Toward Generalization of Mutant Clustering Results in Mutation Testing

Anna Derezińska

Abstract Mutation testing is effectively used for evaluation of test case quality but suffers from high cost required for its realization. Mutated programs are injected with program changes specified by various mutation operators. One of the methods applied to the reduction of mutant number is mutant clustering. Instead of using all generated mutants, special mutant groups are distinguished and group representatives are used in further evaluation of tests. Mutant clustering gave some promising results for C programs. In case of object-oriented programs with standard and object-oriented operators the results were positive but not superior to other cost reduction techniques. An open issue is interpretation of mutant clustering results and their generalization to other projects in terms of used mutation operators. In this paper, three metrics are proposed to comprehend mutation clustering. Experimental results are analyzed toward usefulness of mutants created by various operators, their frequency, and dependency. The evaluation result confirms applicability of the metrics, and the practical guidelines about the mutation operators are concluded from the experimental data.

Keywords Mutation testing · Mutant clustering · Mutation operators · C#

1 Introduction

Mutation testing is a method for evaluating a quality of a test case suite and/or creating a set of test cases [1]. The main idea originates from the fault injection techniques. A change is introduced to a program under test. The change represents typically a possible mistake made by a programmer. It is assumed that the change could not be revealed by a simple program compilation. A changed program is called a *mutant*.

The main obstacle in application of a mutation testing process is its high cost. Therefore there are many approaches to its cost reduction [1, 2]. Some of them

A. Derezińska (✉)
Institute of Computer Science, Warsaw University of Technology,
Nowowiejska 15/19, 00-665 Warsaw, Poland
e-mail: A.Derezinska@ii.pw.edu.pl

A. Wiliński et al. (eds.), *Soft Computing in Computer and Information Science*,
Advances in Intelligent Systems and Computing 342,
DOI 10.1007/978-3-319-15147-2_33

are based on lowering of the number of generated mutants and/or the number of performed test cases. Mutant clustering is one of the cost reduction methods that was considered for the mutation testing process [3, 4].

The first results on mutant clustering for object-oriented programs were reported in [5]. A general experimental scenario was proposed for evaluation of the tradeoff between mutation score accuracy and the complexity of a mutation testing process expressed in a number of generated mutants and a number of test cases. The scenario was adapted to three cost reduction techniques: selection of mutants, mutant sampling, and clustering. The detailed results of mutant clustering experiments for C# programs, the experimental scenario, and evaluation of a quality metric are given in [6].

This paper addresses another problem of mutant clustering. It is a question of how we can generalize the results of experiments on mutant clustering, which might be useful for other projects. Especially, it is interesting how to evaluate relations of mutants generated by different mutation operators. Therefore, three new metrics were developed, dealing with usefulness of mutants generated by a given operator, their frequency, and dependency among mutants. Based on these metrics and data gathered in our previous experiments [5, 6] the approach was applied. We could examine in a quantitative way the differences between standard and object-oriented operators, distinguish a pair of complementary operators, and classify operators that cannot be omitted in order to preserve the mutation result accuracy.

The paper is organized as follows: Sect. 2 describes briefly the basic notion of mutation testing and mutant clustering method. In Sect. 3 metrics used for analysis of mutant clustering results are introduced and illustrated by an example. Section 4 presents an experiment overview and results of the conducted experiments. Finally, Sect. 5 concludes the work.

2 Background

In this section basic concepts of mutation testing as well as an idea of mutant clustering and related works related to it are discussed.

2.1 Mutation Testing

In mutation testing, a program change is specified by a *mutation program operator* and introduced in an automatic way using a mutation tool. *Standard*, or so-called traditional, mutation operators deal with the common programming features, typical to all programming languages, like arithmetic, logical and relational operators, assignment statements, constant usage, etc. Different specialized programming features are also covered by the devoted mutation operators. Features characteristic to object-oriented languages (OO in short) are handled by *object-oriented mutation*

operators proposed, for example, for Java [7] or C# [8, 9]. If a mutation operator is applied only once and in one place of a program, we speak about *first order mutation*.

Evaluation of a test suite is performed in a mutation testing process. For a given program and a set of selected mutation operators, a set of mutants is created. The mutants are run against tests from a test suite under concern. If a mutant behavior is different from the behavior of the original program, the mutant is said to be *killed*. Tests that are able to kill mutants should be good at revealing mistakes represented by the mutation operators. A mutation testing result, called a *mutation score* (MS), is calculated as a ratio of the number of all killed mutants over the number of all nonequivalent mutants. A mutant is *equivalent* if its behavior cannot be distinguished from the original program by any test. In many practical cases, instead of an exact mutation score, its approximate value is calculated, because it is not possible to classify exactly all equivalent mutants in an automatic way.

2.2 Mutant Clustering

There are many approaches to reduction of mutation testing costs based on lowering of considered mutants and therefore reducing also the number of test runs [1, 2]. One such analyzed solution was mutant clustering [3, 4].

The main idea of the mutant clustering originates on the concept of equivalence partitioning. A set of all mutants of a program is divided into groups, called clusters. The division is realized in the context of a given test set, similarly as in mutation score evaluation. Each group is characterized by the similar ability of being killed by the same subset of tests. Allocation of a mutant to a group can be realized by a clustering algorithm such as agglomerative hierarchical or K-means clustering [3, 10].

The mutant clustering is specified for a given set of mutants of a program under test, and a given set of tests. A threshold K denotes a resemblance between mutant groups. Two groups are said to be *similar with K degree*, if the number of tests that kill at least one mutant from one group and kill none mutant from the former group equals K.

Next, in the mutation testing process, instead of all mutants, only one mutant for each group is used. This mutant represents the group (cluster) that should have the comparable features, as far as the subset of tests associated with this group is concerned. Usage of a reduced number of mutants lowers the mutation costs. However, the accuracy of the mutation score can be declined.

2.3 Related Works

Primary experiments on mutant clustering in mutation testing were conducted for C programs [3]. They reported considerable potential benefits, for example, usage of 13 % of all mutants and 8 % of tests gave a mutation score of a high accuracy (99 %). However, this result referred to a simple, not object-oriented programming language and only standard mutation operators, which usually are more redundant.

The above-mentioned result was based on full data, i.e., all mutants run against all tests. The practical solution to mutation clustering based on a static domain analysis was presented in [4]. The proof of concept was illustrated by a small Java program, for which satisfactory results were obtained, namely after running 25 % of mutants with 62 % of tests the mutation score was equal to 94 % of the exact mutation score.

Mutant clustering in the context of object-oriented operators was studied for the first time in the experimental process for comparing of different cost reduction techniques [5]. The detailed analysis of mutant clustering was discussed in [6]. The quantitative data of this experiment is recalled in Sect. 4.1. This paper provides further methods of clustering data analysis in order to generalize the results to other projects.

The research on cost reduction methods applied to C# programs was performed only by the author of [5, 6]. Most of the other work on object-oriented programs was done for Java programs [11–13], but the clustering method was not considered.

3 Metrics for Generalization of Clustering Results

Cluster of mutants includes mutants generated by various mutation operators. Many mutants created with the same mutation operator (Op) can contribute to the same cluster $\{Op1, Op2, \ldots\}$. Therefore, there can exist clusters that are mainly constituted by mutants of selected mutation operators. These mutants are the most probable representative of these clusters.

Mutants of the same operator can also be met in many clusters. Some pairs of operators can be associated and encounter in the same clusters. In such case it could be possible to omit one of the operators.

In order to quantitatively evaluate such phenomena the following metrics were used.

3.1 Metric Definitions

Three additional metrics were proposed in order to evaluate and compare the clustering results. Each metric is calculated for a mutation operator (Op).

The first metric is *usefulness of mutants* (UM). It calculates how big a subset of mutants is that are useful in the context of a given operator.

$$UM(Op) = \frac{NG(Op)}{NM(Op)} \tag{1}$$

where

$NG(Op)$—the number of groups, in which at least one mutant exists that was created using the Op mutation operator,

$NM(Op)$—the number of all mutants generated using the Op mutation operator.

The second metric, so-called *frequency* (*FR*), examines frequency of an operator occurrence. It calculates the amount of groups, which includes at least one mutant designed by the operator in relation to all group number.

$$FR(Op) = \frac{NG(Op)}{NG_{All}} \tag{2}$$

where $NG(Op)$—as above, and NG_{All}—number of all groups.

The third metric is called *dependency* (*DEP*). It evaluates dependency of an ordered pair of mutation operators.

$$DEP(Op_1, Op_2) = \frac{NPM(Op_1, Op_2)}{NM(Op_1)} \tag{3}$$

where

$NM(Op_1)$—a number of all mutants generated using the Op_1 mutation operator.

$NPM(Op_1, Op_2)$—a number of occurrences of mutant pairs created with Op_1 and Op_2 operators. This value is calculated as a sum of all other groups of a minimum of two numbers: a number of mutants of a given group created using Op_1, and an analogous number of mutants created by the second operator Op_2.

$$NPM(Op_1, Op_2) = \sum_{g \in G} \min(NM(g, Op_1), NM(g, Op_2)) \tag{4}$$

where

$NM(g, Op_1)$—a number of mutants from the group g that were generated using the Op_1 mutation operator.

It should be noticed, that the operator dependency metric is not symmetric, i.e. $DEP(Op_1, Op_2) \neq DEP(Op_2, Op_1)$.

3.2 Example

The metrics will be illustrated with a simple example. Three mutation operators were used for generation of mutants: EOC, IOP, and EXS. (The full operator names are listed in Table 1). Using EOC operator five mutants were created. Four mutants were generated with IOP operator and one mutant with EXS.

After performing an algorithm of mutant clustering four groups of mutants were specified. The result groups consist of the following mutants:

$G1 = \{EOC1, EOC2, IOP1, IOP2\}$
$G2 = \{EOC4, EOC5, IOP4\}$
$G3 = \{EOC3\}$
$G4 = \{IOP3, EXS1\}$

Table 1 Standard and object-oriented mutation operators (C# supported by CREAM v.3)

No	Operator type	Abbr.	Name of mutation operator
1	Standard	ABS	Absolute value insertion
2	Standard	AOR	Arithmetic operator replacement ($+, -, *, /, \%$)
3	Standard	ASR	Assignment operator replacement ($=, +=, -=, /=, *=$)
4	Standard	LCR	Logical connector replacement (&&, ||)
5	Standard	LOR	Logical operator replacement (&, |,^)
6	Standard	ROR	Relational operator replacement ($<, <=, >, >=, ==, !=$)
7	Standard	UOI	Unary operator insertion ($+, -, !, \sim$)
8	Standard	UOR	Unary operator replacement ($++, --$)
1	Object-oriented	DMC	Delegated method change
2	Object-oriented	EHR	Exception handler removal
3	Object-oriented	EOA	Reference assignment and content assignment replacement
4	Object-oriented	EOC	Reference comparison and content comparison replacement
5	Object-oriented	EXS	Exception swallowing
6	Object-oriented	IHD	Hiding variable deletion
7	Object-oriented	IHI	Hiding variable insertion
8	Object-oriented	IOD	Overriding method deletion
9	Object-oriented	IOK	Overriding method substitution
10	Object-oriented	IOP	Overriding method calling position change
11	Object-oriented	IPC	Explicit call of a parent's constructor deletion
12	Object-oriented	ISK	Base keyword deletion
13	Object-oriented	JID	Ember variable initialization deletion
14	Object-oriented	JTD	This keyword deletion
15	Object-oriented	OAO	Argument order change
16	Object-oriented	OMR	Overloading method contents change
17	Object-oriented	PRM	Property replacement with member field
18	Object-oriented	PRV	Reference assignment with other compatible type

The usefulness metric *UM* was calculated for each operator in the following way:

$$\begin{aligned} UM(EOC) &= \frac{NG(EOC)}{NM(EOC)} = \frac{3}{5} = 0.6 \\ UM(IOP) &= \frac{NG(IOP)}{NM(IOP)} = \frac{3}{4} = 0.75 \\ UM(EXS) &= \frac{NG(EXS)}{NM(EXS)} = \frac{1}{1} = 1.0 \end{aligned} \tag{5}$$

The calculated values can be interpreted as a useful part of mutants. For example, 60 % of mutants could be selected for the EOC operator and still in each group there would be at least one mutant created by this operator. However, all mutants

(100 %) generated by the EXS operator are indispensable in order to ensure the same condition.

The frequency metric calculated for the example mutants gives the following values:

$$\begin{aligned} FR(EOC) &= \frac{NG(EOC)}{NG_{All}} = \frac{3}{4} = 0.75 \\ FR(IOP) &= \frac{NG(IOP)}{NG_{All}} = \frac{3}{4} = 0.75 \\ FR(EXS) &= \frac{NG(EXS)}{NG_{All}} = \frac{1}{4} = 0.25 \end{aligned} \tag{6}$$

For a given operator, the metric assesses the frequency of mutants belonging to groups. For example, the metric of EOC is equal to 0.75. It means that 75 % of all groups include at least one mutant created using this operator.

Finally, the dependency metric is calculated for any ordered pair of mutation operators. We choose for example two operators EOC and IOP. Because the metric is not symmetric, two ordered pairs are considered: (EOC, IOP) and (IOP, EOC).

$$\begin{aligned} DEP(EOC, IOP) &= \frac{NPM(EOC, IOP)}{NM(EOC)} = \frac{3}{5} = 0.6 \\ DEP(IOP, EOC) &= \frac{NPM(IOP, EOC)}{NM(IOP)} = \frac{3}{4} = 0.75 \end{aligned} \tag{7}$$

Based on the first value (0.6) we can deduce that 60 % of mutants created by EOC can be substituted by IOP mutants. In the opposite case the value is different and is equal to 0.75. This means that 75 % of IOP mutants have a pair of EOC mutants in a group. Comparing both values of the metric, we can conclude that in this example it is better to substitute operator IOP by EOC (0.75) than vice versa (0.6).

4 Experiments on Mutation Clustering

Evaluation of the approach will be presented on experimental data. The metrics were applied to the analysis of mutation clustering results gathered in the experiments on standard and object-oriented mutation of C# programs [5, 6].

4.1 Experiment Setup

Data for the mutant clustering and evaluation of the metrics were collected in experiments carried out with the CREAM v3 tool. CREAM is a mutation testing tool for C# programs [14]. It was the first tool that supported object-oriented mutation

operators for C# programs [15, 16]. Its third version was enhanced with an extension for efficient performing and evaluation experiments on cost reduction techniques: selection of mutants, mutant sampling, and clustering [5]. The tool supports 18 object-oriented operators and eight standard ones (Table 1).

The experiments were conducted on three commonly used open-source programs, Enterprise Logging (http://entlib.codeplex.com), Castle (http://www.castleproject.org) and Mono-Gendarme (http://www.mono-project.com/Gendarme). All first order mutants were generated for the mutation operators given in Table 1. Additionally, only mutants covered by tests from a given test suite were considered, as not covered mutants were not able to be killed by tests. Then all mutants were run against all test cases. The collected results were stored and used in the evaluation process of the cost reduction techniques [5]. For different cost reduction method the appropriate quality measures were calculated that allow to express the tradeoff between mutation score and the number of mutants and the number of tests.

The detailed results of the basic quality analysis of the mutation clustering approach are presented in [6]. For all mutants the agglomerative clustering algorithm was applied. Mutants generated by standard mutation operators and by object-oriented ones (in short—standard and OO mutants) were analyzed separately. The groups of mutants were formulated for the K degree of the clustering algorithm varying from 0 to 19. According to the quality analysis the best results were obtained, for $K = 1$ in case of object-oriented operators and $K = 2$ in case of the standard operators, assuming that the mutation score adequacy contributes of 60 % to the overall quality, whereas number of mutants and number of tests of 20 % each. The experiments showed that it was possible to use 32 % of OO mutants and 18 % of tests to obtain the mutation score of 97 % close to the original one (i.e. calculated using all OO mutants and all test). The analogues data for the best results of standard mutation was 19 % of mutants, 22 % of tests and 91 % of mutation score accuracy.

4.2 Evaluation of Mutation Clustering Results

Mutation data from the above-mentioned experiments were used in the further evaluation of mutation clustering results addressing the generalization problem. The evaluation was based on the metrics specified in Sect. 3. The results were analyzed separately for standard and object-oriented mutations. The metrics were calculated in respect to all mutation operators used in experiments.

Results of two metrics, *usefulness of mutants* and *frequency*, calculated for the subject programs and their average values are shown in Table 2. The upper part of the table includes values of standard operators, whereas the lower part gives data for OO operators. Empty places, denoted by '−' character, correspond to cases when no mutant was generated from a given program (column) using this kind of operator (row).

Analyzing the first metric for object-oriented operators, we can observe that in most of the cases the value of usefulness of mutants is relatively high. A value

Table 2 Usefulness of mutants (UM) and frequency (FR) metrics for standard and object-oriented mutation operators

Oper.	Usefulness of mutants metric (*UM*)				Frequency metric (*FR*)			
	Logging	Castle	Gendarme	Average	Logging	Castle	Gendarme	Average
ABS	0.57	0.89	1.00	0.82	0.01	0.01	0.00	0.01
AOR	0.11	0.42	0.33	0.29	0.10	0.01	0.02	0.04
ASR	0.42	0.53	0.56	0.50	0.12	0.03	0.03	0.06
LCR	0.93	0.82	0.73	0.83	0.07	0.17	0.18	0.14
LOR	–	–	0.64	0.64	–	–	0.01	0.01
ROR	0.54	0.33	0.33	0.40	0.22	0.21	0.18	0.20
UOI	0.50	0.61	0.46	0.52	0.76	0.75	0.81	0.77
UOR	0.40	0.42	0.42	0.41	0.02	0.08	0.04	0.05
DMC	–	–	–	–	–	–	–	–
EHR	1.00	1.00	0.60	0.87	0.02	0.01	0.01	0.01
EOA	–	1.00	1.00	1.00	–	0.01	0.01	0.01
EOC	0.98	0.62	0.65	0.75	0.14	0.34	0.37	0.28
EXS	1.00	1.00	–	1.00	0.00	0.01	–	0.01
IHD	–	–	–	–	–	–	–	–
IHI	–	–	–	–	–	–	–	–
IOD	1.00	1.00	0.78	0.93	0.07	0.03	0.02	0.04
IOK	1.00	1.00	0.75	0.92	0.06	0.02	0.02	0.04
IOP	0.43	1.00	1.00	0.81	0.01	0.01	0.08	0.03
IPC	0.97	0.45	–	0.71	0.12	0.04	–	0.08
ISK	0.81	0.64	1.00	0.82	0.09	0.02	0.11	0.07
JID	0.66	0.67	0.65	0.66	0.07	0.18	0.31	0.19
JTD	0.92	1.00	–	0.96	0.16	0.10	–	0.13
OAO	0.46	0.63	0.48	0.53	0.18	0.19	0.11	0.16
OMR	0.56	0.84	–	0.70	0.03	0.11	–	0.07
PRM	0.64	0.83	0.90	0.79	0.02	0.03	0.03	0.03
PRV	0.24	0.37	0.34	0.32	0.14	0.09	0.04	0.09

can be counted as high if it is bigger than 0.8 for at least one program. Eight OO operators have at least one 1.0 (100 %) value, which means that for this program all mutants generated by this operator contribute as group representatives and could not be omitted in the mutation score analysis.

However, we can observe the PRV operator (*Reference Assignment with other Compatible Type*) for which this metric is low, i.e., about 0.3 for each program. Generating only about 32 % of all PRV mutants it is possible to create the same groups considering their member operators. Moreover, analyzing the frequency metric for PRV we have found that in average 9 % of groups includes at least one PRV mutant. This result is medium high in comparison to other operators but not negligible.

In conclusion, it is worthwhile to limit the number of PRV mutants, as it is possible to reduce the mutant number considerably without loss of the mutation score accuracy.

Comparing results of the usefulness metric for object-oriented and standard operators we can observe that in general the values of standard operators are lower than the object-oriented ones. Only two standard operators (ABS and LCR) have a high value of the first metric. This confirms the other results [5, 11, 17] that among standard mutants can be more surplus (redundant) mutants than in the object-oriented mutants. It should be noted that this effect is visible although the set of standard operators of CREAM and therefore used in this experiment was very limited. It was based mainly on the operators classified as selective in the standard operator analysis [17].

Analogues reasoning for the PRV operator can be performed for selected standard operates, in particular ROR and UOR. In case of these operators, according to the first metric at least 40 % of mutants should be generated for each operator.

Results of the third metric—*dependency* are shown in Table 3 for standard mutation operators and in Table 4 for object-oriented ones. The tables include values averaged over all three programs examined in experiments. Operators DMC, IHD were omitted as no mutants were generated by them in the considered programs.

Analyzing the object-oriented operators, we can see that the maximum values 0.87 and 0.79 are calculated for two operators IOK and IOD. This result denotes that most mutants generated using the IOK operator (87 %) are in the same group as mutants created by IOD operator. The opposite dependency is satisfied in 79 %. Therefore, we can assume that resigning one such operator can reduce the mutation testing cost without considerable loss of the mutation score accuracy, because they are complementary, i.e., mutants of one operator can be substituted by mutants of the second operator. The slightly better choice is selection of IOK, because DEP(IOK, IOD) > DEP(IOD, IOK).

The dependency metric calculated for other pairs of object-oriented operators give in most cases very low results (about few %) or for several pairs results about 10–20 %. Therefore we cannot point at any other pair of object-oriented operators as being dependent in general.

Considering the third metric for standard mutation operators (Table 3), we can find more results above 50 % than for the object-oriented operators. Five standard

Table 3 Dependency metric for standard operators

	ABS	AOR	ASR	LCR	LOR	ROR	UOI	UOR	Sum
ABS	–	0.11	0.19	0.15	0.33	0.48	0.75	0.19	2.2
AOR	0.00	–	0.06	0.04	0.03	0.33	0.51	0.22	1.19
ASR	0.02	0.10	–	0.07	0.04	0.25	0.60	0.09	1.17
LCR	0.00	0.02	0.04	–	0.00	0.15	0.37	0.05	0.63
LOR	0.07	0.14	0.14	0.07	–	0.64	0.57	0.36	1.99
ROR	0.00	0.05	0.04	0.05	0.02	–	0.49	0.10	0.75
UOI	0.01	0.09	0.05	0.05	0.00	0.17	–	0.04	0.41
UOR	0.01	0.12	0.07	0.08	0.05	0.44	0.53	–	1.3

Table 4 Dependency metric for object-oriented operators

	EHR	EOA	EOC	EXS	IHI	IOD	IOK	IOP	IPC	ISK	JID	JTD	OAO	OMR	PRM	PRV	SUM
EHR	–	0	0	0	–	0	0	0	0	0	0.11	0	0.22	0	0	0	0.33
EOA	0	–	0.25	0	–	0	0	0	0	0	0	0	0	0	0	0	0.25
EOC	0	0	–	0	–	0.02	0.01	0	0	0	0.05	0.04	0.02	0.01	0	0.01	0.16
EXS	0	0	0.17	–	–	0	0	0	0	0	0	0	0	0	0	0	0.17
IHI	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	0
IOD	0	0	0.16	0	–	–	0.79	0	0.03	0	0.14	0.05	0.11	0.05	0	0.02	1.35
IOK	0	0	0.14	0	–	0.87	–	0	0.03	0	0.11	0.06	0.10	0.06	0	0.02	1.39
IOP	0	0	0	0	–	0	0	–	0	0.26	0.21	0	0.26	0	0.02	0	0.75
IPC	0	0	0.02	0	–	0.01	0.01	0	–	0	0	0.02	0.02	0.02	0	0	0.1
ISK	0	0	0.03	0	–	0	0	0.05	0	–	0.06	0	0.04	0.02	0	0	0.2
JID	0	0	0.07	0	–	0.03	0.03	0.01	0	0.01	–	0.04	0.01	0.02	0	0.04	0.26
JTD	0	0	0.10	0	–	0.01	0.01	0	0.01	0	0.08	–	0.06	0.03	0.01	0	0.31
OAO	0.01	0	0.04	0	–	0.01	0.01	0.01	0	0.01	0.02	0.02	–	0	0	0.02	0.15
OMR	0	0	0.06	0	–	0.01	0.01	0	0.01	0.03	0.04	0.02	0	–	0	0.01	0.19
PRM	0	0	0.03	0	–	0	0	0.03	0	0	0.03	0.04	0.03	0	–	0.19	0.35
PRV	0	0	0.03	0	–	0	0	0	0	0	0.02	0	0.03	0.01	0.03	–	0.12

operators, namely ABS, AOR, ASR, LOR, and UOR can be partially substituted by the UOI operator. On the other hand the dependency is in the range of 50–60 % (only for ABS equal 75 %). Therefore the dependency is not as definite as in the case of the pair of IOD-IOK operators.

The last column in the tables with dependency metric includes the sum of the numbers in the corresponding row. This sum represents information about to what extent mutants created by the row operator can be substituted by any combination of the remaining operators. The lower the sum, the more reasonable it is to retain this operator in the mutation testing process.

Analyzing this sum of object-oriented operators, we can see that the highest values are for operators IOD, IOK (already recognized as complementary ones) and the IOP operator. Therefore the remaining object-oriented operators should be applied, i.e., the majority of OO operators. This evaluation confirms the fact that OO operators correspond to various advanced programming features and need more specific tests.

For the standard operators the sums are in general higher than for OO operators. Operators LCR, ROR, and UOI turned out to be the most applicable as they have the sum below zero. This result is consistent with the findings on the selective mutation in C# programs [5].

5 Conclusions

This paper presents a study on the result evaluation of mutation clustering. It copes with the question: How can the clustering results be generalized and associated with the selection of mutation operators. The problem was examined with three new metrics about mutant usefulness, frequency, and dependency in terms of mutation operators used for the mutant generation. The metrics were applied to mutant clustering results on three real-world C# programs mutated with standard and object-oriented operators.

Combining the results of usefulness and frequency metrics, we can observe that reducing the number of generated PRV mutants gives noticeable mutant cost reduction without a loss of the mutation score accuracy. It is also worthwhile to select only one operator among IOK and IOD operators. The lessons learned point at different characteristics between structural and object-oriented mutation operators. In general, less OO mutation operators can be omitted if an adequate mutation result has to be assured. This fact can be caused by the higher specialization of the OO mutation operators than the standard ones. For the standard mutation operators, even basing of a preliminary reduced set of operators (including all selective according to [17]), we can still reduce the number of generated mutants. Among standard operators the most useful were LCR, ROR, and UOI, which corresponds to the results on selective mutation in OO programs based on other methodologies [5, 11].

Acknowledgments I am thankful to M. Rudnik for his contribution to the CREAM tool development and for performing mutation testing experiments.

References

1. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. **37**(5), 649–678 (2011)
2. Usaola, M.P., Mateo, P.R.: Mutation testing cost reduction techniques: a survey. IEEE Softw. **27**(3), 80–86 (2010)
3. Hussain, S.: Mutation Clustering. Master's Thesis. King's College London, Strand, London (2008)
4. Ji, C., Chen, Z.Y., Xu, B.W., Zhao, Z.H.: A novel method of mutation clustering based on domain analysis. In: Proceedings of 21st International Conference on Software Engineering & Knowledge Engineering 422–425 (2009)
5. Derezińska, A., Rudnik, M.: Quality evaluation of object-oriented and standard mutation operators applied to C# programs. In: Furia, C.A., Nanz, S. (eds.) TOOLS Europe. LNCS, vol. 7304, pp. 42–57. Springer, Berlin (2012)
6. Derezińska, A.: A quality estimation of mutation clustering in C# programs. In: Zamojski, W. (ed.) New Results in Dependability & Computer Systems. AISC, vol. 224, pp. 119–129. Springer, Switzerland (2013)
7. Ma, Y.-S., Kwon, Y.-R., Offut, A.J.: Inter-class mutation operators for Java. In: Proceedings of 13-th International Symposium on Software Reliability Engineering, pp. 352–363. IEEE Computer Society (2002)
8. Derezińska, A.: Advanced mutation operators applicable in C# programs. In: Sacha, K. (ed.) Software Engineering Techniques: Design for Quality. IFIP, vol. 227, pp. 283–288. Springer, Boston (2006)
9. Derezińska, A.: Quality Assessment of Mutation Operators Dedicated for C# Programs. In: 6th International Conference on Quality Software, QSIC'06, Beijing, China, pp. 227–234, IEEE Computer Society Press, California (2006)
10. Jain, A.K., Murty, M.N., Flynn, P.J.: Data clustering: a review. ACM Comput. Surv. **31**(3), 264–323 (1999)
11. Hu, J., Li, N., Offutt, J.: An analysis of OO mutation operators. In: Proceedings of 4th International Conference Software Testing Verification and Validation Workshops, pp. 334–341 (2011)
12. Zhang, L., Gligoric, M., Marinov, D., Khurshid, S.: Operator-based and random mutant selection: better together. In: 28th IEEE/ACM Conference on Automated Software Engineering (ASE 2013), pp. 92–102. Palo Alto (2013)
13. Bluemke, I., Kulesza, K.: Reduction in mutation testing of Java classes. In: Proceedings of International Joint Conference on Software Technologies (ICSOFT). Vienna (2014)
14. CREAM, http://galera.ii.pw.edu.pl/~adr/CREAM/
15. Derezińska, A., Szustek, A.: Tool-supported mutation approach for verification of C# programs. In: Zamojski, W., et al. (eds.) Proceedings of International Conference on Dependability of Computer Systems, DepCoS-RELCOMEX'08, pp. 261–268 (2008)
16. Derezińska, A., Szustek, A.: Object-oriented testing capabilities and performance evaluation of the C# mutation system, In: Szmuc, T., Szpyrka, M., Zendulka, J. (eds.) CEE-SET 2009. LNCS, vol. 7054, pp. 229–242 (2012)
17. Offut, J., Rothermel, G., Zapf, C.: An experimental evaluation of selective mutation. In: Proceedings of 15th International Conference on Software Engineering, pp. 100–107 (1993)