using other compilers.

The mutation for ORIG2 affects the counting. ROR8 counts negative numbers. ROR9 counts positive numbers. ROR10 counts negative numbers and zero values. ROR11 only counts zeros, while ROR12 counts everything except for zeros. ROR13 counts everything, while ROR14 does not count anything. Only 12 mutants out of 14 can be used for mutation analysis, as ROR6 and ROR7 are removed, because they are stillborn.

From the 12 non-stillborn mutants generated as an example in this case, not all mutants are equally useful. ROR5 is equivalent and should be removed from the test suite as no test can kill it. ROR2, ROR3 and ROR4 are trivial mutants. As any test would kill them, they are not useful to generate tests. Additionally, ROR2, ROR3 and ROR4 are subsumed by every other mutant. ROR2, ROR3 and ROR4 can be removed from the set of mutants to make mutation analysis cheaper while keeping its effectiveness, resulting in 8 mutants instead of the original 14.

The test suite which is able to kill all non-equivalent mutants consists of three test cases, t1:{-1, 1}, t2:{} and t3:{-2, -1, 0}. As it was shown previously, the test suite consisting of {-1, 1} and {} is sufficient for predicate coverage, but that suite would miss the mutants ROR8...ROR10, therefore {-2, -1, 0} is added. When reachability is taken into consideration, {} could be omitted to be executed on ROR8...ROR14 as that test case cannot reach the position of ORIG2, which could also possibly lower the cost of mutation analysis.

This ROR example is a small example and ROR is just one of the many mutation operators. For example, the mutation tool MuJava implements 16 different mutation operators in total and generates 39 mutants for countNatural (Márki and Lindström, 2017; Ma, Offutt, and Kwon, 2005).

This example displays both the strength and weakness of mutation analysis. Looking only at the ROR mutants present in the function countNatural, branch coverage of the test suite is required to kill all of them. Additionally, there are mutants connected to edge cases, something which was not the case for predicate coverage on an earlier example. With all mutation operators utilized, mutation analysis is useful to create strong test suites, something which can be important for industrial use. Some mutants are however less useful. There is an equivalent mutant present even in this example. Three of the mutants would be detected by any test case. While they can be detected, the three mu-

tants does not make the test suite stronger. One test cannot reach seven of the mutants, meaning that test is executed on those seven mutants in vain.

Mutation analysis results				
Original program and mutants	Test t1: {-1,1}	Test t2: {}	Test t3: {-2,-1,0}	Mutant status
ORIG1: i<inputArray.length	1	0	1	(original)
ROR1: i>inputArray.length	0	0	0	dead
ROR2: i<=inputArray.length	exception	exception	exception	dead (trivial)
ROR3: i>=inputArray.length	0	exception	0	dead (trivial)
ROR4: i==inputArray.length	0	exception	0	dead (trivial)
ROR5: i!=inputArray.length	1	0	1	alive (equivalent)
ROR6: true	compilation error	compilation error	compilation error	stillborn
ROR7: false	compilation error	compilation error	compilation error	stillborn
ORIG2: inputArray[i]>=0	1	0	1	(original)
ROR8: inputArray[i]<0	1	0	2	dead
ROR9: inputArray[i]>0	1	0	2	dead
ROR10: inputArray[i]<=0	1	0	3	dead
ROR11: inputArray[i]==0	0	0	1	dead
ROR12: inputArray[i]!=0	2	0	0	dead
ROR13: true	2	0	4	dead
ROR14: false	0	0	0	dead

Table 2.3: Mutation analysis results on the mutants of two original relational operators after executing the tests {-1,1}, {} and {-2,-1,0}

2.2.2 BENEFITS OF MUTATION ANALYSIS

Mutation analysis forces tests to cover all parts of the program under test as the mutants are created by systematically making small changes at every part of the code. Mutation coverage is achieved when a test suite can detect all non-equivalent mutants and detection requires that the location for the mutated code is reached and causes an infection.

Killing mutants using strong mutation also requires propagation and revealability. (Papadakis, Kintis, et al., 2019). To summarize strong mutation ensures both that the test suite includes tests executing all parts of the program and that these tests result in execution differences observable by the output. This makes mutation testing unique, since no other coverage criterion covers the entire RIPR model.

Mutation testing has the potential to subsume other testing techniques (Papadakis, Kintis, et al., 2019; Ammann and Offutt, 2016). Mutation testing was shown to be more effective but also more expensive compared to graph based criteria (Kakarla, Momotaz, and Namin, 2011; Li, Praphamontripong, and Offutt, 2009; Frankl, Weiss, and Hu, 1997).

2.2.3 CHALLENGES OF MUTATION ANALYSIS

Mutation analysis is computationally expensive. Performing mutation analysis means that in worst case, all tests should be executed on both the original program and all of its mutants. The number of mutants is O(References * Variables), which usually results in a high actual number of mutants (Offutt, A. Lee, et al., 1996; Budd, DeMillo, Lipton, and Sayward, 1980). Therefore, mutation analysis is computationally expensive and requires tool support. The need to execute every test in combination with every mutant creates issues to perform mutation analysis ever since the first tool was implemented (DeMillo and Offutt, 1991).

As mentioned in previous sections, not all mutants are however useful. Equivalent mutants are regarded as useless and should be removed, however not all of such mutants can be removed automatically, as differentiating stubborn mutants from equivalent mutants automatically can be difficult (Yao, Harman, and Jia, 2014; Fraser and Zeller, 2012; Offutt and Untch, 2001; Offutt and Pan, 1997; DeMillo, Lipton, and Sayward, 1978). The evaluation of equivalent mutants can also be regarded as a more specific variant of the feasible path problem (Offutt and Pan, 1997). Program slicing and compiler dependent solutions can be used to identify equivalent mutants (Papadakis, Jia, Harman, and Le Traon, 2015; Hierons, Harman, and Danicic, 1999). Keeping equivalent mutants means that the test suite will be judged as weaker than it actually is, resulting in an underestimated mutation score (Yao, Harman, and Jia, 2014). On the other hand, incorrectly marking stubborn mutants as equivalent results in an overestimation of the mutation score (Frankl, Weiss, and Hu, 1997). Overestimating the mutations score can also falsely increase the confidence in the strength of the test suite. Underestimating the mutation score can needlessly increase the cost of testing without any gains. Both the underestimation and overestimation of the mutation score weakens the confidence in mutation analysis as a benchmark tool to compare the strength of test suites. Such benchmarking is necessary to evaluate new test techniques.

To sum it up, the high percentage (90-99%) of redundant mutants creates problems for two reasons. The large number of redundant mutants contributes to mutation analysis being expensive (Jia and Harman, 2011). Mutation score becomes less accurate because of the redundant mutants. Confidence in the strength of the tests is reduced as a test suite can get high mutation score, even when the tests cannot kill the most important non-redundant mutants. Therefore, the correlation between the mutation score and the fault detecting strength of the test suite is weakened (Just and Schweiggert, 2015). Additionally, in case mutation analysis is used to benchmark other techniques, the less effective technique may get a better mutation score because it detects more redundant mutants.

2.2.4 MAKING MUTATION ANALYSIS CHEAPER

There are three overall approaches to make mutation analysis computationally cheaper: "*do smarter*", "*do faster*" and "*do fewer*" (Offutt and Untch, 2001). "Do smarter" focuses on solutions such as using distributed computing or including several syntactical changes into a single mutant. "Do faster" mostly means some incorporated tool optimization when creating and/or executing mutants (Delamaro, Maldonado, and Vincenzi, 2001; Just, 2014; Coles et al., 2016). "Do fewer" targets the reduction of either the number of mutants or the test cases. In the worst case, the cost of mutation testing is O((M+1)*T), where M is the set of all mutants generated for the program and T is the cardinality of the test suite, as all tests are needed to be executed on both the original program and the mutants. Therefore reducing the number of mutants or tests makes mutation analysis cheaper. The proposed work in this research proposal is classified as a do fewer approach since one goal is to reduce the number of mutants.

Do smarter either includes more seeded faults into a single mutant or uses parallel execution. First order mutation seeds a single fault in every mutant (DeMillo, Guindi, et al., 1988), while higher order mutation (HOM) combines several mutations into a single mutant, i.e. higher order mutant (DeMillo, Lipton, and Sayward, 1978; Budd, DeMillo, Lipton, and Sayward, 1978). A subsuming HOM is a variant of higher order mutation (HOM) with mutants, which are subsuming the first order mutants they were created from (Jia and Harman, 2008a). HOM has the benefit of reducing the number of executions as mutants incorporate a number of syntactical changes instead of just one. However, most mutation tools only supports first order mutation. In most cases, when the order is not mentioned, first order mutation is used.

Mutants can be executed in parallel even in case of very large projects (Petrovic and Ivankovic, 2018). Depending on how the tests are executed, mutation analysis scales well in a multiprocessed environment as tests usually can be executed on mutants in a parallel manner.

Tool and compiler optimization is sometimes utilized to make mutation testing more efficient. Meta mutants can be created and mutation can be utilized on the bytecode level (Delamaro, Maldonado, and Vincenzi, 2001; Untch, Offutt, and Harrold, 1993). Compiler-integrated mutation can also be utilized (Just, 2014; Coles et al., 2016; DeMillo, Krauser, and Mathur, 1991). Compiler-integrated solutions tend to be faster than creating mutants and saving mutants in the file system (Márki and Lindström, 2017). However, in some cases it is beneficial to create all mutants on the disc instead of creating mutants as part of the compilation (Ma, Offutt, and Kwon, 2005). Having the mutants available on the disc can help for example with observability.

"Do fewer" approaches decrease the execution time by either reducing the number of executed test cases or the number of mutants. Subsumption relation between mutants can be used to reduce the number of mutants needed to be analyzed. Static analysis can be employed to only execute a test on a mutant if the given test can reach the mutated code element.

With "do fewer" the number of tests executed on mutants can be reduced. A common practice is not to execute any more tests on a mutant, which is already killed by other tests (Just, 2014). Test cases that are not contributing to the mutation score can be removed from the test suite (Offutt, A. Lee, et al., 1996).

Mutation sampling only uses a subset of mutants instead of all of them. One way to do this is to only use a subset of the mutation operators, a technique called *selective mutation* (Offutt, A. Lee, et al., 1996). Alternatively, all mutation operators can be kept,

but instead of all mutants, a random or semi-random set of mutants can be chosen to be used, thus reducing the execution time resulting in a more scalable solution called *random selection* (J. Zhang, Zhu, Hao, and L. Zhang, 2014). Selective mutation techniques can however be ineffective as they are not refined enough to include non-redundant mutants (Kurtz, Ammann, Delamaro, et al., 2014).

2.2.5 DOMINATOR MUTANTS

As described previously, there exists a minimal subset of all mutants with the property that if all mutants in this subset are killed by a test suite, the same test suite is guaranteed to kill all mutants for the given program. These dominator mutants subsume all other mutants for a given program (Kurtz, Ammann, Offutt, et al., 2016; Ammann, Delamaro, and Offutt, 2014; Kurtz, Ammann, Delamaro, et al., 2014). Dominator mutants are therefore non-redundant.

The mutants in the root nodes of an MSG are the dominator mutants. As for dynamic subsumption solutions, the dynamic mutant subsumption graph (DMSG) gives an upper bound on how many of the mutants are necessary to keep the effectiveness of mutation analysis with respect to a particular test suite (Kurtz, Ammann, Offutt, et al., 2016; Kurtz, Ammann, Delamaro, et al., 2014). While a DMSG enables approximation of a TMSG and enables identification of dominator mutants, DMSG has the major drawback that it requires the execution of all tests on all mutants, i.e, a full mutation analysis. Therefore DMSG is not an useful approach for test design and it does not make mutation analysis cheaper.

2.3 SYMBOLIC EXECUTION

Symbolic execution can help to decide which tests (inputs) should or should not be executed on the mutants, depending if the input can reach a given mutant. Symbolic execution is a form of static analysis. Static analysis can be used in other situations less important for this work, for example parsing and type related error checking and verifying program conformance with formal specification.

Symbolic execution enables executing programs without initializing the inputs first, using symbolic values instead of actual data (King, 1976; Boyer, Elspas, and Levitt, 1975; Howden, 1975). This results in the output of the program being a function of symbolic values. A path condition contains information about which requirements the symbolic input values must fulfill, in order for the program to take a given path within the program. The path condition is described as a logical expression.

Constraint solvers can help solving the expressions for the path condition. Constraint solvers are therefore useful in combination with symbolic execution to determine, which tests cases are useful to detect which mutants (Wotawa, Nica, and Aichernig, 2010; DeMillo and Offutt, 1991).

Backward use of symbolic execution is also possible, starting from a final node of a program representation and going backwards to the start node (McMinn, 2004). Infeasible paths can however be harder to detect with this approach (McMinn, 2004).

There is a variant of dynamic symbolic execution, which creates path conditions dynamically called dynamic domain reduction (Offutt, Jin, and Pan, 1999; Offutt, Jin, and Pan, 1994). This approach has the benefit of being able to find paths even in situations where

symbolic execution cannot. A tool for dynamic domain reduction was created, however only for testing software written in Fortran (Offutt, Jin, and Pan, 1999).

Tests created using symbolic execution can be used to create tests which can distinguish a mutant from the original operator (Kurtz, Ammann, and Offutt, 2015). Data gathered during symbolic execution can be used to approximate the DMSG using the statically derived mutant subsumption graph (SDMSG) without actually executing tests (Kurtz, Ammann, and Offutt, 2015).

A special form of static analysis technique called static subsumption relation uses logic-based rules to mark mutants as redundant. Mutants marked as redundant are neither created nor analyzed. Such static subsumption relations are however only identified for the relational operator replacement (ROR) and the conditional operator replacement (COR) operator, but not for other operators (Kaminski, Ammann, and Offutt, 2011; Just, Kapfhammer, and Schweiggert, 2012). It has also been shown that these static subsumption relation only holds for weak mutation (Lindström and Márki, 2018; Lindström and Márki, 2016).

CHAPTER 3
RESEARCH PROBLEM

Mutation analysis has been shown to be effective for test design. However mutation analysis is still considered too expensive for most of the industry to adopt the technique (Papadakis, Kintis, et al., 2019; Ammann and Offutt, 2016). Improving efficiency would make the technique more attractive to industry.

90-99% of the mutants are shown to be redundant (Kurtz, 2018; Papadakis, Henard, et al., 2016; Ammann, Delamaro, and Offutt, 2014). The large percentage of redundant mutants means that the efficiency of mutation analysis could be improved by 1-2 magnitudes without reducing its effectiveness. Additionally, reducing redundancy among mutants would increase validity of benchmarking results.

A possible solution to make mutation analysis cheaper is to use the set of non-redundant dominator mutants instead of all mutants. This requires however the identification of the set of dominator mutants before the mutation analysis and preferably before the redundant mutants are created.

The set of dominator mutants depends on the software under test, however the set is not directly connected to specific mutations or mutation operators. Mutation sampling, i.e. the use of a (semi) randomly selected subset of the mutants is unlikely to include all dominator mutants. Therefore random selection decreases the effectiveness of mutation analysis (Papadakis, Henard, et al., 2016; Kurtz, Ammann, Offutt, et al., 2016; Kurtz, Ammann, Delamaro, et al., 2014; Ammann, Delamaro, and Offutt, 2014). Besides decreased effectiveness of mutation analysis, the test suite strength can be overestimated when dominator mutants are not included, resulting in a weakened reliability of mutation analysis as a gold standard to evaluate other test techniques (Kurtz, Ammann, Offutt, et al., 2016; Papadakis, Henard, et al., 2016; Just and Schweiggert, 2015; Ammann, Delamaro, and Offutt, 2014; Kurtz, Ammann, Delamaro, et al., 2014). To conclude, the dominator mutants should be included to keep mutation analysis effective.

Dominator mutants are difficult to identify. Since they depend on the combination of the program under test and the mutation operators used (Kintis, Papadakis, Papadopoulos, et al., 2018). A true mutation subsumption graph (TMSG) would identify all dominator mutants, however such a TMSG can only be approximated in practice. Approximation can be done by executing tests (DMSG), by analysis or by machine learning (Kurtz, 2018; Kurtz, Ammann, Offutt, et al., 2016; Kurtz, Ammann, Delamaro, et al., 2014). DMSG is effective and works well to achieve accuracy when using mutation analysis to benchmark techniques. A DMSG is however created from the information about which mutant(s) are killed by which test(s) after performing a complete mutation analysis. Hence it requires a test suite and it does not reduce the cost of mutation analysis. Therefore DMSGs are not suitable for efficient mutation-based test design. SMSG uses data generated as part of static analysis to approximate the TMSG, however SMSG is illustrated by a smaller example showing that the technique is viable (Kurtz, Ammann, and Offutt, 2015). Machine learning is only recently utilized to identify non-redundant mutants successfully (Kurtz, 2018). All approximation-based approaches have some advantages

as well as limitations. A possible way to find the dominator mutants could therefore be a single process incorporating a combination of static analysis techniques, machine learning and possibly dynamic analysis in a process with multiple steps of either the selection of non-redundant or removal of redundant mutants. These steps would have the purpose of refining the approximation of the set of dominator mutants, without having to first create the redundant mutants or running any tests against them. The resulting mutation analysis would use the *approximated dominator set* (ADS) for a cheaper mutation analysis. The step-wise refinement process could utilize the strength of the different techniques to create the ADS and give a better approximation of the true set of dominator mutants than what can be achieved by using any technique in isolation.

3.1 GOAL

The **aim** for this study is to reduce the overall cost of mutation analysis by minimizing the set of mutants without removing any dominator mutants. The aim is achieved by creating a process to identify subsumption relations between mutants efficiently, resulting in an *approximated dominator set* (ADS). The identified relations can be used to distinguish dominator mutants from other mutants. By reducing the set of mutants to an approximated dominator set, the proposed process will not only keep the effectiveness of mutation analysis without performing a complete mutation analysis, it will also avoid designing tests that only kill redundant mutants. Hence, the resulting test suite will be more cost-effective than it would have been without the proposed approach. A framework implementing the process is also part of the proposed work, as evaluating the research will require automation.

The expected benefit of the process resulting in ADS has limitations. In some cases the amount of non-redundant mutants is as high as 10%, which limits the upper expectations (Papadakis, Henard, et al., 2016; Ammann, Delamaro, and Offutt, 2014). Additionally, it is assumed that the more closely the dominator set would be approximated, the more expensive the process would become. This is assumed because more precise techniques tend to be more expensive. There is a point, where further refinement of the ADS would cost more than using the current approximation. Further reducing the number of redundant mutants is detrimental from cost perspective. Considering the named factors, the trade-off in terms of the reduction and analysis cost needs to be identified as part of the process.

3.1.1 RESEARCH QUESTIONS

Based on the aim, a number of research questions are formulated:

RQ1 - Which existing techniques might be suitable to reduce the number of (redundant) mutants without first creating or executing those mutants?

RQ2 - Which features are important to compare the techniques identified during RQ1?

RQ3 - How can the selected techniques be adapted and combined into a single automatized process to reduce the number of redundant mutants?

RQ4 - How does the cost of mutation analysis using approximated dominator set compare to the cost of complete mutation analysis?

RQ5 - How can we identify the break-point where further refinement of the approximation becomes too expensive?

3.1.2 OBJECTIVES

The aim is fulfilled by completing the following objectives:

1. Identify the candidate techniques, which could be useful in a process to identify mutant subsumption relations relevant to approximate the set of dominator mutants
   a Identify the potentially suitable techniques by surveying key areas
   b Define selection criteria to evaluate the candidates
   c Select the best candidates based on the selection criteria
2. Evaluate the selected candidates with respect to their efficiency and effectiveness in identifying mutant subsumption relations, as well as how well the candidates can interact with the rest of the process
3. Adapt and integrate the chosen techniques into a mutation framework
4. Evaluate the mutation framework with respect to efficiency and effectiveness

There is a number of techniques that might be used to partially approximate the set of dominator mutants. The first objective is therefore to gather the grounded theory for this research. Some techniques are possibly more suitable than others and some may not work with other techniques for some reasons. Therefore the selection criteria should be established to choose the most suitable techniques to achieve the goal of this work.

After selecting the candidate techniques, their effectiveness and efficiency should be evaluated. This step is important as some techniques are possibly better in some regards than others, for example finding different types of dominator or redundant mutants. Some techniques may not work very well together or would result in the identification of the same mutants. Different techniques also work differently well with other techniques or the process itself. Therefore some techniques might be incompatible with each other or the rest of the process. The expenses connected to utilizing different techniques can vary, some candidate techniques may possibly be too expensive.

To be able to evaluate the candidate techniques and the process in practice, a framework is needed. The framework is necessary because mutation analysis needs heavy automation and a framework enables the comparison of using mutation analysis with and without the proposed process. The third objective is therefore to integrate the chosen techniques into a framework. Different techniques may work better under different parts of the process and the order that the techniques can be used can also vary depending on the process and the individual processes. Integration order of the techniques into the process is therefore important for the process to achieve efficiency when approximating the set of dominator mutants.

It is important to evaluate how well the process as a whole is working in terms of efficiency and effectiveness. By evaluating the framework implementing the process, it is possible to measure how well our process fulfills the aim, i.e. if the process is able to approximate the dominator mutants efficiently.

3.2 PRELIMINARY RESULTS

Results are gathered in two fields connected to this work, one focusing on a mutant reduction technique and one focusing on mutation tools.

Two of the publications evaluate the possibilities and the limitations of a special type of static subsumption (Lindström and Márki, 2018; Lindström and Márki, 2016). The reduced ROR (Kaminski, Ammann, and Offutt, 2013; Kaminski, Ammann, and Offutt, 2011) and reduced COR (Just, Kapfhammer, and Schweiggert, 2012) operators were proposed to reduce the number of mutants generated by the two mutation operators without reducing effectiveness, by defining static subsumption relations between the mutants generated by the ROR and COR mutation operators respectively. Our results show however that the reduced set of ROR mutants selected using the proposed static subsumption relation not necessarily subsume all other redundant mutants when strong mutation is utilized (Lindström and Márki, 2016). Additional results displayed that neither of the reduced ROR and COR operators are sufficient in the context of strong mutation if the mutation is executed more than once, e.g. inside a loop (Lindström and Márki, 2018). Results from this journal paper also displayed that in case the mutation is revisited, there is a significant chance that the tests, which are adequate for the reduced ROR and COR set, would not be adequate for all ROR and COR mutants. Therefore, in the context of strong mutation, location for the mutation should be considered before deciding to apply the original or the reduced operator. This decision needs static analysis in advance, which comes with a cost, meaning that the person performing the testing may want to be able to adjust the proposed process for efficiency. Our results concerning the reduced ROR and COR operator show that even though there are some techniques to make mutation analysis considerably cheaper, they need careful consideration in practice to maintain the effectiveness of mutation analysis.

The third publication compares open-source mutation tools for the programming language Java, with the goal to understand the benefits and drawbacks of different tools available (Márki and Lindström, 2017). A tool for mutation analysis is needed to try out the process of mutation analysis with ADS. Therefore we evaluated the existing analysis tools. The results show that mutants are tool dependent, thus the choice of tool will affect the the approximated dominator set.

Results show that there is no clear winner from the three evaluated tools for mutation analysis (Márki and Lindström, 2017). MuJava is effective and it gives the best observability and control but it is considerably slower than the other tools. Major is efficient, effective, and still gives good observability and control over the process, however Major lacks support for Windows. PIT displays high efficiency, at a cost of effectiveness, observability and control. To conclude, Major is expected to work best as a host to utilize ADS for mutation analysis.

The previous publications strengthen the importance of the proposed project. Relations between mutants can be identified and used to remove redundant mutants, as it demonstrated by the reduced ROR and COR operators. However both techniques can lower effectiveness when implemented without considering the location of the mutated faults within the program, as both are prone to mark mutants as redundant false positively in the context of strong mutation. Therefore, both techniques should be combined with other techniques to maintain the effectiveness of mutation analysis. This information however only became apparent after evaluating both techniques in an isolated experimental setup. Also, both reduced ROR and COR only consider relations between mutants from the same operator and applied to the same location. The difference between tools affect how mutants are created, stored, executed and evaluated. The tools available

have different internal implementations, which can affect how easy or difficult it is for some technique to be integrated into the tool. It is expected that even when there are techniques which are beneficial to the mutation analysis process using ADS, integrating these techniques into the tool will require modifications. Based on the previous publication, Major is expected to be a suitable candidate for a framework to implement and refine the process for mutation analysis using ADS.

CHAPTER 4
RESEARCH METHOD AND APPROACH

4.1 THE SCIENTIFIC SEARCH PROCESS BY DESIGN SCIENCE

The suitable method for the proposed research is design science. To be able to create a mutation analysis process using ADS, a number of techniques are needed in combination. The connection between techniques needed to create the ADS is complex and requires an iterative process. As design science is utilized, the main purpose of this work is not to create (or test) new theories.

Design science is suitable as a search process to the creation an IT artifact (Gregor and Hevner, 2013; Oates, 2005; Simon, 1996). Design science is a proactive paradigm that is suitable for problem solving, creation, innovation and creating prescriptive knowledge (Goes, 2014; Gregor and Hevner, 2013). Design science is relevant for improvements, exaptations and innovations (Goes, 2014; Gregor and Hevner, 2013). An improvement is when a known problem is solved with a new solution. Exaptation is when a new problem is solved by adopting already working solutions from other domains. Innovation in this case means using new solutions to solve new problems. Depending both on the solution maturity and the application domain maturity, solutions can in practice overlap between improvement, exaptation and innovation (Gregor and Hevner, 2013).

For this study, it is expected that improvements, exaptation and innovation will all be needed. The overall goal of the study is to improve mutation analysis by making it cheaper and to achieve this goal, underlying techniques may need to be improved too. It is expected that some techniques working in the domain of static and dynamic analysis and also machine learning will be partially suitable to identify mutant subsumption relations, resulting in exaptation. It is also expected that at some points during the project, new problems will arise requiring new solutions, resulting in innovation.

Using design science as a research method defines the search process to create the new, innovative solution. Research using design science usually incorporates the following elements (Goes, 2014; Gregor and Hevner, 2013):

- Introduction of the artifact
- Gathering of relevant descriptive knowledge
- Methods needed for creation of the artifact
- Artifact description
- Artifact evaluation
- Discussion and conclusion

These elements are valid both for individual research publications and for longer ongoing research like dissertations. Independent on the size of the work, research rigour is

always a key factor for successful research science. The previous elements are also used in an iterative manner. For example when the evaluation of the artifact shows that the artifact does not fulfill its purpose, more descriptive knowledge may be needed, methods may change, the artifact may need to be modified, resulting in a different description, which also leads to a need for re-evaluating the artifact. Smaller adjustments are also possible, but some kind of re-evaluation would still be needed. Discussion over the artifact may also vary between the iterative steps.

The **introduction** of the artifact requires the formulation of the problem that the artifact solves alongside the scope of the problem (Goes, 2014). For this work, the introduction translates to this **research proposal**.

The gathering of **descriptive knowledge** as well as already existing prescriptive knowledge is a key element for design science. This section is sometimes also called the literature review section (Goes, 2014). For this work, the literature review section is connected to the first objective, e.g. the identification and the choice of state of art existing techniques that are beneficial to identify subsumption relations to approximate the dominator set. As new techniques are expected to emerge during the time of the proposed research, literature review should be kept up-to-date throughout the project.

The section for the **artifact creation method** is important to create the right artifact in line with the goal of the research. While design science does not have theory testing as an overall goal, it can still incorporate other methodologies for testing theories as well. The method section is closely connected to the second objective of this work, also the evaluation of the effectiveness and efficiency of the chosen techniques. When a new technique is deemed suitable as a candidate for a partial solution to the problem, it needs to be evaluated separately. This evaluation results in either adapting the technique into the mutation analysis process using ADS, or discard the technique as a candidate solution.

The **artifact description** part should contain both a description of the artifact and the design search process itself. For this work, the artifact description is closely connected to objective three, the integration of the chosen techniques into the artifact. After a technique previously is deemed suitable, it needs to be included in the process and adapted to the framework.

The **artifact evaluation** part presents evidence about the usefulness of the artifact. After a new technique is integrated into the mutation analysis using ADS, the refined process needs to be evaluated. When the artifact fulfills the goal of the proposed research, design and science can move towards discussion and conclusion. Otherwise, either more descriptive knowledge should be gathered, or a new technique should be selected to be evaluated.

Discussion and conclusion is important to put gathered results from the creation and evaluation phases in a broader picture, resulting in generalization as well as the definition of possible future work. The design process steps are also usually closely connected to the publication list for the artifact (Goes, 2014; Gregor and Hevner, 2013). For the planned research, the artifact evaluation is connected to the fourth objective, while the complete discussion and evaluation is part of the complete PhD report.

4.2 LITERATURE REVIEW

Design science requires descriptive knowledge to form its grounded theoretical base. To have an thorough understanding of the current knowledge base, a literature survey is commonly used (Goes, 2014). Literature survey is appropriate to gather current knowl-

edge both in depth and width within a domain. Literature review is also useful to ensure that the topic is interesting, the planned research is not just a repetition of someone else's work and that the research results are novel (Oates, 2005). The knowledge base is gathered for objective 1, and will answer research questions RQ1 and RQ2. The literature review is fulfilled when the potentially suitable cost-reduction techniques are identified and the important features for the evaluation of the cost-reduction techniques are defined.

As an approach, a literature survey can be used as a combination of a systematic literature review (SLR) and snowballing. SLR combined with snowballing enables an exhaustive construction of the necessary background knowledge (Jalali and Wohlin, 2012). SLR can be performed on the relevant databases with a number of search strings. Snowballing uses a seed set of relevant articles and either follows up the articles referenced by this seed set of articles (backward snowballing), or the articles referencing the seed set of articles (forward snowballing). Independent of whether SLR or snowballing is used, the identification of relevant articles is similar. With systematic literature review, the number of possibly relevant articles is narrowed down in multiple steps. First, all articles are removed that are not relevant based on their title. Afterwards the abstract is checked for every possibly relevant article. If the possibly relevant articles are still deemed to be relevant, then their introduction and conclusion is evaluated, followed by an evaluation of the complete article. In case an article is still deemed relevant after reading the complete article, it is added to the list of relevant articles. For systematic literature review, the search is done when all articles are checked like this for all search strings on all databases. For snowballing, the search is done when no new relevant papers are found among the referenced or referencing papers. Snowballing is therefore recursive.

The two search patterns SLR and snowballing cover the knowledge domain with two different approaches, thus minimizing the risk of missing some important articles. Documentation of the search criteria alongside with documented limitations makes the literature survey replicable and expendable. Two factors which increase trust in the used approach. For example if the documentation is right, a literature survey can later be extended with new publications by other researchers.

4.3 EXPERIMENT AND CASE STUDY

Design science requires research rigor for both creating and evaluating the artifact through the process (Hevner and Chatterjee, 2010). Depending on the progress with the creation and evaluation of the artifact, different research methods may fit better with the overall design science process.

The proposed research will create a process for mutation analysis with ADS. The process will use a combination of techniques in a step-wise refinement of the ADS. These techniques can and should be evaluated in isolation to better understand their suitability for either removing redundant mutants or identifying dominator mutants and also to compare and contrast the techniques to each other. The process as a whole needs to be evaluated in both a controlled environment and an environment closer to real world application. As it has been previously described, the research process is iterative.

For a controlled (laboratory) environment, experiment is a suitable method (Wohlin et al., 2012). Experiments work well for hypothesis testing and testing of cause - effect pairs in a controlled environment (Oates, 2005). By measuring the effect of independent variables on the dependent variables, effects can measured, and statistical analysis applied (Wohlin et al., 2012; Oates, 2005).

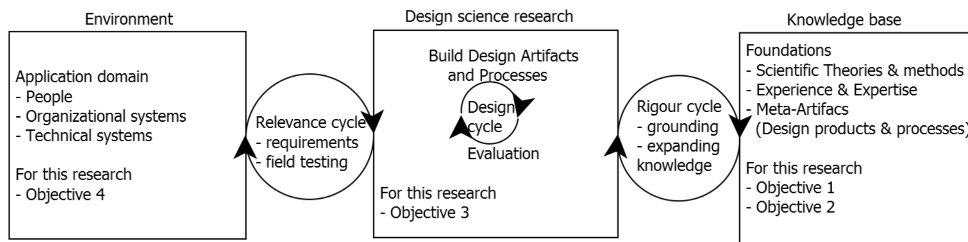

Figure 4.1: The design science process.

Case studies are observational studies (Wohlin et al., 2012). A case study performed within design science focuses on the artifact or some aspect of the artifact in its own context (Oates, 2005; Runeson and Höst, 2009). Performing a case study compared with an experiment result in less control in terms of the observation of cause and effect, but it enables evaluation of phenomena in a more natural setting.

For this research several experiments and at least one case study are needed to fulfill objective 2, 3 and 4. The planned experiments and case studies enable research questions RQ3 and RQ4 to be answered by giving insight on how the different techniques can be combined and how effective the proposed process is. Additionally, research question RQ5 is expected to be answered by evaluating the overall cost of mutation analysis using ADS to determine when performing mutation analysis on the rest of the mutants is more economical than trying to better approximate the set of dominator mutants.

4.4 THE COMBINATION OF THE METHODS

Figure 4.1 gives an overview of the process and connects design science with the proposed work (Hevner, 2007). The rigour cycle maps against section 4.2. The Relevance cycle maps against the case study part of section 4.3. The design cycle maps against the experiments in section 4.3 in general.

The knowledge base is connected to the first two objectives. The knowledge base about the techniques is created based on a literature survey. The survey identifies candidate techniques to identify mutation subsumption relations as well as the selection criteria that should be used to evaluate the candidate techniques. This results in the experience and expertise needed to create the process of mutation analysis using ADS. The second objective is about the meta-artifacts, e.g the candidates are selected and evaluated on their own with respect to their effectiveness and efficiency to identify subsumption relations. This is also done by smaller scale experiments evaluating also how the techniques can work in isolation and with each other. This includes for example which data the different techniques need, what they produce and if they have some limitations, for example depending on some logical operators or loops. Techniques being able to classify dominator mutants more precisely than other techniques can also be more expensive. Some techniques might be redundant with regard to other techniques because they provide the same information. Some techniques might need data which can be difficult to provide. Therefore, having a good understanding on which techniques there are and how they compare to each other are important for this project.

The design cycle maps to building the artifact, including adaptation and integration of the chosen techniques e.g. the third objective. The different techniques are evaluated

here as part of the process for mutation analysis using dominator set. During a design cycle it can also happen that promising techniques would not work very well with the rest of the mutation analysis process using ADS. This new knowledge will then interact with the knowledge base, expanding the knowledge on how the next technique would be selected. The creation and the evaluation of the process for mutation analysis with ADS needs to be step-wise, as it is not known in advance which practical limitations and possibilities can arise during the design cycle.

The environment part in figure 4.1 is connected to the fourth objective via the relevance cycle. While smaller scale evaluations are needed for the design science research and the knowledge base part, eventually the process should be field tested against a larger, more generalizable problem. This requires a case study on a larger system comparing mutation analysis with all mutants and with the mutants selected by the process. This way, effectiveness and efficiency of using the process will be evaluated compared to not using the process, and whether further refinement is necessary.

CHAPTER 5
RESEARCH PLAN

This chapter contains the proposed time line of the project.

The research plan follows the steps associated by design science (Goes, 2014; Gregor and Hevner, 2013), with figure 5.1 giving an overview.

The project proposal acts as an initial introduction of the artifact. As such, a publication is planned based on the content of the research proposal.

Descriptive knowledge is required to fulfill objective 1, answer research question RQ1 and RQ2, and also to identify and select suitable techniques to approximate the dominator set. Descriptive knowledge is gathered during the initial phase of the process and then kept up-to-date. The descriptive knowledge need to cover mutation analysis, static and dynamic analysis techniques and machine learning. A systematic literature survey is useful to gather the existing knowledge base, but it is an unnecessary step if recent systematic literature survey papers of quality already exists or if a technique is very recent and there are no or few papers available. No separate publication in form of literature review is planned, although the method is still important for this research.

For mutation testing, there is a recent survey paper of high quality (Papadakis, Kintis, et al., 2019). This paper is based on previous survey papers in the area of mutation analysis (Jia and Harman, 2011; Offutt, 2011; Offutt and Untch, 2001). Backward snowballing on all relevant references of (Papadakis, Kintis, et al., 2019) is done as part of writing this research proposal, and forward snowballing is done periodically by following key researchers in the area as well as monitoring key venues and journals in the area such as MUTATION, ICST, STVR and ICSE.

A static or dynamic analysis based technique is only interesting for the research if it can help to approximate the set of dominator mutants. As part of writing this research proposal, an initial search utilizing forward and backward snowballing is performed based on two survey papers (Anand, Burke, et al., 2013; McMinn, 2004). Other more recent publications are identified, however not yet surveyed (Baldoni et al., 2018; Su et al., 2017; Wong et al., 2016). Snowballing on the more recent survey papers is planned, as well as monitoring of the literature for advances in the field.

Machine learning is the third topic and is an active field. Machine learning was recently added to the study and therefore the literature review is in a very early stage on the topic at the time of this research proposal. Some survey papers are identified in the area and snowballing is planned focusing on techniques which can help to create the process of mutation analysis using ADS. The advances in the field of machine learning are planned to be monitored. Machine learning is recently utilized successfully to approximate the dominator set (Kurtz, 2018). No other publications are known at the writing of this research proposal in the combined area of machine learning and mutation testing.

The methods needed for the creation of the artifact translate into a series of experiments using different techniques, connected to objective 2 and answering research question RQ3. Two experiments are already performed and published. The first experiment re-

Activity	Design science connection	2018		2019		2020		2021		2022		2023		2024		2025	
Courses																	
Objective1	Gathering descriptive knowledge																
Objective2	Artifact creation method																
Objective3	Artifact description																
Objective4	Artifact evaluation																
Research proposal	Project introduction																
Thesis proposal	(Artifact description)																
Final thesis	All of the previous + discussion																
Final seminar	& conclusion																

Figure 5.1: Research plan timeline

sulted in a better understanding of the reduced ROR and COR technique (Lindström and Márki, 2018; Lindström and Márki, 2016). The second experiment resulted in the knowledge on which tool for mutation analysis would be suitable to serve as a framework for the process of mutation analysis with ADS (Márki and Lindström, 2017). Two additional sets of experiments are planned, and both sets connect to objective 2 and 3. The first set of experiment evaluates a set of static and dynamic analysis techniques. The second set of experiment evaluates different machine learning techniques.

The artifact description needs to be finalized halfway through the project when using design science, otherwise the time is usually not sufficient to perform the necessary studies using the artifact. This is considered in the plan and the thesis proposal is planned accordingly. The thesis proposal covers the description of the artifact in creation. Thus, fulfilling objective 4 and partially answering research question RQ4 and RQ5. After the thesis proposal, the work focuses on the evaluation and integration of machine learning techniques and their integration into the artifact. This may require expanding the necessary descriptive knowledge. New emerging techniques might also be identified, evaluated or adapted.

When the iterative refinement and evaluation of the artifact is finished and the goal of the artifact is reached fulfilling objective 4 by comparing mutation analysis using ADS to mutation analysis using all mutants and also mutation analysis using (approximated) TMSG, the work is planned to focus on remaining publications and writing the doctoral dissertation. At this point, research question RQ4 and RQ5 are fully answered.

The publication plan is set according to the progression of the work. The publication plan also contains courses taken as they are required for the degree. Publications are planned connected to the following activities:

- Project introduction: 2019 Q4 (research proposal)
- Artifact creation method: 2017 Q2 (Lindström and Márki, 2018), 2018 Q1 (Márki and Lindström, 2017), 2020 Q4 (static and dynamic analysis based techniques using experiment), 2023 Q1 (machine learning based techniques using experiment), 2024 Q1 (techniques in combination using case study)
- Artifact description: 2021 Q1 (thesis proposal)
- Artifact evaluation: 2025 Q3 (doctoral dissertation)

CHAPTER 6
EXPECTED RESULTS

The aim for this study is to make mutation analysis more efficient without reducing its effectiveness, by integrating cost reduction techniques into a single process where mutation analysis is used with ADS. The expected results can therefore be divided into the objectives targeting the identification of the grounded theory, the research done on the individual techniques to achieve the process, the process itself with combined techniques and the artifact as an instantiation of the process.

The following results are expected out of the proposed research:

1. An overview of the techniques useful to approximate the set of dominator mutants
2. An evaluation of comparing and contrasting cost reduction techniques
3. A process for mutation analysis using ADS
4. An artifact where techniques are combined to achieve mutation analysis with ADS
5. An evaluation of the tool and the framework in a larger scale study

Result 1 is a contribution in terms of a better understanding of the techniques on their own and in combination. Result 2 is closely connected and is expected to result in a better understanding of the techniques, how they can be used on their own and in combination. Similarly to the previous publication on the reduced ROR and COR technique (Lindström and Márki, 2018; Lindström and Márki, 2016), publications are expected from this research using the combination of mutation analysis, static and dynamic subsumption and machine learning.

Result 3 is the process itself to achieve mutation analysis using ADS, which is expected to be published at least in two iterations as part of the thesis proposal and the doctoral dissertation. Result 4 is the artifact incorporating the techniques for mutation analysis using ADS. The artifact will be using Major based on previous findings (Márki and Lindström, 2017).

Result 5 is an evaluation of artifact implementing the combined techniques to achieve mutation analysis using ADS. This will be presented as part of the thesis proposal and the doctoral dissertation. The evaluation is expected to be a comparison with mutation analysis using all mutants and dominator mutants based on TMSG approximated by different method. The expected result is an overview on how well the proposed combined process holds in terms of efficiency and effectiveness, as well as a better understanding on how well the different techniques can work in conjunction.

CHAPTER 7
RELATED WORK

Mutation analysis is and always was considered a costly testing technique through the years, resulting in a plethora of approaches to lower the cost of the technique dating back before the discovery of subsumption between mutants (Ferrari, Pizzoleto, and Offutt, 2018; Papadakis, Kintis, et al., 2019; Jia and Harman, 2011; Offutt and Untch, 2001). This chapter focuses only on the relevant related work classified as the reduction of the number of mutants, including constrained mutation, mutant sampling, higher order mutation, subsumption, symbolic execution as well as combination of the techniques (Ferrari, Pizzoleto, and Offutt, 2018).

The number of mutants can be systematically reduced if less mutation operators are used. Constrained mutation uses fewer mutation operators for the mutation analysis, thus reducing the number of mutants created. Selective mutation omits the operators that either create a large number of mutants or a large percentage of equivalent mutants. Empirical results show that test suites that are 100% adequate for the five mutation operators {ABS, AOR, LCR, ROR, UOI} also detect the majority of the mutants generated by the full set of 22 mutant operators originally implemented for the Mothra system. Dominator mutants were however not considered and there is no relation between mutation operator and dominator mutants (Offutt, A. Lee, et al., 1996). The set of {ABS, AOR, LCR, ROR, UOI} is implemented and used in MuJava (Ma, Offutt, and Kwon, 2005) and Major (Just, 2014). The full set of mutation operators implemented by different tools can vary, which affects the set of mutants created (Kintis, Papadakis, Papadopoulos, et al., 2018; Márki and Lindström, 2017). The work proposed in this research proposal focuses on removing the redundant mutants instead of removing mutation operators. All mutation operators might produce dominator mutants. As all dominator mutants should be kept for effectiveness, this work does not utilize selective mutation.

Mutant sampling uses all available mutation operators, but instead of using every mutant, only a percentage of all mutants are selected to be used. Mutation sampling was used ever since the first working mutation analysis tool, as mutation sampling is a scalable and easy to implement and can greatly increase efficiency (J. Zhang, Zhu, Hao, and L. Zhang, 2014; DeMillo, Guindi, et al., 1988). However, recent advances identifying 90-99% redundancy among mutants indicate that randomly selecting a number of mutants can result in reduced effectiveness (Papadakis, Henard, et al., 2016; Kurtz, Ammann, Offutt, et al., 2016; Kurtz, Ammann, and Offutt, 2015; Kurtz, Ammann, Delamaro, et al., 2014; Ammann, Delamaro, and Offutt, 2014). As effectiveness can be considerably lower and the technique generally does not help to approximate the set of dominator mutants, mutation sampling is not considered for this work.

Higher order mutation (HOM) uses the idea to combine multiple first order mutants into one higher order mutant (Harman, Jia, and Langdon, 2010). With other words, higher order mutants contain a number of syntactical changes instead just one. This approach reduces the number of mutants to be covered by each test. The data from the mutant kill matrix can be used to create higher order mutants that subsume the first order mutants, which they are constructed of. Such mutants are referred as super mutants (Gopinath,

Mathis, and Zeller, 2018). HOM has however problems with fault masking and does not help distinguishing dominator mutants from redundant mutants. Therefore HOM is not considered for this work.

Mutant subsumption exploits the relation between mutants. Dominator mutants exist as a subset of all mutants so that if a test suite can detect all dominator mutants, the same test suite is guaranteed to detect the complete set of mutants. Static subsumption relations are rule based and depend on the mutation operator as well as the original syntactical element, which they were applied to (Kaminski, Ammann, and Offutt, 2013; Just, Kapfhammer, and Schweiggert, 2012; Kaminski, Ammann, and Offutt, 2011). Static subsumption makes it possible to avoid creation of subsumed mutants. However such relations are only defined for the COR and ROR mutation operators, but not to others and only between mutants within the same mutated element (Kaminski, Ammann, and Offutt, 2013; Just, Kapfhammer, and Schweiggert, 2012). Additionally, the subsumption relations for ROR only hold in the context of weak mutation or when the syntactically changed element cannot be revisited during execution (Lindström and Márki, 2018). Dynamic subsumption is needed to construct the dynamic mutant subsumption graph (DMSG), which is an approximation of the True Mutant Subsumption Graph (TMSG) (Kurtz, Ammann, Offutt, et al., 2016; Kurtz, Ammann, and Offutt, 2015; Kurtz, Ammann, Delamaro, et al., 2014). The difference between DMSG and TMSG is that DMSG is minimal with respect to the current test suite, while TMSG is minimal with respect to any test suite. DMSGs are created based which mutant was killed by which test after executing all tests on all mutants, i.e. after performing a complete mutation analysis, which means that it cannot be used to reduce the cost and it cannot be used for test design since a test suite needs to be in place. DMSGs can be utilized as means to counter redundancy related validity threats in benchmark studies. Static analysis has been utilized to create static mutant subsumption graph (SMSG) and to approximate the a statically-derived DMSG (SDMSG) from the information derived from the static analysis (Kurtz, Ammann, and Offutt, 2015). The SDMSG approximates DMSG closely on a small-scale solution, however large-scale validation is still needed (Kurtz, Ammann, and Offutt, 2015). Instead of trying to identify the dominator mutants, techniques can also identify and remove redundant mutants. The lack of semantic difference between mutants can be utilized to mark mutants as redundant to each other, meaning that some of them can be removed without losing effectiveness. Trivial compiler equivalence (TCE) considers machine code to remove mutants equivalent to the original and to identify redundant mutants (Kintis, Papadakis, Jia, et al., 2017). Both techniques for identification of dominator mutants and for removal of redundant mutants are important for this work. None of the above approaches provide a solution to the proposed work, but they can still be useful as partial solutions in the proposed framework.

Symbolic execution and control flow analysis can be used to limit the combination of mutants and tests used, i.e. distinguishing test cases can be created (Kurtz, Ammann, and Offutt, 2015; Wotawa, Nica, and Aichernig, 2010; DeMillo and Offutt, 1991). Such distinguishing test cases are related to path coverage in a way that if a test cannot reach a mutant, it would not be able to kill it (Anand, Păsăreanu, and Visser, 2007). Dynamic domain reduction (DDR) is an extension to dynamic symbolic execution (DSE) to constraint input variables depending on the program branches (Offutt, Jin, and Pan, 1999). An implementation is created for the Fortran based Mothra system, but that implementation has scalability limitations (Offutt, Jin, and Pan, 1999). The evolution of static analysis, DSE and DDR, combined with the use of random values resulted in increased scalability of the technique (Godefroid, Klarlund, and Sen, 2005). Constraint solvers are required to create distinguishing test cases, and to achieve branch coverage (Anand, Burke, et al., 2013), which is important to reach all the mutants. Concolic testing is a

combination where DSE and constraint solvers are utilized together to achieve scalability (Sen, Marinov, and Agha, 2005). The location of the mutants can also be used to identify equivalent mutants either by slicing or considering path conditions (Hierons, Harman, and Danicic, 1999; Offutt and Pan, 1997). We intend to approximate the set of dominator mutants without performing a full mutation analysis and both symbolic execution and control flow analysis techniques are considered at this point to be potential candidates to contribute to this aim.

Different techniques can also be used in combination to achieve cost reduction. Static analysis has been combined with other approaches previously. Path aware mutation sampling shows better effectiveness compared to mutation sampling (Sun, F. Xue, Liu, and X. Zhang, 2017). Machine learning techniques based on the mutants' locations can also be combined with mutation sampling to create fault revealing mutants, increasing the effectiveness of random selection (Chekam, Papadakis, Bissyandé, et al., 2018; Chekam, Papadakis, Le Traon, and Harman, 2017). Branch coverage can be used to reduce the number of mutants by identifying subsumption relation between the branches (Gong, G. Zhang, Yao, and Meng, 2017; Papadakis and Malevris, 2011). Differentiating mutants can also be achieved using simplified Jimple code, control-flow-graph representation and code coverage data as implemented in MuRanker (Namin, X. Xue, Rosas, and Sharma, 2015). An implementation of symbolic execution, Directed Incremental Symbolic Execution (DiSE) was shown to be useful to approximate TMSG, however only on a small scale example (Kurtz, Ammann, and Offutt, 2015). Machine learning based on features like the mutation operator, the mutation and some abstract syntax tree values is useful to predict which mutants are equivalent or non-redundant (Kurtz, 2018). The approach of MuRanker is theorized to help with the estimation of the set of dominator mutants, however it only presents a partial solution on its own. Our work plans to utilize symbolic execution in combination of subsumption based techniques. Furthermore, some of the combined work is only validated for smaller scale programs, while our goal is to have a scalable process.

To sum it up, the techniques presented in this chapter use different approaches to make mutation analysis more efficient. The used research methods in the referred studies are usually experiments or case studies, but a literature survey paper is also present. The main difference between the listed research and the proposed work is that the proposed work will define a process and design a framework where a combination of techniques can be utilized to achieve cost-effective mutation-based test design.

REFERENCES

Ammann, Paul, Delamaro, Márcio Eduardo, and Offutt, Jeff (2014). “Establishing theoretical minimal sets of mutants.” In: *Proceedings of the 7th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, pp. 21–30.
Ammann, Paul and Offutt, Jeff (2016). *Introduction to software testing*. 2nd ed. Cambridge: Cambridge University Press.
Anand, Saswat, Burke, Edmund K., Chen, Tsong Yueh, Clark, John, Cohen, Myra B., Grieskamp, Wolfgang, Harman, Mark, Harrold, Mary Jean, and McMinn, Phil (2013). “An orchestrated survey of methodologies for automated software test case generation.” In: *Journal of Systems and Software* 86.8, pp. 1978–2001. ISSN: 0164-1212. DOI: `https://doi.org/10.1016/j.jss.2013.02.061`.
Anand, Saswat, Păsăreanu, Corina S., and Visser, Willem (2007). “JPF–SE: A Symbolic Execution Extension to Java PathFinder.” In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Orna Grumberg and Michael Huth. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 134–138. ISBN: 978-3-540-71209-1.
Baldoni, Roberto, Coppa, Emilio, D’elia, Daniele Cono, Demetrescu, Camil, and Finocchi, Irene (2018). “A survey of symbolic execution techniques.” In: *ACM Computing Surveys (CSUR)* 51.3, p. 50.
Boyer, Robert S, Elspas, Bernard, and Levitt, Karl N (1975). “SELECT—a formal system for testing and debugging programs by symbolic execution.” In: *ACM SigPlan Notices* 10.6, pp. 234–245.
Budd, Timothy A., DeMillo, Richard A., Lipton, Richard J., and Sayward, Frederick G. (1978). “The design of a prototype mutation system for program testing.” In: *Proceedings of the AFIPS National Computer Conference*. Vol. 74, pp. 623–627.
– (1980). “Theoretical and empirical studies on using program mutation to test the functional correctness of programs.” In: *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, pp. 220–233.
Chekam, Thierry Titcheu, Papadakis, Mike, Bissyandé, Tegawendé, Traon, Yves Le, and Sen, Koushik (2018). “Selecting Fault Revealing Mutants.” In: *arXiv preprint arXiv:1803.07901*.
Chekam, Thierry Titcheu, Papadakis, Mike, Le Traon, Yves, and Harman, Mark (2017). “An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption.” In: *39th International Conference on Software Engineering*, pp. 597–608.
Coles, Henry, Laurent, Thomas, Henard, Christopher, Papadakis, Mike, and Ventresque, Anthony (2016). “PIT: a practical mutation testing tool for java.” In: *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA)*. ACM, pp. 449–452.
Delahaye, Mickaël and Du Bousquet, Lydie (2013). “A comparison of mutation analysis tools for java.” In: *Proceedings of the 13th international conference on Quality software (QSIC)*. IEEE, pp. 187–195.
Delamaro, Márcio Eduardo, Maldonado, José Carlos, and Vincenzi, Auri Marcelo Rizzo (2001). “Proteum/IM 2.0: An integrated mutation testing environment.” In: *Mutation testing for the new century*. Springer, pp. 91–101.
DeMillo, Richard A., Guindi, Dany S., McCracken, W. M., Offutt, Jeff, and King, Kim N. (1988). “An extended overview of the Mothra software testing environment.” In:

Proceedings of the Second Workshop on Software Testing, Verification, and Analysis, 1988. IEEE, pp. 142–151.
DeMillo, Richard A., Krauser, Edward W., and Mathur, Aditya P (1991). "Compiler-integrated program mutation." In: *[1991] Proceedings The Fifteenth Annual International Computer Software & Applications Conference*. IEEE, pp. 351–356.
DeMillo, Richard A., Lipton, Richard J., and Sayward, Frederick G. (1978). "Hints on test data selection: Help for the practicing programmer." In: *Computer* 11.4, pp. 34–41.
DeMillo, Richard A. and Offutt, Jeff (1991). "Constraint-based automatic test data generation." In: *IEEE Transactions on Software Engineering* 17.9, pp. 900–910.
Denisov, Alex and Pankevich, Stanislav (2018). "Mull it over: mutation testing based on LLVM." In: *13th International Workshop on Mutation Analysis (MUTATION 2018)*. IEEE, pp. 25–31.
Derezińska, Anna and Szustek, Anna (2009). "Object-oriented testing capabilities and performance evaluation of the C# mutation system." In: *IFIP Central and East European Conference on Software Engineering Techniques*. Springer, pp. 229–242.
Ferrari, Fabiano Cutigi, Pizzoleto, Alessandro Viola, and Offutt, Jeff (2018). "A Systematic Review of Cost Reduction Techniques for Mutation Testing: Preliminary Results." In: *13th International Workshop on Mutation Analysis (MUTATION 2018)*. IEEE.
Frankl, Phyllis G., Weiss, Stewart N., and Hu, Cang (1997). "All-uses vs mutation testing: an experimental comparison of effectiveness." In: *Journal of Systems and Software* 38.3, pp. 235–253.
Frankl, Phyllis G. and Weyuker, Elaine Jessica (1988). "An applicable family of data flow testing criteria." In: *IEEE Transactions on Software Engineering* 14.10, pp. 1483–1498.
Fraser, Gordon and Zeller, Andreas (2012). "Mutation-driven generation of unit tests and oracles." In: *IEEE Transactions on Software Engineering* 38.2, pp. 278–292.
Gates, Bill (2002). *Trustworthy computing*.
Godefroid, Patrice, Klarlund, Nils, and Sen, Koushik (2005). "DART: directed automated random testing." In: *ACM Sigplan Notices*. Vol. 40. 6. ACM, pp. 213–223.
Goes, Paulo B (2014). "Design science research in top information systems journals." In: *MIS Quarterly: Management Information Systems* 38.1, pp. iii–viii.
Gong, Dunwei, Zhang, Gongjie, Yao, Xiangjuan, and Meng, Fanlin (2017). "Mutant reduction based on dominance relation for weak mutation testing." In: *Information and Software Technology* 81, pp. 82–96. ISSN: 0950-5849. DOI: `https://doi.org/10.1016/j.infsof.2016.05.001`.
Gopinath, Rahul, Mathis, Björn, and Zeller, Andreas (2018). "If You Can't Kill a Supermutant, You Have a Problem." In: *13th International Workshop on Mutation Analysis (MUTATION 2018)*. IEEE, pp. 18–24.
Gregor, Shirley and Hevner, Alan R. (2013). "Positioning and presenting design science research for maximum impact." In: *MIS quarterly* 37.2.
Harman, Mark, Jia, Yue, and Langdon, William B (2010). "A manifesto for higher order mutation testing." In: *Third International Conference on Software Testing, Verification, and Validation Workshops*. IEEE, pp. 80–89.
Harrold, Mary Jean (2000). "Testing: a roadmap." In: *Proceedings of the Conference on the Future of Software Engineering*. ACM, pp. 61–72.

Hevner, Alan R. (2007). "A three cycle view of design science research." In: *Scandinavian journal of information systems* 19.2, p. 4.
Hevner, Alan R. and Chatterjee, Samir (2010). "Design science research in information systems." In: *Design research in information systems*. Springer, pp. 9–22.
Hierons, Rob, Harman, Mark, and Danicic, Sebastian (1999). "Using program slicing to assist in the detection of equivalent mutants." In: *Software Testing Verification and Reliability* 9.4, pp. 233–262.
Howden, William E. (1975). "Methodology for the generation of program test data." In: *IEEE Transactions on computers* 100.5, pp. 554–560.
– (1982). "Weak Mutation Testing and Completeness of Test Sets." In: *IEEE Transactions on Software Engineering* SE-8.4, pp. 371–379. ISSN: 0098-5589. DOI: 10.1109/TSE.1982.235571.
Jalali, Samireh and Wohlin, Claes (2012). "Systematic literature studies: database searches vs. backward snowballing." In: *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*. ACM, pp. 29–38.
Jia, Yue and Harman, Mark (2008a). "Constructing subtle faults using higher order mutation testing." In: *Proceedings of the 8th IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE, pp. 249–258.
– (2008b). "MILU: A customizable, runtime-optimized higher order mutation testing tool for the full C language." In: *Practice and Research Techniques, 2008. TAIC PART'08. Testing: Academic & Industrial Conference*. IEEE, pp. 94–98.
– (2011). "An analysis and survey of the development of mutation testing." In: *IEEE transactions on software engineering* 37.5, pp. 649–678.
Juristo, Natalia, Moreno, Ana M., and Vegas, Sira (2004). "Reviewing 25 years of testing technique experiments." In: *Empirical Software Engineering* 9.1-2, pp. 7–44.
Just, René (2014). "The Major mutation framework: Efficient and scalable mutation analysis for Java." In: *Proceedings of the 24th International Symposium on Software Testing and Analysis (ISSTA)*. ACM, pp. 433–436.
Just, René, Kapfhammer, Gregory M, and Schweiggert, Franz (2012). "Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis." In: *Proceedings of the 23th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, pp. 11–20.
Just, René and Schweiggert, Franz (2015). "Higher accuracy and lower run time: efficient mutation analysis using non-redundant mutation operators." In: *Software Testing, Verification and Reliability* 25.5-7, pp. 490–507. ISSN: 1099-1689. DOI: 10.1002/stvr.1561.
Kakarla, Sahitya, Momotaz, Selina, and Namin, Akbar Siami (2011). "An evaluation of mutation and data-flow testing: A meta-analysis." In: *Proceedings of the 4th International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, pp. 366–375.
Kaminski, Gary, Ammann, Paul, and Offutt, Jeff (2011). "Better Predicate Testing." In: *Proceedings of the 6th International Workshop on Automation of Software Test*. AST '11. Waikiki, Honolulu, HI, USA: ACM, pp. 57–63. ISBN: 978-1-4503-0592-1. DOI: 10.1145/1982595.1982608.
– (2013). "Improving logic-based testing." In: *Journal of Systems and Software* 86.8, pp. 2002–2012. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2012.08.024.

King, James C. (1976). "Symbolic Execution and Program Testing." In: *Commun. ACM* 19.7, pp. 385–394. ISSN: 0001-0782. DOI: 10.1145/360248.360252.

Kintis, Marinos, Papadakis, Mike, Jia, Yue, Malevris, Nicos, Le Traon, Yves, and Harman, Mark (2017). "Detecting Trivial Mutant Equivalences via Compiler Optimisations." In: *IEEE Transactions on Software Engineering* PP.99, pp. 1–1. ISSN: 0098-5589. DOI: 10.1109/TSE.2017.2684805.

Kintis, Marinos, Papadakis, Mike, Papadopoulos, Andreas, Valvis, Evangelos, Malevris, Nicos, and Le Traon, Yves (2018). "How effective are mutation testing tools? An empirical analysis of Java mutation testing tools with manual analysis and real faults." In: *Empirical Software Engineering* 23.4, pp. 2426–2463.

Kurtz, Bob (2018). "Improving mutation testing with dominator mutants." PhD thesis. George Mason University.

Kurtz, Bob, Ammann, Paul, Delamaro, Márcio Eduardo, Offutt, Jeff, and Deng, Lin (2014). "Mutant subsumption graphs." In: *9th International Workshop on Mutation Analysis (MUTATION 2014)*. IEEE, pp. 176–185.

Kurtz, Bob, Ammann, Paul, and Offutt, Jeff (2015). "Static analysis of mutant subsumption." In: *10th International Workshop on Mutation Analysis (MUTATION 2015)*. IEEE, pp. 1–10.

Kurtz, Bob, Ammann, Paul, Offutt, Jeff, Delamaro, Márcio E, Kurtz, Mariet, and Gökçe, Nida (2016). "Analyzing the validity of selective mutation with dominator mutants." In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, pp. 571–582.

Laurent, Thomas, Papadakis, Mike, Kintis, Marinos, Henard, Christopher, Le Traon, Yves, and Ventresque, Anthony (2017). "Assessing and improving the mutation testing practice of PIT." In: *Proceedings of the 10th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, pp. 430–435.

Li, Nan and Offutt, Jeff (2017). "Test oracle strategies for model-based testing." In: *IEEE Transactions on Software Engineering* 43.4, pp. 372–395.

Li, Nan, Praphamontripong, Upsorn, and Offutt, Jeff (2009). "An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage." In: *Proceedings of the 2th International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, pp. 220–229.

Lindström, Birgitta and Márki, András (2016). "On strong mutation and subsuming mutants." In: *11th International Workshop on Mutation Analysis (MUTATION 2016)*. IEEE, pp. 112–121.

– (2018). "On strong mutation and the theory of subsuming logic-based mutants." In: *Software Testing, Verification and Reliability* 29.1-2. DOI: 10.1002/stvr.1667.

Ma, Yu-Seung, Offutt, Jeff, and Kwon, Yong Rae (2005). "MuJava: An automated class mutation system." In: *Software Testing, Verification and Reliability* 15.2, pp. 97–133.

Márki, András and Lindström, Birgitta (2017). "Mutation Tools for Java." In: *Proceedings of the Symposium on Applied Computing*. SAC '17. Marrakech, Morocco: ACM, pp. 1364–1415. ISBN: 978-1-4503-4486-9.

McMinn, Phil (2004). "Search-based software test data generation: a survey." In: *Software Testing, Verification and Reliability* 14.2, pp. 105–156.

Morell, Larry J. (1990). "A theory of fault-based testing." In: *IEEE Transactions on Software Engineering* 16.8, pp. 844–857.

Myers, Glenford J, Badgett, Tom, Thomas, Todd M, and Sandler, Corey (2004). *The art of software testing*. 2nd ed. New Jersey: John Wiley & Sons, Inc.
Namin, Akbar Siami, Xue, Xiaozhen, Rosas, Omar, and Sharma, Pankaj (2015). "MuRanker: a mutant ranking tool." In: *Software Testing, Verification and Reliability* 25.5-7, pp. 572–604. ISSN: 1099-1689. DOI: 10.1002/stvr.1542.
Oates, Briony J. (2005). *Researching information systems and computing*. Sage.
Offutt, Jeff (2011). "A mutation carol: Past, present and future." In: *Information and Software Technology* 53.10, pp. 1098–1107.
Offutt, Jeff, Jin, Zhenyi, and Pan, Jie (1994). "The dynamic domain reduction approach for test data generation: Design and algorithms." In: *George Mason University, Fairfax, Virginia, Technical Report ISSE-TR-94-110*.
– (1999). "The dynamic domain reduction procedure for test data generation." In: *Software-Practice and Experience* 29.2, pp. 167–93.
Offutt, Jeff, Lee, Ammei, Rothermel, Gregg, Untch, Roland H., and Zapf, Christian (1996). "An Experimental Determination of Sufficient Mutant Operators." In: *ACM Tranactions on Software Engineering and Methodology (TOSEM)* 5.2, pp. 99–118. ISSN: 1049-331X. DOI: 10.1145/227607.227610.
Offutt, Jeff and Lee, Stephen D. (1991). "How strong is weak mutation?" In: *Proceedings of the Symposium on Testing, Analysis, and Verification*, pp. 200–213.
Offutt, Jeff and Pan, Jie (1997). "Automatically detecting equivalent mutants and infeasible paths." In: *Software testing, verification and reliability* 7.3, pp. 165–192.
Offutt, Jeff and Untch, Roland H. (2001). "Mutation 2000: Uniting the orthogonal." In: *Mutation testing for the new century*. Ed. by Ahmed K. Elmagarmid and Amit Sheth. Springer, pp. 34–44.
Offutt, Jeff, Voas, Jeff, and Payne, Jeff (1996). *Mutation operators for Ada*. Tech. rep. Technical Report ISSE-TR-96-09, Information and Software Systems Engineering, George Mason University.
Papadakis, Mike, Chekam, Thierry Titcheu, and Le Traon, Yves (2018). "Mutant Quality Indicators." In: *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 32–39.
Papadakis, Mike, Henard, Christopher, Harman, Mark, Jia, Yue, and Le Traon, Yves (2016). "Threats to the Validity of Mutation-based Test Assessment." In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ISSTA 2016. Saarbrücken, Germany: ACM, pp. 354–365. ISBN: 978-1-4503-4390-9. DOI: 10.1145/2931037.2931040.
Papadakis, Mike, Jia, Yue, Harman, Mark, and Le Traon, Yves (2015). "Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique." In: *Proceedings of the 37th IEEE International Conference on Software Engineering*. Vol. 1, pp. 936–946. DOI: 10.1109/ICSE.2015.103.
Papadakis, Mike, Kintis, Marinos, Zhang, Jie, Jia, Yue, Le Traon, Yves, and Harman, Mark (2019). "Mutation testing advances: an analysis and survey." In: *Advances in Computers*. Vol. 112. Elsevier, pp. 275–378.
Papadakis, Mike and Malevris, Nicos (2011). "Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing." In: *Software Quality Journal* 19.4, p. 691. ISSN: 1573-1367. DOI: 10.1007/s11219-011-9142-y.

Petrovic, Goran and Ivankovic, Marko (2018). “State of Mutation Testing at Google.” In: *Proceedings of the International Conference on Software Engineering—Software Engineering in Practice (ICSE SEIP)*.

Phan, Duy Loc, Kim, Yunho, and Kim, Moonzoo (2018). “MUSIC: Mutation Analysis Tool with High Configurability and Extensibility.” In: *13th International Workshop on Mutation Analysis (MUTATION 2018)*.

Runeson, Per and Höst, Martin (2009). “Guidelines for conducting and reporting case study research in software engineering.” In: *Empirical software engineering* 14.2, p. 131.

Sen, Koushik, Marinov, Darko, and Agha, Gul (2005). “CUTE: a concolic unit testing engine for C.” In: *ACM SIGSOFT Software Engineering Notes*. Vol. 30. 5. ACM, pp. 263–272.

Simon, Herbert A (1996). *The sciences of the artificial*. 3rd ed. MIT press.

Su, Ting, Wu, Ke, Miao, Weikai, Pu, Geguang, He, Jifeng, Chen, Yuting, and Su, Zhendong (2017). “A survey on data-flow testing.” In: *ACM Computing Surveys (CSUR)* 50.1, p. 5.

Sun, Chang-ai, Xue, Feifei, Liu, Huai, and Zhang, Xiangyu (2017). “A path-aware approach to mutant reduction in mutation testing.” In: *Information and Software Technology* 81, pp. 65–81. ISSN: 0950-5849. DOI: `https://doi.org/10.1016/j.infsof.2016.02.006`.

Untch, Roland H., Offutt, Jeff, and Harrold, Mary Jean (1993). “Mutation analysis using mutant schemata.” In: *ACM SIGSOFT Software Engineering Notes*. Vol. 18. 3. ACM, pp. 139–148.

Weyuker, Elaine Jessica, Weiss, Stewart N., and Hamlet, Dick (1991). “Comparison of program testing strategies.” In: *Proceedings of the Symposium on Testing, Analysis, and Verification*, pp. 1–10.

Wohlin, Claes, Runeson, Per, Höst, Martin, Ohlsson, Magnus C, Regnell, Björn, and Wesslén, Anders (2012). *Experimentation in software engineering*. Springer Science & Business Media.

Wong, W Eric, Gao, Ruizhi, Li, Yihao, Abreu, Rui, and Wotawa, Franz (2016). “A survey on software fault localization.” In: *IEEE Transactions on Software Engineering* 42.8, pp. 707–740.

Wotawa, Franz, Nica, Mihai, and Aichernig, Bernhard K. (2010). “Generating Distinguishing Tests Using the Minion Constraint Solver.” In: *Proceering of the 3rd International Conference on Software Testing, Verification, and Validation Workshops (ICSTW)*, pp. 325–330. DOI: `10.1109/ICSTW.2010.11`.

Yao, Xiangjuan, Harman, Mark, and Jia, Yue (2014). “A study of equivalent and stubborn mutation operators using human analysis of equivalence.” In: *Proceedings of the 36th International Conference on Software Engineering*. ACM, pp. 919–930.

Zhang, Jie, Zhu, Muyao, Hao, Dan, and Zhang, Lu (2014). “An empirical study on the scalability of selective mutation testing.” In: *Proceedings of the 25th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, pp. 277–287.