Nguyen, Q. V., and Madeyski, L. Problems of mutation testing and higher order mutation testing. In Advanced Computational Methods for Knowledge Engineering, T. Do, H. A. L. Thi, and N. T. Nguyen, Eds., vol. 282 of Advances in Intelligent Systems and Computing. Springer International Publishing, 2014, pp. 157–172. DOI: 10.1007/978-3-319-06569-4_12 BibTeX: http://madeyski.e-informatyka.pl/download/MadeyskiRefs.bib

Problems of Mutation Testing and Higher Order Mutation Testing

Quang Vu Nguyen, Lech Madeyski

Institute of Informatics, Wroclaw University of Technology, WybrzezeWyspianskiego 27, 50370 Wroclaw, Poland
{quang.vu.nguyen, Lech.Madeyski}@pwr.wroc.pl

Abstract. Since Mutation Testing was proposed in the 1970s, it has been considered as an effective technique of software testing process for evaluating the quality of the test data. In other words, Mutation Testing is used to evaluate the fault detection capability of the test data by inserting errors into the original program to generate mutations, and after then check whether tests are good enough to detect them. However, the problems of mutation testing such as a large number of generated mutants or the existence of equivalent mutants, are really big barriers for applying mutation testing. A lot of solutions have been proposed to solve that problems. A new form of Mutation Testing, Higher Order Mutation Testing, was first proposed by Harman and Jia in 2009 and is one of the most promising solutions. In this paper, we consider the main limitations of Mutation Testing and previous proposed solutions to solve that problems. This paper also refers to the development of Higher Order Mutation Testing and reviews the methods for finding the good Higher Order Mutants.

Keywords: Mutation Testing, Higher Order Mutation, Higher Order Mutants.

1 Introduction

According to IEEE Std 829-1983 (IEEE Standard Glossary of Software Engineering Terminology), software testing is the process of analyzing a software item to detect the differences between existing and required conditions and to evaluate the features of the software items. In other words, software testing is execution a program using artificial data and evaluating software by observing its execution in order to find faults or failures. It is worth mentioning that "testing can only show the presence of errors, not their absence" which is often referred as Dijkstra's law. Where, according to IEEE Std 829-1983, error is a human action that produces an incorrect result, fault is an incorrect step, process, or data definition in a computer program and failure is the inability of a system or component to perform its required functions within specified performance requirements.

Software testing is always one of the important activities in order to evaluate the software quality. However, the quality of the set of testcases is a problem to be discussed. In addition, there are many cases that testers have not mentioned in the set of testcases. Mutation Testing has been introduced as a technique to assess the quality of the testcases.

Mutation Testing (MT), a technique that has been developed using two basic ideas: Competent Programmer Hypothesis ("programmers write programs that are reasonably close to the desired program") and Coupling Effect Hypothesis ("detecting simple faults will lead to the detection of more complex faults"), was originally proposed in 1970s by DeMillo et al.[1] and Hamlet[2]. While other software testing techniques focus on the correct functionality of the programs by finding error, MT focuses on test cases used to test the programs. In other words, the purpose of software testing is to find all the faults in a particular program whilst the purpose of MT is to create good sets of testcases. A good set of testcases is a set which is able to discover all the faults. With MT, mutants of a program are the different versions of the program. More specifically, each of which is generated by inserting only one semantic fault into original program (Table 1 gives an example to mutant). That generation is called mutation and that semantic fault is called mutation operator. It is a rule that is applied to a program to create mutants, for example modify expressions by replacing operators and inserting new operators. Mutation operators depend on programming languages, but there are traditional mutation operators: deletion of a statement; replacement of boolean expressions; replacement of arithmetic; replacement of a variable.

Table 1. An example of mutant (First Order Mutant)

Program P	Mutant P'

`...`
`while (hi<50) {`
`system.out.print(hi);`
`hi = lo +hi;`
`lo = hi -lo;`
`}`
`...`
	`...`
	`while (hi>50) {`
	`system.out.print(hi);`
	`hi = lo +hi;`
	`lo = hi -lo;`
	`}`
	`...`

The process of MT can be explained simply in following steps:

1. Suppose we have a program P and a set of testcases T

2. Produce mutant P1 from P by inserting only one semantic fault into P

3. Execute T on P and P1 and save results as R and R1

4. Compare R1 with R:

4.1 If R1 ≠ R: T can detect the fault inserted and has killed the mutant.

4.2 If R1=R: There could be 2 reasons:

+ T can't detect the fault, so have to improve T.

+ The mutant has the same semantic meaning as the original program. It's equivalent mutant (an example of equivalent mutant is showed in Table 2).

MT evaluates a set of testcases T by Mutation Score (MS), will be between 0 and 1, which is calculated by the following formula:

$$MS = \frac{\text{Number of killed mutants}}{\text{Total mutants} - \text{Equivalent mutants}}$$

A low score means that the majority of faults cannot be detected accurately by the test set. A higher score indicates that most of the faults have been identified with this particular test set. A good test set will have a mutation score close to 100%. When MS = 0, have no any testcase that can kill the mutants and when MS=1, we say that mutants are very easy to kill.

Table 2. An example of equivalent mutant

Program P	Mutant P'

…
int a =2;
if (b==2) {
System.out.print(b);
b = a + b;}
…
	…
	int a =2;
	if (b==2) {
	System.out.print(b);
	b = a * b;}
	…

In the next section, we summarize main limitations of mutation testing. Section 3 shows the previous proposed solutions for solving the limitations of mutation testing. Section 4 presents Higher Order Mutation Testing and its effectiveness. Section5 presents techniques to find good Higher Order Mutants. Section 6 presents conclusions and future work.

2 Main limitations of mutation testing

Although MT is a high automation and effective technique for evaluating the quality of the test data, Mutation Testing has three main problems in our view.

The first limitation of mutation testing is **a large number of mutants**, because program may have a fault in many possible places and with only one inserted semantic fault we will have one mutant. Thus, a large number of mutants will be generated in the mutant generation phase of mutation testing. Typically, this is a large number for even small program. For example, a simple program with just a sentence such as return a+b (where a, b are integers) may be mutated into many different ways: a−b, a*b, a/b, a+b++, −a+b, a+−b, 0+b, a+0, etc. This problem leads to a **very high execution cost** because the test cases are executed on not only original program but also each mutants. For example, assume that we have a program under test with 150 mutants and 200 testcases, it requires (1+150)*200 = 30200 executions with their corresponding results.

The second limitation of mutation testing is **realism**. Mutations are generated by single and simple syntactic changes, hence they do not denote realistic faults. While according to Langdon et al.[9], 90 percent of real faults are complex. In addition, it is not sure that we have found a large proportion of real faults present even if we have killed all the killable mutants[4].This is one of big limitations for applying mutation testing.

The third limitation of mutation testing is **equivalent mutant problem**[68]. In fact, many mutation operators can produce equivalent mutants which have the same behavior as original program. In this case, there is no testcase which could "kill" that mutants and the detection of equivalent mutants often involves additional human effort. Madeyski et al.[68] manually classified 1000 mutants as equivalent or non-equivalent and confirmed the finding by Schuller and Zeller[89] that it takes about12 minutes to assess one single mutation for equivalence. Therefore, there is often a need to ignore equivalent mutants, which would mean that we are ready to accept the lower bound on mutation score named mutation score indicator MSI [98][99][100][101].

3 Solutions proposed to solve problems of mutation testing

In order to collect a complete set of all the techniques to solve the problems of MT since the 1970s (after MT was proposed by DeMillo et al. [1] and Hamlet [2]) and sort them in chronologic order, with search terms which have either "mutation testing", "mutation analysis", "mutants + testing", "mutants + methods", "mutants + techniques", "mutants + problems", "mutation testing + improve", "equivalent mutants", "mutants*", "higher order mutants", we searched papers that were published in IEEE Explore, ACM Portal, Springer Online Library, Wiley Online Library, Citeseerx and journals or conference proceedings as IST (Information and Software Technology), JSS (Journal of Systems and Software), TSE (IEEE Transactions on Software Engineering), IET Software. After then, we continued to look at the references of each paper to find other articles related to our purpose. In addition, we also searched for Master and PhD theses that have contents related to "mutation testing". Figure 1 shows number of publications proposing techniques to solve the problems of Mutation Testing.

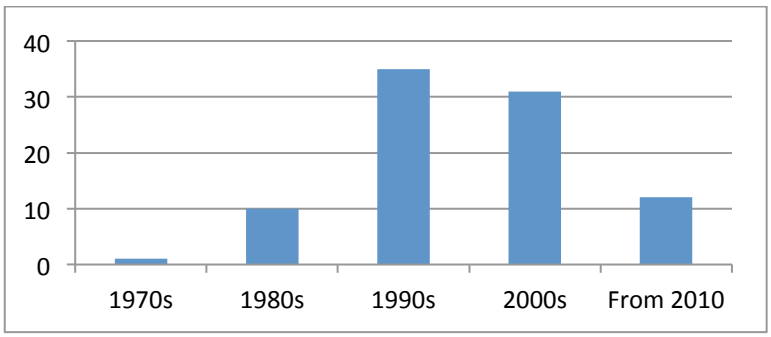

Fig.1. Number of publications proposing techniques to solve the problems of MT.

We searched and listed techniques to solve the problems of MT as reduce the number of mutants and execution cost, solve the realism problem and solve the equivalent mutant problem in chronologic order as follows:

3.1 Mutant and Execution cost Reduction Techniques

Mutant Sampling approach[14-20] was first proposed by Acree[14] and Budd[15] in 1980. In this approach, a small subset of mutants are randomly selected from the entire set. Some studies implied that Mutant Sampling is valid with a x% value higher than 10% (x% is % of mutants were selected randomly).

Reduction number of mutation operators will leads reduction number of mutants. This is idea of Agrawal et al.[23] and Mathur[24] and it was called Selective Mutation by Offutt et al.[18][25]. This approach suggested finding a small operators set that generate a subset of all possible mutants without losing test effectiveness. There are many authors have been applied this approach to effectively reduce generated mutants[20][26-32].

Instead of selecting mutants randomly, Husain[21] in 2008 used clustering algorithms to choose a subset of mutants. There is a empirical study of this approach in the work of Li et al.[22].

Second Order Mutation Testing[68][76][83][84[97] in particular and Higher Order Mutation Testing[3][4][9][10][35] in general are the most promising solutions to reduce number of mutants. Basic idea of this approach is improvement of MT by inserting two or more faults into original program to generate its mutants. For example, by combining two First Order Mutants to generate Second Order Mutant[68][83][84[97] or by using Subsuming Higher Order Mutants algorithms[3][4], the number of generated mutants can be reduced to about 50%.

Weak Mutation[36-42][93][94] was first proposed by Howden[36] in 1982 proposed. This is the first technique to optimize the execution of Traditional Mutation Testing (called Strong Mutation). Suppose that a program P is constructed from a set of components C= $\{c_1,..,c_n\}$. And mutant m is made by changing c_m,. So, we need only compare immediately the result of mutant m with the result of c_m execution to say m be killed or not (immediately check after mutated component be executed).

In order to improve quality of Strong Mutation and Weak Mutation, Woodward et al. [43][44] suggested an approach named Firm Mutation in 1988. This approach provides a continuum of intermediate possibilities to overcome the disadvantages of weak and strong mutation. In firm mutation, components may be groups of statements and partial program executions may be considered rather than each separate component execution as in weak mutation or the complete program execution as in strong mutation. It combines the reduced execution cost of weak mutation with the greater transparency of strong mutation.

In group of Run-time Optimization techniques, there are 6 techniques have been proposed. Interpreter-Based technique was first proposed in 1987 by Offutt and King[45][18] to reduce execution cost with idea: the original program is translated into an intermediate form, and then mutants are generated from this intermediate code. With Compiler-Based technique[46-48], each mutant is first compiled into an executable program, and then execute testcases on the compiled mutant. It is faster because execution of compiled binary code takes less time than code interpretation. Whilst main idea of Compiler-Intergrated[49][50] is designing an instrumented compiler to generate and compile mutants instead of compiling each mutant individually in the Compiler-Based technique. In order to improve the Interpreter-Based tech-

nique, Mutant Schema Generation[51-54][95] approach has been proposed. All mutants are encoded into a metaprogram, then the metaprogram is compiled and run with faster speed, basically, compared to compile and execute each individual mutation. Another technique is Bytecode Translation Technique[55-58] which was first proposed by Ma et al.[56] in 2005 to reduce execution cost of MT by using bytecode translation. Instead of from source code, mutants are generated from the compiled bytecode of original program and that generated mutants can be executed directly without compilation[55-58]. The last technique of this group is Aspect-Oriented Mutation which has been proposed by Bogacki and Walter[59][60]. In this approach, do not need to compile each mutant. An aspect patch is generated and each aspect will run programs twice: First for the original program and then for mutants.

There are also some techniques to improve computational cost base on computer architectures as SIMD (Single Instruction Multiple Data)[61-63], MIMD (Multiple Instruction Multiple Data)[64][65] and Parallel[66][67].

In 2013, Lisherness et al.[96] suggested a new approach by using coverage discounting for mutation analysis can reveal which functions are changed in the mutant, and in turn what is not being adequately tested if the mutation is undetected.

3.2 Techniques to solve the realism problem

There are many works have been demonstrated that the majority of real faults are complex faults. E.g. in [34][35], it is known that 90%. Base on that suggestion, Second Order Mutation Testing[68][77][84][85[97] in particular and Higher Order Mutation Testing[3][4][9][10] in general have been considered as the most promising solutions to solve the realism problem. That approaches have been suggested using more complex faults to generate mutants by inserting two or more faults into original program. In addition, Langdon et al.[9][35] introduced a new form of mutation testing: Multi Objective Higher Order Mutation Testing (with Genetic Programming)[35] in order to find higher order mutants that more realistic complex faults.

3.3 Techniques to solve the equivalent mutant problem

According to Madeyski et al.[68], the techniques to solve the equivalent mutant problem can be classified into 3 groups:

- Applying techniques to detect equivalent mutants such as Compiler Optimization[69][70], Mathematical Constraints[71][90],Program Slicing[72], Semantic differences in terms of running profile[73],Margrave's change-impact analysis[74] and Lesar model-checker[75].
- Applying techniques to avoid (or reduce) generating the equivalent mutants in the process of mutants generation such as Selective mutation[77], Program dependence analysis[78], Co-evolutionary[79], Equivalency conditions[80], Fault hierarchy[81], Semantic exception hierarchy[82], Higher order mutation testing[3], [83], [84], [68].
- Applying techniques in order to suggest the equivalent mutants such as Using Bayesian-learning based guidelines[85], Examining the impact of equivalent mutants

on coverage [86], Using the impact of dynamic invariants [87], Examining changes in coverage[88][89].

4 Higher order mutation testing and its effectiveness

Second Order Mutation Testing in particular and Higher Order Mutation Testing in general is an approach for generating mutants by applying mutation operators more than once. The idea about Second Order Mutation Testing was first mentioned by Offut[76] in 1992. After then, Polo et al. [100] in 2008, Kintis et al. [83] and M. Papadakis and N. Mlevris [84] in 2010, and L. Madeyski et al.[68] in 2013 further studied to suggest their algorithms to combine First Order Mutants to generate Second Order Mutants (SOMs). With this approach, not only at least 50% mutants were reduced without loss of effectiveness of testing, but also the number of equivalent mutants can be reduced (the reduction in the mean percentage of equivalent mutants passes from about 18.66% to about 5%) and generated second order mutants can be harder to kill [97][84][85][68].

In 2009, Higher Order Mutation Testing (HOM Testing) was first proposed by Jia, Harman and Langdon[3][4]. According to them, mutants can be classified into two types: First Oder Mutants (FOMs – used in Traditional Mutation Testing) - are generated by applying mutation operators only once - and Higher Oder Mutants (HOMs-used in Higher Order Mutation Testing) – are constructed by inserting two or more faults. A simple example is presented as Table 3:

Table 3. An example of higher order mutant

Program P	**FOM1**

…
while ((hi < 50) && (hi>lo)){
System.out.print(hi);
hi = lo + hi;
lo = hi – lo; }
…
	…
	while ((**hi > 50**) && (hi>lo)){
	System.out.print(hi);
	hi = lo + hi;
	lo = hi – lo; }
	…

FOM2	**HOM (Createdfrom FOM1 and FOM2)**

…
while ((hi < 50) && (**hi<lo**)){
System.out.print(hi);
hi = lo + hi;
lo = hi – lo; }
…
	…
	while ((**hi > 50**) && (**hi<lo**)){
	System.out.print(hi);
	hi = lo + hi;
	lo = hi – lo; }
	…

The same as Second Order Mutants, the combination of faults of higher order mutation testing can reduces the number of generated mutants and limits the number of mutants and those generated mutants are harder to kill than any of the individual constituent faults. E.g. in case of subsuming Higher Order Mutants [3][4].A subsuming HOM was constructed from constituent FOMs and so that HOM is only killed by

subset of the intersection of testcases which kill each constituent FOMs. Therefore, the quality of testcases will be better but the number of testcases will be reduced. Fewer number mutants and fewer number of test cases lead to a fewer execution cost. In addition, with the combination of faults, we also limit unrealistic and avoid generating equivalent mutants [3][68][76][83-84].

To be more clearly, let's consider the example given in Table 3. In this case, we have only one HOM created from FOM1 and FOM2 instead of FOMs. And there are TWO test cases which kill FOM1: {hi<50 && hi>lo} and {hi>50 && hi>lo}, TWO test cases which kill FOM2: {hi<50 && hi>lo} and {hi<50 && hi<lo}. But we need ONLY ONE test case which kills HOM: {hi<50 && hi>lo} and this test case also kills both FOM1 and FOM2. But the reverse is not true, the test case {hi>50 && hi>lo} does not kill HOM and the test case {hi<50 && hi<lo} also does not kill HOM. And so, with HOM, we can reduce number mutants generated and number of test cases without leading to loss of effectiveness during testing. Similarly, the combination of two or more errors to generate mutants will cause limited number of generated equivalent mutants and number of "easy to kill" mutants.

Unlike some other techniques presented in part 3 that only solve individual problems of traditional mutation testing, higher order mutation testing could help us to deal with three main problems of the traditional mutation testing at the same time. That is why we choose higher order mutation testing to study. In the next part, we will present more detail on this issue.

5 Finding the good HOMs

5.1 Second Order Mutants (SOMs)

In 2008, with the results of their study, Polo et al.[97] believed that "mutant combination does not decrease the quality of the test suite". They suggested[97] 3 algorithms to generate second order mutants (mutants containing two simple faults are called second order mutants). With the LastToFirst algorithm, assume that we have the list of n mutants of First Order Mutants (FOMs), SOMs were generated by combining the first FOM with the FOM number n, the second-FOM with the FOM number (n-1), and so on. In the second one, DifferentOperators algorithm, SOMs were generated by combining two FOMs generated by different mutation operators. And the last algorithm, RandomMix, combines any two FOMs to generate SOMs. In their study, after FOMs were generated from 6 programs under test (Bisect with 31 LOC-Line of Code, Bub-54, Find-79, Fourballs-47, Mid-59 and TriTyp-61), the three combination algorithms were used to generate SOMs. Number of test cases corresponding to each program are 25, 256, 135, 96, 125, 216 were passed to the FOMs and the SOMs. Results showed that the number of SOMs decreased about 50% compared with FOMs, and the mean percentage of equivalent SOMs is only about 5% compared with 18,66% of equivalent FOMs [97].

Based on the algorithms of the Polo et al.[97], M. Papadakis and N. Malevris studied and proposed five new strategies to combine FOMs: First2Last , SameNode, SameUnit , SU_F2Last and SU_DiffOp[84]. They executed both of First Order Mu-

tation Testing strategies (Strong Mutation, Rand 10%, Rand 20%, Rand 30%, Rand 40%, Rand 50%, Rand 60%) and Second Order Mutation Testing strategies (RandomMix, DifferentOperators, First2Last, SameNode, SameUnit, SU_F2Last and-SU_DiffOp) with the same test programs[84]. With their empirical study[84], the mean number of equivalent mutants with the First Order Mutation Testing strategies is about 6237, and with the Second Order Mutation Testing strategies is about 2727. Meanwhile, Second Order Mutation Testing strategies reduced the number of generated SOMs by about 50%, because of SOMs also were generated combining two FOMs, and of course will lead to reduced execution cost.

In 2010, Kintis et al.[83] suggested two categories of Second Order Mutation Testing strategies: The Second Order Strategies category includes RDomF and SDomF strategy; The Hybrid Strategies category includes HDom(20%) and HDom(50%) strategy. The results of their study indicates that the number of generated equivalent mutants of Weak Mutation, RDomF, SDomF, HDom(20%) and HDom(50%) strategies reduced about 73%, 85.4%, 86.8%,81.4% and 65.5% in turn compared with the number of generated equivalent mutants of Strong Mutation. Meanwhile the Mutation Scores are 99.94%, 99.99%, 99.91%, 99.91% for the RDomF, SDomF, HDom(20%) and HDom(50%) strategies respectively. That mutation socres are higher than Mutation Score of Weak Mutation strategy (96.90%).

Most recently, in 2013, Madeyski et al.[68] introduced the JudyDiffOp algorithm and NeighPair algorithm. JudyDiffOp algorithm is a modification of the DifferentOperatorsalgorithm[100] with the idea that "both constituent FOMs were used only once for producing a SOM". And NeighPair algorithm was introduced with idea "generate SOM by combining FOMs which are as close to each other as possible". They experimented[68] 4 algorithms RandomMix, Last2First[97] and JudyDiffOp, NeighPair[68] on programs under test Barbecue (7.413 LOC – Line of Code), Commons IO (16.283 LOC), Commons Lang (48.507 LOC) and Commons Math (80.023 LOC) with the number of test cases are 21,43,88,221 respectively. That algorithms reduce the number of generated SOMs at least by half compared with generated FOMs, especially JudyDiffOp algorithms over 60%. The mean reductions of equivalent mutants number are about 47%, 58.5%, 66% for the RandomMix, Last2First and JudyDiffOp in turn. And the NeighPair algorithm is not good in terms of equivalent mutant reduction. However, "In most of the cases mutation score estimations were higher for the SOM strategies than for FOM"[68].

5.2 Subsuming HOMs

In 2009, Jia and Harman[3] introduced the term "Subsuming HOMs" as follows: The HOM is named "Subsuming HOM" if it is harder to kill than their constituent FOMs. And then they suggested some approaches to find the Subsuming HOMs by using some meta-heuristic algorithms[3]: Greedy Algorithm, Genetic Algorithm and Hill-Climbing Algorithm. In order to find the Subsuming HOMs more effectively, they introduced two definitions[3]: Fragility value, a value for measuring the ability of a FOM or HOM can be killed and Fitness Function is the ratio of the fragility of a HOM to the fragility of the constituent FOMs. In their approach, they used this Fit-

ness Function to evaluate the fitness of HOMs. They experimented with 10 benchmark C programs under test (14850 LoC and 35473 test cases in total) and the results indicate that genetic algorithm is the most efficient algorithm for finding those subsuming HOMs, while the greedy algorithm and hill climbing algorithm can also be used to improve the quality of the results[3].

5.3 Multi-Object Genetic Programming

In order to find higher order mutants that are hard to kill and more realistic complex faults, Langdon et al.[9][35] introduced a new form of mutation testing: Multi Objective Higher Order Mutation Testing (with Genetic Programming)[35]. They believed that although there are a lot of number of FOMs was generated but most are simply and do not denote realistic faults. So they suggested inserting faults that are semantically close to the original program instead of inserting faults that are syntactically close to the original program in order to find higher order mutants that are syntactically similar to the original program under test. With this, multi objective Pareto optimal genetic programming approach, the number of mutants grows exponentially with order but the number of equivalent mutants fall rapidly with number of changes made [35]. For example, with the chosen set of the C comparison operations (<,<=, ==, !=, >=, >), and the program under test that contains 17 comparison operators (i.e. the Triangle benchmark Triangle.c), there are 85 1st order mutants (17×5 programs with one change), 3400 2nd order mutants (17×5×16×5/2 with two changes), 85000 3rd order mutants(17×5×16×5×15×5/6 with three changes), and 1487500 4th order mutants (17×5×16×5×15×5×14×5/24 with four changes),whilst respectively only 8, 28, 56, 70 equivalent mutants that pass all test cases and 7, 55, 189, 371 mutants that fail just one test[35].

6 Conclusion and future work

Since was proposed in 1970s by DeMillo et al.[1] and Hamlet[2], Mutation Testing has been considered as a powerful technique for evaluating the quality of the test cases. Basically, there is still work to be done to improve the quality of mutation testing. This paper reviewed a range of strategies that were proposed to solve three main problems of mutation testing: a vast number of mutants (and also high execution cost), realism of faults and equivalent mutant problem. Each technique has its own advantages and disadvantages and we focus on Higher Order Mutation testing because this is not only a newest method but also a promising solution of three main problems of the traditional mutation testing at the same time. However, the number of mutants grows exponentially with order. So, in the future, we will research to improve and solve that problem for finding good HOMs by applying Multi-Object optimization algorithm. Specifically we are going to:

- Use Java language programming and Judy mutation testing tool for Java (http://madeyski.e-informatyka.pl/tools/judy/)[68][98].
- Search for strongly subsuming HOMs applying multi-objective optimization.

- Assess results, according to the criteria of solving the problems of traditional mutation testing (reduce the number of mutants and execution cost, realism and the equivalent mutant problem), and compare that results with the results of algorithms that have been proposed previously.

References

1. DeMillo, R.A., Lipton R.J., Sayward, F.G.: Hints on test data selection: help for the practicing programmer. IEEE Computer 11 (4), 34–41 (1978)
2. Hamlet, R.G.: Testing programs with the aid of a compiler. IEEE Transactions on Software Engineering SE-3 (4), 279–290 (1977)
3. Jia, Y., Harman, M.: Higher order mutation testing. Information and Software Technology 51, 1379–1393 (2009)
4. Harman, M., Jia, Y., Langdon, W. B.: A Manifesto for Higher Order Mutation Testing. In: Third International Conf. on Software Testing, Verification, and Validation Workshops, (2010)
5. Afzal, W., Torkar, R., Feldt, R.: A systematic review of search-based testing for non-functional system properties. Information and Software Technology, 51(6), 957–976, (2009)
6. Harman, M.: The current state and future of search based software engineering. In: L. Briand and A. Wolf, editors, Future of Software Engineering 2007, pages 342–357.. IEEE Computer Society Press. Los Alamitos, California, USA (2007)
7. Harman, M., Mansouri, A., Zhang, Y.: Search based software engineering: A comprehensive analysis and review of trends techniques and applications. Technical Report TR-09-03, Department of Computer Science, King's College London (2009)
8. Raiha, O.: A survey on search based software design. Technical Report Technical Report D-2009-1, Department of Computer Sciences, University of Tamper (2009)
9. Langdon, W.B., Harman, M., Jia, Y.: Efficient multi-objective higher order mutation testing with genetic programming. The Journal of Systems and Software 83 (2010)
10. Jia, Y., and Harman, M.: Constructing Subtle Faults Using Higher Order Mutation Testing. In: Proc. Eighth Int'l Working Conf. Source Code Analysis and Manipulation (2008)
11. Jia, Y., Harman, M.: MILU: A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language. In: Proceedings of the 3rd Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART'08), pages 94–98. IEEE Computer Society. Windsor, UK (2008)
12. Offutt, A.J.: Investigations of the software testing coupling effect. ACM Transactions on Software Engineering and Methodology 1 (1) , 5–20 (1992)
13. Adamopoulos, K., Harman, M., Hierons, R. M.: How to Overcome the Equivalent Mutant Problem and Achieve Tailored Selective Mutation Using Co-evolution. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'04), volume 3103 of LNCS, pages 1338–1349. Seattle, Washington, USA (2004)
14. Acree, A.T: On Mutation. PhD thesis, Georgia Inst. Of Technology (1980)
15. Budd, T.A.: Mutation Analysis of Program Test Data. PhD thesis, Yale Univ. (1980)
16. DeMillo, R.A, Guindi, D.S., King, K.N., McCracken, W.M., Offutt, A.J.: An Extended Overview of the Mothra Software Testing Environment. In: Proceedings Second Workshop Software Testing, Verification, and Analysis, pp. 142-151 (1988)
17. Sahinoglu, M., Spafford, E.H.: A Bayes Sequential Statistical Procedure for Approving Software Products. In: Proc. IFIP Conf. Approving Software Products, pp. 43-56 (1990)

18. King, K.N., Offutt, A.J.: A Fortran Language System for Mutation-Based Software Testing. Software: Practice and Experience, vol. 21, no. 7, pp. 685-718 (1991)
19. Mathur A.P., Wong, W.E.: An Empirical Comparison of Mutation and Data Flow Based Test Adequacy Criteria. Technical Report, Purdue Univ. (1993)
20. Wong, W.E.: On Mutation and Data Flow. PhD thesis, Purdue Univ. (1993)
21. Hussain, S.: Mutation Clustering. Master's thesis, King's College London (2008)
22. Ji, C., Chen, Z., Xu, B., Zhao, Z.: A Novel Method of Mutation Clustering Based on Domain Analysis. In: Proc. 21st Int'l Conf. Software Eng. and Knowledge Eng. (2009)
23. Agrawal, H., DeMillo, R.A., Hathaway, B., Hsu, W., Krauser, E.W., Martin, R.J., Mathur, A.P., Spafford, E.: Design of Mutant Operators for the C Programming Language. Technical Report SERC-TR-41-P, Purdue Univ. (1989)
24. Mathur, A.P.: Performance, Effectiveness, and Reliability Issues in Software Testing. In: Proc. Fifth Int'l Computer Software and Applications Conf.,pp. 604-605 (1991)
25. Offutt, A.J., Rothermel, G., Zapf, C.: An Experimental Evaluation of Selective Mutation. In: Proc. 15th Int'l Conf. Software Eng.,pp. 100-107 (1993)
26. Wong, W.E., Mathur, A.P.: Reducing the Cost of Mutation Testing: An Empirical Study. J. Systems and Software, vol. 31, no. 3, pp. 185-196 (1995)
27. Offutt, A.J., Lee, A., Rothermel, G., Untch, R.H., Zapf, C.: An Experimental Determination of Sufficient Mutant Operators. ACM Trans. Soft. Eng. and Methodology (1996)
28. Mresa, E.S., Bottaci, L.: Efficiency of Mutation Operators and Selective Mutation Strategies: An Empirical Study. Software Testing, Verification, and Reliability,vol. 9, no. 4, pp. 205-232 (1999)
29. Barbosa, E.F., Maldonado, J.C., Vincenzi, A.M.R: Toward the Determination of Sufficient Mutant Operators for C. Software Testing, Verification, and Reliability,vol. 11, no. 2, pp. 113-136 (2001)
30. Namin, A.S., Andrews, J.H.: Finding Sufficient Mutation Operators via Variable Reduction. In: Proc. Second Workshop Mutation Analysis, p. 5 (2006)
31. Namin, A.S., Andrews, J.H.: On Sufficiency of Mutants. In: Proc. 29th Int'l Conf. Software Eng., pp. 73-74 (2007)
32. Namin, A.S., Andrews, J.H., Murdoch, D.J.: Sufficient Mutation Operators for Measuring Test Effectiveness. In: Proc. 30th Int'l Conf. Software Eng., pp. 351-360 (2008)
33. Polo, M., Piattini, M., Garcia-Rodriguez, I.: Decreasing the Cost of Mutation Testing with Second-Order Mutants. Software Testing, Verification, and Reliability,vol. 19, no. 2, pp. 111-131 (2008)
34. Purushothaman, R., Perry, D.E.: Toward Understanding the Rhetoric of small source code changes. IEEE transactions on Software Engineering, vol. 31, No. 6 (2005)
35. Langdon, W.B., Harman, M., Jia, Y.: Multi Objective Higher Order Mutation Testing with Genetic Programming. In: Proc. Fourth Testing: Academic and Industrial Conf. Practice and Research (2009)
36. Howden, W.E.: Weak Mutation Testing and Completeness of Test Sets. IEEE Trans. Soft. Eng., vol. 8, no. 4, pp. 371-379 (1982)
37. Girgis, M.R., Woodward, M.R.: An Integrated System for Program Testing Using WeakMutation and Data Flow Analysis. In: Proc. Eighth Int'l Conf. Software Eng. (1985)
38. Horgan, J.R., Mathur, A.P.: Weak Mutation is Probably Strong Mutation. Technical Report SERC-TR-83-P, Purdue Univ. (1990.)
39. Woodward, M.R.: Mutation Testing-An Evolving Technique. In: Proc. IEE Colloquium on Software Testing for Critical Systems, pp. 3/1-3/6 (1990)
40. Marick, B.: The Weak Mutation Hypothesis. In: Proc. Fourth Symp. Software Testing, Analysis and Verification, pp. 190-199 (1991)

41. Offutt, A.J., Lee, S.D.: How Strong is Weak Mutation. In: Proc. Fourth Symp. Software Testing, Analysis and Verification, pp. 200-213 (1991)
42. Offutt, A.J, Lee, S.D.: An Empirical Evaluation of Weak Mutation. IEEE Trans. Software Eng., vol. 20, no. 5, pp. 337-344 (1994)
43. Woodward, M.R., Halewood, K.: From Weak to Strong Dead or Alive? An Analysis of Some Mutationtesting Issues. In: Proc. Second Workshop Software Testing, Verification, and Analysis, pp. 152-158 (1988)
44. Jackson, D., Woodward, M.R.: Parallel Firm Mutation of Java Programs. In: Proc. First Workshop Mutation Analysis, pp. 55-61 (2000)
45. Offutt, A.J., King, K.N.: A Fortran 77 Interpreter for Mutation Analysis. ACM SIGPLAN Notices, vol. 22, no. 7, pp. 177-188 (1987)
46. Choi, B., Mathur, A.P.: High-Performance Mutation Testing. J. Systems and Software, vol. 20, no. 2, pp. 135-152 (1993)
47. Delamaro, M.E.: Proteum-A Mutation Analysis Based Testing Environment. Master's thesis, Univ. of Sao Paulo (1993)
48. Delamaro, M.E., Maldonado, J.C.: Proteum: A Tool for the Assessment of Test Adequacy for C Programs. In: Proc. Conf. Performability in Computing Systems (1996)
49. DeMillo, R.A., Krauser, E.W., Mathur, A.P.: Compiler-Integrated Program Mutation. In: Proc. Fifth Ann. Computer Software and Applications Conf., pp. 351-356 (1991)
50. Krauser, E.W.: Compiler-Integrated Software Testing. PhD thesis, Purdue Univ. (1991)
51. Untch, R.H.: Mutation-Based Software Testing Using Program Schemata. In: Proc. 30th Ann. Southeast Regional Conf., pp. 285-291 (1992)
52. Mathur, A.P.: CS 406 Software Engineering I. Course Project Handout (1992)
53. Untch, R.H., Offutt, A.J., Harrold, M.J.: Mutation Analysis Using Mutant Schemata. In: Proc. Int'l Symp. Software Testing and Analysis, pp. 139-148 (1993)
54. Untch, R.H.: Schema-Based Mutation Analysis: A New Test Data Adequacy Assessment Method. PhD thesis, Clemson Univ. (1995)
55. Offutt, A.J., Ma, Y.S., Kwon, Y.R.: An Experimental Mutation System for Java. ACM SIGSOFT Software Eng. Notes, vol. 29, no. 5, pp. 1-4, Sept (2004)
56. Ma, Y.S., Offutt, A.J., Kwon, Y.R.: MuJava: An Automated Class Mutation System. Software Testing, Verification, and Reliability, vol. 15, no. 2, pp. 97-133 (2005)
57. Ma, Y.S., Offutt, A.J., Kwon, Y.R: MuJava: A Mutation System for Java. In: Proc. 28th Int'l Conf. Software Eng., pp. 827-830 (2006)
58. Schuler, D., Dallmeier, V., Zeller, A.: Efficient Mutation Testing by Checking Invariant Violations. In: Proc. Int'l Symp. Software Testing and Analysis (2009)
59. Bogacki, B., Walter, B.: Evaluation of Test Code Quality with Aspect-Oriented Mutations. In: Proc. Seventh Int'l Conf. eXtreme Programming and Agile Processes in Software Eng.,pp. 202-204 (2006)
60. Bogacki, B., Walter, B.: Aspect-Oriented Response Injection: An Alternative to Classical Mutation Testing. Soft. Eng. Techniques: Design for Quality, Springer (2007)
61. Mathur, A.P., Krauser, E.W.: Mutant Unification for Improved Vectorization. Technical Report SERC-TR-14-P, Purdue Univ. (1988)
62. Krauser, E.W., Mathur, A.P., Rego, V.J.: High Performance Software Testing on SIMD Machines. In: Proc. Second Workshop Software Testing, Verification and Analysis (1988)
63. Krauser, E.W., Mathur, A.P., Rego, V.J.: High Performance Software Testing on SIMD Machines. IEEE Trans. Software Eng., vol. 17, no. 5, pp. 403-423 (1991)
64. Offutt, A.J., Pargas, R.P., Fichter, S.V., Khambekar, P.K.: Mutation Testing of Software Using a MIMD Computer. In: Proc. Int'l Conf. Parallel Processing, pp. 255-266 (1992)

65. Zapf, C.N.: A Distributed Interpreter for the Mothra Mutation Testing System. Master's thesis, Clemson Univ. (1993)
66. Weiss, S.N., Fleyshgakker, V.N.: Improved Serial Algorithms for Mutation Analysis. ACM SIGSOFT Software Eng. Notes, vol. 18, no. 3, pp. 149-158 (1993)
67. Fleyshgakker, V.N., Weiss, S.N.: Efficient Mutation Analysis: A New Approach. In: Proc. Int'l Symp. Software Testing and Analysis, pp. 185-195 (1994)
68. Madeyski, L., Orzeszyna, W., Torkar, R., Józala, M.: Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation. IEEE Transactions on Software Engineering, accepted in 2013.http://dx.doi.org/10.1109/TSE.2013.44
69. Baldwin, D., Sayward, F.G.: Heuristics for determin-ing equivalence of program mutations. Yale University, New Haven, Connecticut, Tech Report 276 (1979)
70. Offutt, A.J., Craft, W.M.: Using compiler optimization techniques to detect equivalent mutants. Software Testing, Verification and Reliability, vol. 4, no. 3, pp. 131–154 (1994)
71. Offutt, A.J., Pan, J.: Detecting equivalent mutants and the feasible path problem. In: Proc. Eleventh Annual Conf. 'Systems Integrity Computer Assurance COMPASS '96 Software Safety. Process Security', pp. 224–236 (1996)
72. Hierons, R., Harman, M., Danicic, S.: Using program slicing to assist in the detection of equivalent mutants. Software Testing, Verification and Reliability (1999)
73. Ellims, M., Ince, D., Petre, M.: The Csaw C mutation tool: Initial results. In: Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, pp. 185–192. IEEE Computer Society, Washington, DC, USA (2007)
74. Martin, E., Xie, T.: A fault model and mutation testing of access control policies. In: Proceedings of the 16th international conference on World Wide Web, ser. WWW '07, New York, USA: ACM Press, pp. 667–676 (2007)
75. DuBousquet, L., Delaunay, M.: Towards mutation analysis for Lustre programs. Electronic Notes in Theoretical Computer Science, vol. 203, no. 4, pp. 35–48 (2008)
76. Offutt, A.J.: Investigations of the software testing coupling effect. ACM Transactions on Software Engineering Methodology, vol. 1, pp. 5–20 (1992)
77. Mresa, E.S., Bottaci, L.: Efficiency of mutation operators and selective mutation strategies: An empirical study. Software Testing, Verification and Reliability (1999)
78. Harman, M., Hierons, R., Danicic, S.: The relationship between program dependence and mutation analysis. Mutation testing for the new century, W. E. Wong, Ed. Norwell, MA, USA: Kluwer Academic Publishers, pp.5–13 (2001)
79. Adamopoulos, K., Harman, M., Hierons, R.M.: How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. Lecture Notes in Computer Science, vol. 3103. Springer Berlin /Heidelberg, pp. 1338–1349 (2004)
80. Offutt, A.J., Ma, Y.S., Kwon, Y.R.: The class-level mutants of MuJava. In: Proceedings of the 2006 international workshop on Automation of software test - AST '06, ser. AST '06, New York, USA: ACM Press, pp. 78–84 (2006)
81. Kaminski, G., Ammann, P.: Using a fault hierarchy to improve the efficiency of DNF logic mutation testing. In: Proc. Int. Conf. Software Testing Verification and Validation ICST'09, pp. 386–395 (2009)
82. Ji, C., Chen, Z., Xu, B., Wang, Z.: A new mutation analysis method for testing Java exception handling. In: Proc. 33rd Annual IEEE Int. Computer Software and Applications Conf. COMPSAC '09, vol. 2, pp. 556–561 (2009)
83. Kintis, M., Papadakis, M., Malevris, N.: Evaluating mutation testing alternatives: A collateral experiment. In: Proc. 17th Asia Pacific Soft. Eng. Conf. (APSEC) (2010)

84. Papadakis, M., Malevris, N.: An empirical evaluation of the first and second order mutation testing strategies. In: Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops, ser. ICSTW'10, IEEE Computer Society, pp. 90–99 (2010)
85. Vincenzi, A.M.R., Nakagawa, E.Y., Maldonado, J.C., Delamaro, M.E., Romero, R.A.F: Bayesian-learning based guide-lines to determine equivalent mutants. International Journal of Soft. Eng. and Knowledge Engineering, vol. 12, no. 6, pp. 675–690 (2002)
86. Grün, B.J.M., Schuler, D., Zeller, A.: The impact of equivalent mutants. In: Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops, Denver, Colorado, USA: IEEE Computer Society, pp. 192–199 (2009)
87. Schuler, D., Dallmeier, V., Zeller, A.: Efficient mutation testing by checking invariant violations. In: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ser. ISSTA '09, New York, USA: ACM Press (2009)
88. Schuler, D., Zeller, A.: (Un-)covering equivalent mutants. In: Proceedings of the 3rd International Conference on Software Testing Verification and Validation (ICST'10), Paris, France, pp. 45–54 (2010)
89. Schuler, D., Zeller, A.: Covering and uncovering equivalent mutants. Software Testing, Verification and Reliability, 23(5), pp.353-374 (2012)
90. Offutt, A.J., Pan, J.: Automatically Detecting Equivalent Mutants and Infeasible Paths. Software Testing, Verification and Reliability,vol. 7, no. 3, pp. 165-192 (1997)
91. Clark, J.A., Dan, H., Hierons, R.M.: Semantic Mutation Testing. Third International Conf. on Software Testing, Verification and Validation Workshops (2010)
92. Dan, H., Hierons, R.M.: Semantic Mutation Analysis of Floating-point Comparison. IEEE Fifth International Conference on Software Testing, Verification and Validation (2012)
93. Boonyakulsrirung, P., Suwannasare, T.: A Weak Mutation Testing Framework for WS-BPEL. Eighth International Joint Conf.on Computer Science and Soft.Engineering (2011)
94. Durelli, V.H.S., Offutt, A.J., Delamaro, M.E.: Toward Harnessing High-level Language Virtual Machines for Further Speeding up Weak Mutation Testing. IEEE Fifth International Conference on Software Testing, Verification and Validation (2012)
95. Mateo, P.R., Usaola, M.P.: Mutant Execution Cost Reduction Through MUSIC (MUtant Schema Improved with extra Code). IEEE Fifth International Conference on Software Testing, Verification and Validation (2012)
96. Lisherness, P., Lesperance, N., Cheng, K.T.: Mutation Analysis with Coverage Discounting. Design, Automation and Test in Europe Conference and Exhibition (2013)
97. Polo, M., Piattini, M., Garcia-Rodriguez, I.: Decreasing the Cost of Mutation Testing with Second-Order Mutants. Software Testing, Verification, and Reliability,vol. 19, no. 2, pp. 111-131 (2008)
98. Madeyski, L., Radyk, N.: Judy - a mutation testing tool for Java.IET Software 4(1): 32-42 (2010).http://madeyski.e-informatyka.pl/download/Madeyski10b.pdf
99. Madeyski, L.: On the effects of pair programming on thoroughness and fault-finding effectiveness of unit tests. Product-Focused Software Process Improvement, ser. Lecture Notes in Computer Science, J. Münch and P. Abrahamsson, Eds. Springer, vol. 4589, pp. 207–221 (2007). http://madeyski.e-informatyka.pl/download/Madeyski07.pdf
100. Madeyski, L.: Impact of pair programming on thoroughness and fault detection effectiveness of unit test suites. Software Process: Improvement and Practice, vol. 13, no.3, pp. 281–295 (2008). http://madeyski.e informatyka.pl/download/Madeyski08.pdf
101. Madeyski, L.: The impact of test-first programming on branch coverage and mutation score indicator of unit tests: An experiment. Information and Software Technology, vol. 52, pp. 169–184 (2010). http://madeyski.e-informatyka.pl/download/Madeyski10c.pdf