Nguyen, Q. V., and Madeyski, L. Searching for Strongly Subsuming Higher Order Mutants by Applying Multi-objective Optimization Algorithm. In Advanced Computational Methods for Knowledge Engineering, H. A. Le Thi, N. T. Nguyen, and T. V. Do, Eds., vol. 358 of Advances in Intelligent Systems and Computing. Springer, 2015, pp. 391–402. DOI: 10.1007/978-3-319-17996-4_35 BibTeX: http://madeyski.e-informatyka.pl/download/MadeyskiRefs.bib

Searching for Strongly Subsuming Higher Order Mutants by applying multi-objective optimization algorithm

Quang Vu Nguyen, Lech Madeyski

Faculty of Computer Science and Management, Wroclaw University of Technology,
Wybrzeze Wyspianskiego 27, 50-370 Wroclaw, Poland
{quang.vu.nguyen, Lech.Madeyski}@pwr.edu.pl

Abstract. Higher order mutation testing is considered a promising solution for overcoming the main limitations of first order mutation testing. Strongly subsuming higher order mutants (SSHOMs) are the most valuable among all kinds of higher order mutants (HOMs) generated by combining first order mutants (FOMs). They can be used to replace all of its constituent FOMs without scarifying test effectiveness. Some researchers indicated that searching for SSHOMs is a promising approach. In this paper, we not only introduce a new classification of HOMs but also new objectives and fitness function which we apply in multi-objective optimization algorithm for finding valuable SSHOMs.

Keywords: Mutation Testing, Higher Order Mutation, Higher Order Mutants, Strongly Subsuming, Multi-objective optimization algorithm.

1 Introduction

First order mutation testing (traditional mutation testing (MT)) is 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"). It was originally proposed in 1970s by DeMillo et al.[1] and Hamlet[2]. MT is based on generating different versions of the program (called mutants) by insertion (via a mutation operator) a semantic fault or change into each mutant. The process of MT can be explained by the following steps:

1. Suppose we have a program P and a set of test cases 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 we have to improve T.
+ The mutant has the same semantic meaning as the original program. It's an equivalent mutant.

The ratio of killed mutants to all generated mutants is called mutation score indicator (MSI) [8][9][10][11]. MSI is a quantitative measure of the quality of test cases. It is different from mutation score (MS)that was defined [1][2] as the ratio of killed mutants to difference of all generated mutants and equivalent mutants.

Although MT is a high automation and effective technique for evaluating the quality of the test data, there are three main problems[4][11][18]: a large number of mutants (this also leads to a very high execution cost); realism and equivalent mutant problem[11][19]. A number of approaches were proposed for overcoming that problems[18] including second order mutation testing [11][12][13][14][15] in particular and higher order mutation testing[3][4][5][6] in general. These approaches use more complex changes to generate mutants by inserting two or more faults into original program.

In 2009, Jia and Harman[3] introduced six types of HOMs on a basis of two definitions:

Definition 1: The higher order mutant (HOM) is called "Subsuming HOM" if it is harder to kill than their constituent first order mutants (FOMs). If set of test cases (TCs) which kill HOM are only inside the intersection of the sets of TCs which kill FOMs, it is a "Strongly Subsuming HOM", otherwise it is a "Weakly Subsuming HOM".

Definition 2: The HOM is called "Coupled HOM" if set of TCs that kill FOMs also contains cases that kill HOM. Otherwise, it is called "De-coupled HOM".

Six types of HOMs are (1)Strongly Subsuming and Coupled; (2)Weakly Subsuming and Coupled; (3)Weakly Subsuming and Decoupled; (4)Non-Subsuming and Decoupled; (5)Non-Subsuming and Coupled; (6)Equivalent. **Strongly Subsuming and Coupled HOM** is harder to kill than any constituent FOM and it is only killed by subset of the intersection of set of test cases that kill each constituent FOM. As a result, **Strongly Subsuming and Coupled HOM** can be used to replace all of its constituent FOMs without loss of test effectiveness. Finding **Strongly Subsuming and Coupled HOMs** can help us to overcome the above-mentioned three limitations of first order mutation testing.

Jia and Harman[3] introduced also some approaches to find the Subsuming HOMs by using the following algorithms: Greedy, Genetic and Hill-Climbing. In their experiments, approximately 15% of all found Subsuming HOMs were Strongly Subsuming HOMs. In addition, Langdon et al.[7] suggested applying multi-objective optimization algorithm to find higher order mutants that represent more realistic complex faults. As a result, our research focuses on searching valuable SSHOMs by applying multi-objective optimization algorithm.

In the next section, we present our approach to classify HOMs. Section 3 describes the proposed objectives and fitness function used for searching and identifying HOMs. Section 4 presents our experiment on searching for SSHOMs by applying

multi-objective optimization algorithm. And the last section is conclusions and future work.

2 HOMs classification

Our HOMs classification approach based on the combination of a set of test cases (TCs) which kill HOM and sets of TCs which kill FOMs. Let H be a HOM, constructed from FOMs F1 and F2. We also explain some notation below (See Fig. 1).

T: The given set of test cases

$T_{F1} \subset T$: Set of test cases that kill FOM1

$T_{F2} \subset T$: Set of test cases that kill FOM2

$T_H \subset T$: Set of test cases that kill HOM generated from FOM1 and FOM2

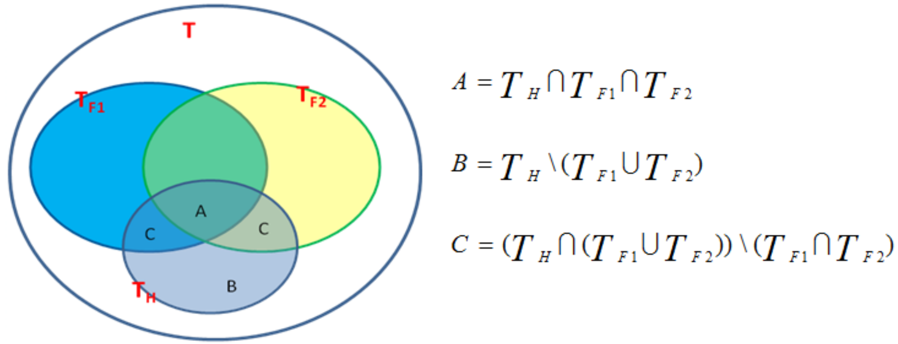

Fig. 1. The combination of sets of TCs

The categories of HOMs were named on a basis of two definitions above (see Section 1) and three new definitions below, and are illustrated in Figure 2 and Table 1.

Definition 3: The HOM that is killed by set of TCs that can kill FOM1 or FOM2 (this set of TCs ⊂ C) is called "HOM With Old Test Cases".

Definition 4: The HOM that is killed by set of TCs that cannot kill any their constituent FOM (this set of TCs ⊂ B) is called "HOM With New Test Cases".

Definition 5: The HOM that is killed by set of TCs which has some TCs ⊂ B and some others TCs ⊂ A or ⊂ C is called "HOM With Mixed Test Cases".

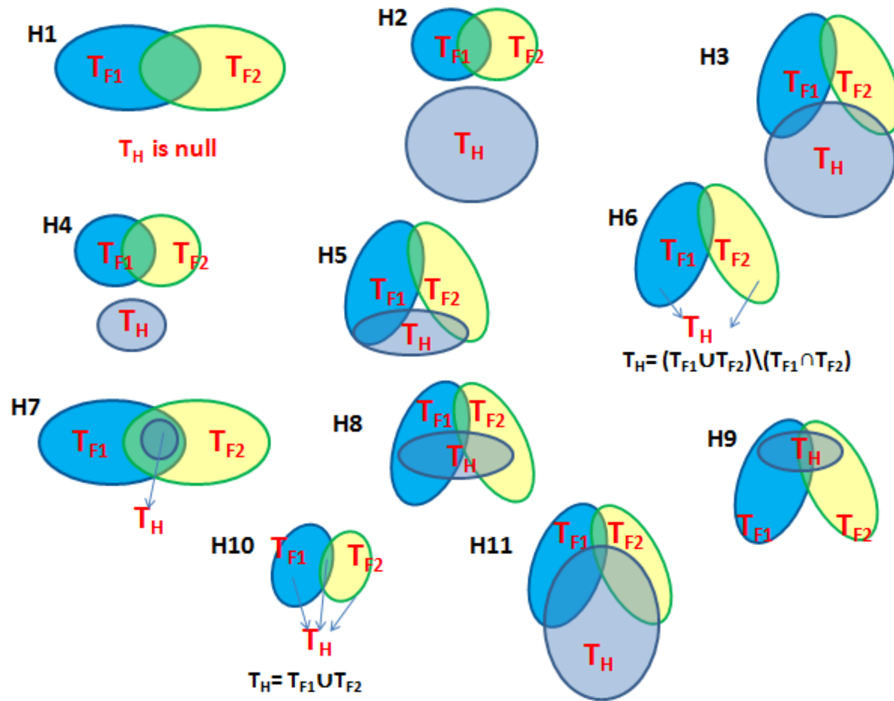

Fig. 2.11 categories of HOM based on the combination of sets of test cases.

Table 1. Elevencategoriesof HOMs

Case	HOM is
H1	Equivalent Mutant (T_H is null)
H2	Non-Subsuming, De-coupled and With New TCs.
H3	Non-Subsuming, De-coupled and With Mixed TCs.
H4	Weakly Subsuming, De-coupled and With New TCs.
H5	Weakly Subsuming, De-coupled and With Mixed TCs.
H6	Weakly Subsuming, De-coupled and With Old TCs.
H7	**Strongly Subsuming and Coupled**
H8	Weakly Subsuming, Coupled and With Mixed TCs.
H9	Weakly Subsuming, Coupled and With Old TCs.
H10	Non-Subsuming, Coupled and With Old TCs.
H11	Non-Subsuming, Coupled and With Mixed TCs.

3 Identify HOMs based on objective and fitness functions

We propose objectives and fitness function in order to identify HOMs, especially the most valuable SSHOMs. Each SSHOM represents an optimal solution that we want to find.

The set of TCs that kill HOM (TH) can be divided into 3 subsets:

- The first subset that can kill HOM and all its constituent FOMs (subset A in Fig. 1)
- The second subset that kill HOM and can kill FOM1 or FOM2 (subset C in Fig. 1)
- The third subset that only kill HOM (subset B in Fig. 1)

When the subsets B, C are empty and subset A is not empty, HOM is Strongly Subsuming HOM. In this case, the number of TCs that kill SSHOM is as small as possible. If all 3 subsets are empty, HOM is equivalent HOM. If A and C are empty, B is not empty, we call HOM is "HOM With New Test Cases". By contrast, if B is empty but A and C are not empty, we call HOM is "HOM With Old Test Cases". And if B is not empty, at least one of A or C is not empty, we call HOM is "HOM With Mixed Test Cases".

Hence, we proposed three objectives and one fitness function (see Equation 4) to apply to multi objectives optimization algorithm for searching for Strongly Subsuming HOMs.

Objective 1: Minimum number of TCs that kill HOM and also kill all its constituent FOMs (Minimum value of OB1 in Equation 1).

Objective 2: Minimum number of TCs which kill HOM but cannot kill any their constituent FOM (Minimum value of OB2 in Equation 2).

Objective 3: Minimum number of TCs which kill HOM and can kill FOM1 or FOM2 (Minimum value of OB3 in Equation 3).

$$OB1 = \frac{T_H \cap T_{F1} \cap T_{F2}}{T_H} \tag{1}$$

$$OB2 = \frac{T_H \setminus (T_{F1} \cup T_{F2})}{T_H} \tag{2}$$

$$OB3 = \frac{(T_H \cap (T_{F1} \cup T_{F2})) \setminus (T_{F1} \cap T_{F2})}{T_H} \tag{3}$$

$$fitness(H) = \frac{\#T_H}{\#(T_{F1} \cup T_{F2})} \tag{4}$$

The values of OB1, OB2, OB3 and fitness functions lie between 0 and 1. Generated HOMs are identified on a basis of that values as shown in Table 2.

Table 2. Identilfy HOMs based on objective and fitness functions.

Case	Value of OB1 (V1)	Value of OB2 (V2)	Value of OB3 (V3)	Value of fitness function (F)	HOM is
H1	0	0	0	0	Equivalent Mutant (T_H is null)
H2	0	1	0	>=1	Non-Subsuming, De-coupled and With New TCs.
H3	0	0<V2<1	0<V3<1	>=1	Non-Subsuming, De-coupled and With Mixed TCs.
H4	0	1	0	<1	Weakly Subsuming, De-coupled and With New TCs.
H5	0	0<V2<1	0<V3<1	<1	Weakly Subsuming, De-coupled and With Mixed TCs.
H6	0	0	1	<=1	Weakly Subsuming, De-coupled and With Old TCs.
H7	**0<V1<=1**	**0**	**0**	**<=1**	**Strongly Subsuming and Coupled**
H8	0<V1<=1	0<V2<1	0<=V2<1	<1	Weakly Subsuming, Coupled and With Mixed TCs.
H9	0<V1<=1	0	0<V2<1	<1	Weakly Subsuming, Coupled and With Old TCs.
H10	0<V1<=1	0	0<V2<1	1	Non-Subsuming, Coupled and With Old TCs.
H11	0<V1<=1	0<V2<1	0<=V2<1	>=1	Non-Subsuming, Coupled and With Mixed TCs.

4 Experimental evaluation

4.1 Research questions

We posed the following research questions (RQ) that the study will address(Section 4.4 provides answers):

RQ1: What is mutation score indicator (MSI) for each project under test?

RQ2: How many HOMs are Non-Subsuming and De-coupled?

Non-Subsuming and De-coupled HOMs are the HOMs which are easier to kill than their constituent FOMs and the set of test cases which kill HOMs cannot kills simultaneously their constituent FOMs.

RQ3: What is the ratio of subsuming HOMs to total number of HOMs as well as ratio of strongly subsuming HOMs to Subsuming HOMs?

4.2 Software under test and supporting tool

For our experiment, we used three open source projects (see Table 3), which are real-world software. The following projects have been selected for the experiment:

- ***BeanBin*** is a tool to make persisting EJB (Enterprise JavaBeans) 3.0 entity beans easier.
- ***Barbecue*** is a library that provides the means to create barcodes for Java applications.
- ***JWBF***(Java Wiki Bot Framework) is a library to maintain Wikis like Wikipedia based on MediaWiki and provides methods to connect, modify and read collections of articles, to help created wiki bot.

To implement our experiment, we have used Judy[11] [16], a mutation testing tool for Java language programming. It has the largest set of mutation operators, support for HOM generation, HOM execution and mutation analysis. The list of all mutation operators available in Judy [11][16] is presented in Table 4.

Table 3. Software under test

Project	**NOC**	**LOC**	**# Test cases**
BeanBin (http://beanbin.sourceforge.net)	72	5925	68
Barbecue (http://barbecue.sourceforge.net)	57	23996	190
JWBF (http://jwbf.sourceforge.net)	51	13572	305

Table 4. Java mutation operators available in Judy

AIR	AIR_ Add	Replaces basic binary arithmetic instructions with ADD
	AIR_ Div	Replaces basic binary arithmetic instructions with DIV
	AIR_ LeftOperand	Replaces basic binary arithmetic instructions with their left operands
	AIR_ Mul	Replaces basic binary arithmetic instructions with MUL
	AIR_ Rem	Replaces basic binary arithmetic instructions with REM
	AIR_ RightOperand	Replaces basic binary arithmetic instructions with their right operands
	AIR_ Sub	Replaces basic binary arithmetic instructions with SUB
JIR	JIR_ Ifeq	Replaces jump instructions with IFEQ (IF_ ICMPEQ, IF_ ACMPEQ)
	JIR_ Ifge	Replaces jump instructions with IFGE (IF_ ICMPGE)
	JIR_ Ifgt	Replaces jump instructions with IFGT (IF_ ICMPGT)
	JIR_ Ifle	Replaces jump instructions with IFLE (IF_ ICMPLE)
	JIR_ Iflt	Replaces jump instructions with IFLT (IF_ ICMPLT)
	JIR_ Ifne	Replaces jump instructions with IFNE (IF_ ICMPNE, IF_ ACMPNE)
	JIR_ Ifnull	Replaces jump instruction IFNULL with IFNONNULL and vice-versa
LIR	LIR_ And	Replaces binary logical instructions with AND
	LIR_ LeftOperand	Replaces binary logical instructions with their left operands
	LIR_ Or Replaces	binary logical instructions with OR
	LIR_ RightOperand	Replaces binary logical instructions with their right operands
	LIR_ Xor	Replaces binary logical instructions with XOR
SIR	SIR_ LeftOperand	Replaces shift instructions with their left operands
	SIR_ Shl	Replaces shift instructions with SHL
	SIR_ Shr	Replaces shift instructions with SHR
	SIR_ Ushr	Replaces shift instructions with USHR
Inheritance	IOD	Deletes overriding method

	IOP	Relocates calls to overridden method
	IOR	Renames overridden method
	IPC	Deletes super constructor call
	ISD	Deletes super keyword before fields and methods calls
	ISI	Inserts super keyword before fields and methods calls
Polymorphism	OAC	Changes order or number of arguments in method invocations
	OMD	Deletes overloading method declarations, one at a time
	OMR	Changes overloading method
	PLD	Changes local variable type to super class of original type
	PNC	Calls new with child class type
	PPD	Changes parameter type to super class of original type
	PRV	Changes operands of reference assignment
Java-Specific Features	EAM	Changes an access or method name to other compatible access or method names
	EMM	Changes a modifier method name to other compatible modifier method names
	EOA	Replaces reference assignment with content assignment (clone) and vice-versa
	EOC	Replaces reference comparison with content comparison (equals) and vice-versa
	JDC	Deletes the implemented default constructor
	JID	Deletes field initialization
	JTD	Deletes this keyword when field has the same name as parameter
	JTI	Inserts this keyword when field has the same name as parameter
Jumble-Based	Arithmetics	Mutates arithmetic instructions
	Jumps	Mutates conditional instructions
	Returns	Mutates return values
	Increments	Mutates increments

4.3 Multi-objective optimization algorithm

With the objective and fitness functions above, we use NSGA-II algorithm to generate HOMs and searching for Strongly Subsuming HOMs. NSGA-II is the second version of the Non-dominated Sorting Genetic Algorithm that was proposed by Deb et al. [17] for solving non-convex and non-smooth single and multi objective optimization problems. Its main features are: it uses an elitist principle; it emphasizes non-dominated solutions; and it uses an explicit diversity preserving mechanism.

The general scheme of NSGA-II to generate and evaluate HOMs in pseudo-code as follows:

```
Input: List of FOMs
Output: List of HOMs
Create HOMs from list of FOMs;
fitness(HOMs);
rank(HOMs)
```

```
while not (Stop Condition()) do
  Select(HOMs);
  Crossover(HOMs);
  Mutate(HOMs);
  fitness(HOMs);
  combine (HOMs parent and child);
//Recombination;//
  rank (HOMs after combined)
  select individuals (HOMs)
end while
```

4.4 Results and analysis

The following table shows the number of generated HOMs as well as HOMs in each category (defined in Section 3) in different projects under test (using NSGA-II algorithm with the proposed objectives and fitness function).

Table 5. Number of 11 categories of generated HOMs

Project	BeanBin		Barbecue		JWBF	
	Number	%	Number	%	Number	%
Generated HOMs	**402**	**100**	**891**	**100**	**238**	**100**
H1	259	64.43	251	28.17	13	5.46
H2	0	0	1	0.11	0	0
H3	7	1.74	90	10.10	6	2.52
H4	38	9.45	18	2.02	0	0
H5	0	0	9	1.01	0	0
H6	55	13.68	458	51.40	193	81.09
H7	**15**	**3.73**	**44**	**4.94**	**4**	**1.68**
H8	0	0	0	0	0	0
H9	0	0	5	0.56	2	0.84
H10	28	6.97	9	1.01	17	7.14
H11	0	0	6	0.67	3	1.26

Answer to RQ1.

We used the same full set of mutation operators of Judy for 3 projects. Because of each project has different source code including different operators. So, during the process generating HOMs, there are 3 different sets of operators used. Table 6 lists mutation operators used to generate HOMs for each project under test and 15 is the maximum mutation order in our experiment. MSI for each project under test after is presented in Table 7. JWBF is project with the highest MSI.

Table 6. Mutation operators used

Project	Operators
BeanBin	JIR_Ifle;JIR_Ifeq;JIR_Iflt;JIR_Ifne;JIR_Ifge;JIR_Iflt;JIR_Ifnull; AIR_Div;AIR_LeftOperand;AIR_RightOperand;AIR_Rem;AIR_Mul; EAM;EOC;PLD;CCE;LSF;REV;ISI;JTD;JTI;OAC;DUL;
Barbecue	JIR_Ifge;JIR_Ifeq;JIR_Ifgt;JIR_Ifne;JIR_Iflt;JIR_Ifle;JIR_Ifnull; AIR_Add;AIR_Div;AIR_Sub;AIR_LeftOperand;AIR_RightOperand; AIR_Mul;AIR__Rem; JTI;JTD;JID;JDC;EAM;OAC;EOC;FBD;IPC;EGE;EMM;CCE;LSF;REV
JWBF	JIR_Ifge;JIR_Ifeq;JIR_Ifgt;JIR_Ifne;JIR_Iflt;JIR_Ifle; EAM;EOA;JTD;JTI;JID;PRV;OAC;PLD;

Table 7.MSI for each project under test

Project	Total HOMs	Killed HOMs	MSI
BeanBin	402	146	36.32 %
Barbecue	891	629	70.59%
JWBF	238	225	94.54%

Answer to RQ2.

RQ2 is designed to evaluate percentage of Non-Subsuming and De-coupled HOMs. These HOMs are “not good” because they are the HOMs which are easier to kill than their constituent FOMs and the set of test cases which kill HOMs cannot kills simultaneously their constituent FOMs. Our results indicate that the percentage of these HOMs is not large. The percentage of Non-Subsuming and De-coupled HOMs corresponding to the BeanBin, Barbecue, and JWBF projects are 1.74%, 10.21%, 2.52%, respectively.

Answer to RQ3.

Table 8. The percentage of subsuming HOMs and strongly subsuming HOMs

Project	**Ratio of Subsuming HOMs to total HOMs**	**Ratio of Strongly Subsuming to Subsuming HOMs**
BeanBin	26.9%	14%
Barbecue	60%	8.2%
JWBF	83.6%	2%

The ratio of subsuming HOMs to total generated HOMs is fairly high. This indicates that we can find the mutants that are harder to kill and more realistic (reflecting complex faults) than FOMs by applying multi objectives optimization algorithm. Ratio of strongly subsuming HOMs to subsuming HOMs is smaller, but they can be used to replace a large number of FOMs without loss of test effectiveness. In our study, the number of FOMs corresponding to number of Strongly Subsuming HOMs which can

be used to replace FOMs for the JWBF, BeanBin and Barbecue projects are 11-4, 40-15 and 143-44, respectively.

5 Conclusion and future work

We proposed a new, extended classification of HOMs based on the combination of set of test cases (TCs) that kill HOM and sets of TCs which kill FOMs. Furthermore, we presented the objectives and the fitness function to apply multi-objective optimization algorithm NSGA-II to search for strongly subsuming HOMs and ten other types of HOMs. The results indicate that our approach can be useful in searching for strongly subsuming HOMs. However, number of equivalent HOMs still is large. In this case, equivalent HOMs cannot be killed by any test case of the given set of test cases T (see Section 2). It means that they could be killed by some test cases that not belong to T. So, in the future, we will research to improve our approach to assess whether HOMs are equivalent or not. In addition, we will apply other search-based algorithms based on our proposed objectives and fitness function in order to compare the effectiveness of different algorithms for searching for strongly subsuming HOMs.

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. 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)
6. 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)
7. 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)
8. L. Madeyski, "On the effects of pair programming on thoroughness and fault-finding effectiveness of unit tests," Lecture Notes in Computer Science, vol. 4589, pp. 207–221, 2007. DOI: 10.1007/978-3-540-73460-4_20
9. L. Madeyski, "The impact of pair programming on thoroughness and fault detection effectiveness of unit tests suites," Wiley, Software Process: Improvement and Practice, vol. 13, no. 3, pp. 281–295, 2008. DOI: 10.1002/spip.382
10. L. Madeyski, "The impact of test-first programming on branch coverage and mutation score indicator of unit tests: An experiment," Information and Software Technology, vol. 52, no. 2, pp. 169–184, 2010. DOI: 10.1016/j.infsof.2009.08.007

11. 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*, 40 (1), pp. 23-42, 2014. DOI: 10.1109/TSE.2013.44
12. Mresa, E.S., Bottaci, L.: Efficiency of mutation operators and selective mutation strategies: An empirical study. Software Testing, Verification and Reliability (1999)
13. 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)
14. 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)
15. 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)
16. Madeyski, L., Radyk, N.: Judy - a mutation testing tool for Java.IET Software 4(1): 32-42 (2010).DOI: 10.1049/iet-sen.2008.0038
17. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. IEEE transactions on evolutionary computation, vol.. 6, No. 2, April 2002.
18. Nguyen, Q.V., Madeyski, L.: Problems of Mutation Testing and Higher Order Mutation Testing. Advanced Computational Methods for Knowledge Engineering, Advances in Intelligent Systems and Computing 282, Springer 2014.DOI: 10.1007/978-3-319-06569-4_12
19. Papadakis, M.,Yue, J., Harman M., and Traon, Y.L.: Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique, in 37th International Conference on Software Engineering (ICSE), 2015.