Proceedings of the Federated Conference on Computer Science and Information Systems pp. 1315–1324
DOI: 10.15439/2017F375
ISSN 2300-5963 ACSIS, Vol. 11

# Evaluation of Mutant Sampling Criteria in Object-Oriented Mutation Testing

Anna Derezińska
Warsaw University of Technology
Institute of Computer Science
Nowowiejska 15/16, 00-665
Warsaw, Poland
Email: A.Derezinska@ii.pw.edu.pl

Marcin Rudnik
Warsaw University of Technology
Institute of Computer Science
Nowowiejska 15/16, 00-665
Warsaw, Poland

***Abstract*—Mutation testing of object-oriented programs differs from that of standard (traditional) mutation operators in accordance to the number of generated mutants and ability of tests to kill mutants. Therefore, outcomes of cost reduction analysis cannot be directly transferred from a standard mutation to an object-oriented one. Mutant sampling is one of reduction methods of the number of generated and tested mutants. We proposed different mutant sampling criteria based on equivalence partitioning in respect to object-oriented program features. The criteria were experimentally evaluated for object-oriented and standard mutation operators applied in C# programs. We compared results using a quality metric, which combines mutation score accuracy with mutation cost factors. In result, class random sampling and operator random sampling are recommended for OO and standard mutation testing, accordingly. With a reasonable decline of result accuracy, the mutant sampling technique is easily applicable in comparison to other cost reduction techniques.**

## I. Introduction

FAULT detection ability of a test suite can be measured with assistance of mutation analysis [1]. After seeding a fault into a program, its mutated variant - a *mutant* is run against tests. If any test detects a changed behavior of the mutant, it is called to be *killed* by the test suite. Capability of a test suite to reveal faults introduced through mutations is expressed by a *mutation score* (MS). It is calculated as a ratio between the number of killed mutants and the number of all non-equivalent mutants. A mutant is said to be *equivalent* with the original program if its behavior is identical and none test case can kill it.

Faults injected automatically into a program are specified with so-called *mutation operators*. Standard (traditional) operators introduce simple changes in typical expressions of general purpose languages. According to experiments with thousands of mutants on several C# programs [2], standard mutation operators are not sufficient in dealing with flows in object-oriented program structures. Such flows can be served by OO and other specialized mutation operators.

The major drawback of the mutation method is its high cost because of executing many mutants against many tests. There are many approaches trying to lower the mutation cost that are based on decreasing a number of considered mutants [1], [3], such as mutant sampling, selective mutation, higher order mutation, mutant clustering, etc. However, due to different characteristic of object-oriented mutation, their benefits could be not necessarily as promising as for standard mutation operators. For example, mutation score accuracy for mutation operator selection was worse about few to 10% for OO operators in comparison to standard ones [4].

Therefore, we undertake research on cost reduction techniques with OO operators applied to C# programs, including mutant sampling. In mutant sampling, test runs are performed on a random subset of mutants [5]. The subset includes $R$% of all mutants, where $R$ is a parameter of the method (*sampling degree*). In the opposite to other mutant selection approaches [6], none of mutation operators is discarded. Apart from this simple sampling method, we proposed and experimentally investigated five other sampling criteria based on an equivalence partitioning according to OO program structure. The sampling results were evaluated using a quality metric [4] that approximates a tradeoff between the mutation score accuracy and the mutation cost in terms of mutant number and test number. A unified investigation process was used, which helps to compare results of different programs and different cost reduction methods, as mutant selection [4] and mutation clustering [7]. The main contributions of the paper are:

- proposal and evaluation of different sampling criteria,
- comparison of sampling in regard to OO and standard mutation operators,
- quality analysis of mutant sampling results based on the quality matric that concerns an impact of a number of mutants and a number of tests,

- preformation of comprehensive experiments on mutant sampling with C# programs.

This paper is organized as follows: the next Section describes mutant sampling methodology. Section III presents details about an experimental set-up and results of the experiments carried out. The final sections present related work and conclusions.

## II. Mutant Sampling

In this section, we present methodology on which experiments were based: different criteria of mutant sampling, a flow of the investigation process, and how results are evaluated with a quality metric.

### A. Mutant Sampling Criteria

Mutant sampling was proposed by Acree [8] and Budd [9]. In a simple mutant sampling approach, a subset of mutants is randomly chosen from a defined set of mutants [5]. It will be referred as the first sampling criterion.

While taking into account a structure of an object-oriented program, different sampling criteria can also be proposed. The idea behind these sampling criteria is to divide a set of all mutants into disjoint partitions, i.e. equivalence classes. Then, random selection refers not to the whole mutant set, as in the fully random sampling, but some mutants are selected from each partition. In this way, each partition is represented in a reduced set of mutants. The criteria differ in the way such partitions are constituted. This general idea is analogous to the equivalence partitioning-based testing [10], in which selection of tests from different partitions assures a test coverage for all partitions. In this paper, the following sampling criteria have been investigated (*R* denotes a *sampling degree):*

1. *fully random* - *R%* of mutants is randomly chosen from the set of all mutants,
2. *class random* - random selection of mutants is equally distributed for all classes, i.e. for each class *R%* of its mutants is chosen,
3. *file random* - random selection of mutants is equally distributed for all files of the source code, for each file *R%* of its mutants is chosen,
4. *method random* - random selection of mutants is equally distributed for all methods of the source code, for each method *R%* of its mutants is chosen,
5. *mutation operator random* - random selection of mutants is equally distributed for all mutation operators, i.e. for each operator *R%* of mutants generated by this operator are randomly chosen,
6. *namespace random* - random selection of mutants is equally distributed for all namespaces of the source code, for each namespace *R%* of its mutants is chosen.

It should be stressed that the fifth criterion, *mutation operator random*, is not equivalent to the selective mutation [6]. In the mutant sampling according to this criterion, we use subsets of mutants generated by each considered mutation operator; whereas in the selective mutation all mutants generated by specified operators are used and mutants of remaining operators are discarded.

### B. Investigation Process

The experiment under concern investigates influence of the sampling criteria and their parameter, i.e. an amount of percentage of chosen mutants, on mutation results.

A prerequisite of the investigation process is generation of all first order mutants for a given program using a considered set of mutation operators. This set of all mutants will be denoted as $M_{All}$. Afterwards, all mutants are run against all tests from a given pool ($T_{All}$). Mutation results are referred as positions in a mutant execution matrix, where a pair <mutant *m*, test *t*> evaluates to an outcome whether the mutant *m* was killed by the test *t* or not.

After having tested all mutants with all tests, we can determine a reference mutation score (a ratio of killed mutants to nonequivalent). This measure called here *original mutation score* $MS_{orig} = MS\ (M_{All}, T_{All})$ is calculated using the mutant execution matrix. The value of $MS_{orig}$ is treated as the most accurate *MS* of the process but obtained in the most costly way - using many mutants and tests.

### C. Minimal Test Sets

A research question is whether mutant sets reduced by sampling are efficient in assessing the quality of all tests. Therefore, using a concept of minimal test sets we refer results of reduced sets to those of all possible mutants. Minimal test sets have the same ability of killing mutants and its notion can be explained in the following way.

Let assume that $M_X$ is a subset of all considered mutants $M_X \subseteq M_{All}$ that satisfies the following condition: if all tests from a given test pool $T_{All}$ are used, this subset determines the maximal mutation score $MS_{Xmax} = MS\ (M_X, T_{All})$. However, it could be possible to obtain the same mutation score using a smaller number of tests than $|T_{All}|$ (where $|S|$ states for the cardinality of set *S*). A subset of all tests $T_j \subseteq T_{All}$ is a *minimal test set* in accordance to $M_X$ if evaluation of mutation results of tests from this set gives the maximum mutation score $MS_{Xmax} = MS\ (M_X, T_j)$. Moreover, this test set includes the minimal number of tests, i.e. none of its tests could be omitted. In further steps of the process, we investigate if such minimal test sets are able to kill mutants from the whole mutant set $M_{All}$.

In general, many different minimal test sets for $M_X$ can exist giving the same mutation score. All minimal test sets can be effectively generated using the prime implicant of a monotonous Booolean function [11].

### D. Process Steps

After a mutant execution matrix has been evaluated, results for different sampling criteria and different sampling

degree $R$ are calculated. The following steps are executed for a given pair of parameters (*criterion, R*):

C1) Based on a given sampling criterion and a selected sampling degree $R$, a subset of all mutants is determined: $M_{C1} \subseteq M_{All}$. This subset includes all mutants (if $R$=100%) or a proper subset (for a lower sampling degree).

Then, we can calculate the mutation score that would be obtained running mutants from this subset against all tests from the considered test pool: $MS_{C1max}$= $MS$ ($M_{C1}$, $T_{All}$). This mutation score will be called the *maximum mutation score* for the set $M_{C1}$.

C2) According to the maximum mutation score for the set $M_{C1}$ we create a collection $L$ that includes minimal subsets of tests sufficient to obtain $MS_{C1max}$. The collection contains all minimal test sets determined by $M_{C1}$ or a limited number of such tests. A maximal cardinality of the collection - *TestSetLimit* is an experiment parameter.

C3) Mutation scores are calculated for each minimal test set comprised in collection $L$ and the set of all mutants $M_{All}$: $MS_{C3j}$= $MS(M_{All},T_j)$, where $T_j \in L$, $j=1..|L|$.

C4) An average mutation score is determined taking into account mutation results of all components of $L$ calculated in the previous step. We also compute an average number of tests over all minimal test sets included in $L$.

D) The steps C1)-C4) are repeated many times with the same sampling parameters in order to get different random statistics. Using average values obtained in consecutive steps C4), the final average mutation score $MS_{avg}$ and the average test number $NT_{avg}$ are calculated over the number of sampling repetition runs.

Finally, the whole process is recalculated for other values of sampling parameter $R$ and other sampling criteria.

All average values mentioned in the process description are calculated as an arithmetic average.

It should be noted that the process described in this section requires generating and running all mutants against all tests from a given test suite. However, the process is for research purposes. In a practical mutant sampling, only a subset of mutants is run against tests. Furthermore, not all mutants have to be generated. It is possible to generate a randomly selected subset of mutants according to a given sampling criterion. Moreover, this facility can be easily incorporated into existing mutation tools.

### *E. Metric-Based Quality Evaluation*

Comparison of different approaches to cost reduction of mutation testing should take into account a tradeoff between benefits and possible shortcoming of a method. Benefits can relate to a lower number of mutants that have to be generated and run in tests. Another advantage could be a reduced number of tests used in test runs of mutants. However, application of cost reduction methods can cause decline of mutation score adequacy in comparison to the one obtained using all mutants and more tests. Therefore, we proposed a quality metric [4] that can be adjusted for balancing these factors in study on cost reduction.

The metric depends on three components (Eq. 1). Each component is a normalized variable multiplied by a weight coefficient. The whole metric is a normalized sum of the components. Assuming a given sampling criterion, values of variables and the whole metric are normalized over their data set calculated for all values of a sampling parameter $R$.

$$EQ(W_{MS},W_T,W_M) = I(W_{MS} * I(S_{MS}) + W_T * I(Z_T) + W_M * I(Z_M)) \quad (1)$$

The weight coefficients $W_{MS}$, $W_T$, $W_M$ determine an impact of particular variables into the quality measure. The sum of coefficients must be equal to 1. A normalization function is denoted by $I()$. Three variables approximate the following measures:

- $S_{MS}$ - a loss of mutation score adequacy in an experiment,
- $Z_T$ - a cost decrease due to a reduced number of tests required for killing mutants in an experiment,
- $Z_M$ - a cost decrease due to a reduced number of mutants considered in an experiment.

The variables in mutant sampling experiments were calculated according to the following formulae (Eq. 2).

$$S_{MS} = MS_{avg} / MS_{orig} \quad (2)$$

$$Z_T = \begin{cases} 1 - (NT_{avg} / |T_{All}|) & \text{if } NT_{avg} > 0 \\ 0 & \text{otherwise} \end{cases}$$

$$Z_M = \begin{cases} 1 - (|M_{C1}| / |M_{All}|) & \text{if } |M_{C1}| > 0 \\ 0 & \text{otherwise} \end{cases}$$

Where symbols $MS_{avg}$, $MS_{orig}$, $NT_{avg}$, $T_{All}$, $M_{C1}$ and $M_{All}$ have the same meaning as in the process description.

While examining quality results with respect to different sampling criteria and different sampling degree $R$, we are looking for a "good randomization mode". The idea behind this notion is selection of promising sampling criteria and values of $R$ towards generalization of results. For a given sampling criterion, we can analyze the quality metric as a function of a parameter $R$ and observe maxima of the function. A good randomization mode should meet two following requirements:

*Unambiguous maximum* - we would like to avoid two situations: first - when increase in the number of randomly selected mutants (increase in $R$) gives the quality measure of the same high value (close to 1), and second - when there are several local maxima. The first situation would imply that taking more mutants we do not benefit in the mutation testing process. The second case corresponds to an ambiguous situation, where a quality measure does not monotonously depend on a sampling degree.

*Repeatability* - the maximum should be independent of a program. It means that quality metric *EQ* calculated for a given sampling criterion should reach its maximum within the similar range of parameter *R* for each project.

## III. Results and Discussion

In this section we describe the experimental set-up and discuss outcomes of our mutant sampling experiments.

### A. Experimental Set-up

Experiments were conducted on the following open-source C# programs corresponding to different software engineering tools:

1. Enterprise Logging.
2. Castle (modules Castle.Core, Castle.DynamicProxy2, CastleMicroCernel and Castle.Windsor).
3. Mono Gendarme.

The programs were companioned with unit tests. The tests were partially originated from the source projects and partially developed in order to improve code coverage. The basic complexity measures of the programs, the number of code lines and the number of classes and interfaces, as well as obtained coverage results are summarized in Table I.

The experiments were conducted using the CREAM tool (CREAtor of Mutants) devoted to mutation testing of C# programs [12],[13]. Apart from the support of the typical mutation testing process, the third version of the tool facilities experiments on cost reduction methods [4]. The tool is extended with a wizard that assists in performing experiments on mutation operator selection, mutant sampling and mutation clustering. Having created all mutants and performed all test runs, the mutation results are evaluated according to a given investigation process. Then the quality metrics are calculated and analyzed.

The experiments have been performed and their results evaluated under to the following assumptions:

- *Mutation operators* - in experiments first-order mutants were created with use of all object-oriented (18) and all standard (8) mutation operators implemented in CREAM v.3, including all standard mutation operators proposed to be selective [6].

Table I.
Program metrics

| No | LOC | | Classes & Interfaces | | Line coverage [%] |
|---|---|---|---|---|---|
| | with tests | without tests | with tests | without tests | |
| 1 | 87552 | 57885 | 991 | 587 | 82 |
| 2 | 54496 | 41288 | 724 | 493 | 77 |
| 3 | 51228 | 25692 | 907 | 171 | 87 |
| Sum | 193276 | 124865 | 2622 | 1251 | |

- *Covered mutants* - only mutants covered by tests were taken into account in evaluation of the mutation score. CREAM has an option to generate only covered mutants if required. We checked that none of uncovered mutant of these programs was killed by any tests from $T_{All}$.
- *Independent analysis for mutation operator categories*- evaluation of experiment results was performed independently for object-oriented and standard mutation operators. In the OO analysis, the set $M_{All}$ corresponds to all mutants of a given program generated with all OO operators. In the latter case, all standard mutation operators are considered.
- *Sampling criteria* - experiment results were evaluated independently for six sampling criteria (Sec. II.A).
- *Sampling parameter* - for every sampling criterion, parameter *R* was equal to 5%, 10%, 15%, …, 100% in consecutive experiments.
- *TestSetLimit* - the number of minimal test sets considered in each collection *L* was bounded by 15 sets (see step C2 in Sec. II.B).
- *Sampling repetition number* - for a given program, a selected sampling criterion and a given sampling parameter *R,* each sampling was repeated 10 times (compare point D in Sec. II.B).
- *Quality metric coefficients* - quality metric was calculated with weight coefficients $W_{MS,}$ $W_T$, $W_M$ equal to 0.6, 0.2, 0.2, if not stated elsewhere. These values are interpreted in the following way: mutation score accuracy amounts to 60% in the quality metric whereas the number of mutants and the number of tests amounts per 20% each (Sec. II.C).
- *Normalization* - metric variables $S_{MS}$ , $Z_T$ , $Z_M$, and the whole quality metric were normalized over the data set calculated for each sampling parameter value, i.e. 5%, 10%, 15%, …100%.

During a preliminary step, all mutants were generated and run with all tests. The basic outcome of the mutation testing of the subjects is given in Table II. Mutants that were not killed might be equivalent, i.e. not to be killable by any tests, although CREAM tries to prevent from generating equivalent mutants. After manual examination some mutants were determined being equivalent. The last column shows the original mutation score $MS_{orig}$ (i.e. covered mutants not recognized as equivalent divided by killed mutants). Those values were used as a reference in evaluation of mutant sampling.

### A. Evaluation of Mutant Sampling Results

Evaluation of results stored in the mutant execution matrix was performed according to the investigation process presented in Sec. II.B. Experiment results are given in three tables (Table III, Table IV, Table V) for each considered project, accordingly. They present average mutation scores obtained for different sampling criteria and different values of the parameter. A mutation score was computed as

TABLE II.
MUTATION RESULTS

| No, operator type | Generated covered mutants | Killed mutants | Equivalent mutants | Mutation score ($MS_{orig}$) [%] |
|---|---|---|---|---|
| 1 OO | 1341 | 558 | 438 | 61.8 |
| 1 Stand. | 1683 | 1151 | 60 | 70.9 |
| 2 OO | 1208 | 701 | 143 | 65.8 |
| 2 Stand. | 2379 | 1611 | 60 | 69.6 |
| 3 OO | 998 | 478 | 143 | 55.9 |
| 2 Stand | 4153 | 3009 | 79 | 73.9 |

percentage of killed mutants versus all generated mutants $M_{All}$. Mutants were killed using minimal sets of test cases determined for randomly selected subset of mutants. The mutation scores given in the tables are average values calculated over all random runs $M_{Savg}$ (p. D in Sec. II.B). Due to brevity reasons, only results for selected $R$ values are shown in the tables.

Analyzing the mutation results in dependence of the random sampling degree $R$, we can observe that even a small decline in number of mutants ($R$=95%) resulted in the lowering of the mutation score. However, when numbers of selected mutants are considerably high, deviation of the mutation score from the original value is quite small.

We calculated quality metric $EQ$, which took into account not only the mutation score but also two remaining quality factors (number of mutants and number of tests). In general, values of the quality metric are small for the low number of selected mutants (low $R$) because the mutation score is inaccurate. On the other hand, the quality is also not maximal (lower than 0.99) for the highest $R$, as in this case the number of mutants is the biggest. The tradeoff between the quality factors is represented by the maxima of the quality results.

Quality metric flow in dependence of increase in the sampling parameter $R$ is presented in the Appendix. The results of object-oriented mutation operators are shown in Fig.1-Fig. 6, and of standard operators in Fig. 7 - Fig. 12. For each kind of mutation operators, six diagrams are shown, which correspond to different sampling criteria. Three lines in any diagram represent different subject programs. In respect to the sampling parameter, the diagrams cover subsets of results, i.e. parameter $R$ varies from 20% to 75% for OO operators, and from 15% to 60% in case of standard operators. The selected scopes of the parameter give a chance to observe maxima of the quality metric and consequently interpret the results.

Based on the idea of "a good randomization mode" introduced in Sec. II.C, we specify its requirements in a quantitative way:

*Unambiguous maximum* - we discard situations when $EQ$ is of the same high value (>0.9975) for the increase in $R$ or there are several local maxima for $EQ$ above 0.99.

*Repeatability* - Maximum of quality metric EQ (equal 1) calculated for a given sampling criterion should be similar for each project, i.e. the appropriate value of parameter $R$ should be the same or differ only ±5% of mutants.

Taking into account the above requirements, we analyzed the results independently for the object-oriented and standard mutation operators.

For OO operators, selection of mutants in a *fully random* way (1) or according to *namespace* (6) does not meet both requirements. The first requirement is also not fulfilled for the *file random* (2) and mutation *operator random* (5) criteria. Only the remaining two criteria, *class random* and

TABLE III.
AVERAGE MUTATION RESULTS (MS IN [%]) OF MUTANT SAMPLING FOR ENTERPRISE LOGGING

| R [%] | (1) Fully random | | (2) File random | | (3) Class random | | (4) Method random | | (5) Operator random | | (6) Namespace random | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | OO | St | OO | St | OO | St | OO | St | OO | St | OO | St |
| 5 | 27.7 | 50.5 | 22.4 | 47.1 | 20.1 | 46.4 | 18.8 | 41.4 | 27.2 | 49.3 | 28.6 | 49.5 |
| 10 | 37.8 | 57.9 | 34.7 | 54.4 | 34.3 | 54.2 | 29.7 | 51.0 | 36.0 | 56.7 | 35.6 | 56.2 |
| 20 | 44.7 | 63.2 | 44.0 | 61.3 | 44.0 | 61.5 | 41.2 | 60.7 | 45.6 | 62.8 | 46.7 | 63.0 |
| 30 | 51.2 | 65.6 | 49.0 | 64.4 | 49.2 | 64.8 | 46.9 | 64.2 | 50.0 | 65.9 | 50.6 | 65.4 |
| 40 | 53.7 | 67.2 | 52.8 | 67.3 | 53.1 | 66.4 | 50.3 | 66.0 | 53.8 | 67.5 | 54.0 | 66.9 |
| 50 | 55.8 | 68.4 | 55.3 | 68.1 | 55.9 | 68.5 | 54.2 | 67.5 | 56.4 | 68.4 | 56.3 | 68.4 |
| 60 | 57.7 | 69.3 | 57.0 | 68.7 | 56.8 | 68.7 | 55.4 | 68.2 | 57.6 | 69.2 | 57.7 | 69.0 |
| 70 | 59.1 | 70.0 | 57.9 | 69.3 | 57.9 | 68.9 | 56.2 | 68.4 | 58.9 | 69.8 | 59.1 | 70.0 |
| 80 | 60.1 | 70.3 | 58.9 | 69.6 | 58.5 | 69.3 | 56.9 | 68.9 | 60.0 | 70.2 | 59.8 | 70.2 |
| 90 | 60.9 | 70.6 | 59.4 | 69.8 | 59.2 | 69.5 | 57.4 | 69.1 | 60.9 | 70.6 | 60.7 | 70.5 |
| 95 | 61.5 | 70.8 | 59.6 | 70.1 | 59.2 | 69.7 | 57.4 | 69.2 | 61.3 | 70.8 | 61.2 | 70.6 |
| 100 | 61.8 | 70.9 | 61.8 | 70.9 | 61.8 | 70.9 | 61.8 | 70.9 | 61.8 | 70.9 | 61.8 | 70.9 |

TABLE IV.
AVERAGE MUTATION RESULTS (MS IN [%]) OF MUTANT SAMPLING FOR CASTLE

| R [%] | (1) Fully random | | (2) File random | | (3) Class random | | (4) Method random | | (5) Operator random | | (6) Namespace random | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | OO | St | OO | St | OO | St | OO | St | OO | St | OO | St |
| 5 | 34.6 | 51.7 | 24.1 | 48.7 | 22.8 | 47.4 | 25.5 | 38.1 | 33.7 | 52.4 | 32.4 | 51.1 |
| 10 | 41.5 | 58.9 | 36.9 | 57.7 | 35.1 | 57.5 | 33.2 | 52.5 | 41.0 | 58.2 | 41.0 | 58.9 |
| 20 | 51.1 | 63.6 | 48.1 | 62.8 | 47.6 | 62.9 | 44.0 | 61.4 | 49.1 | 64.0 | 50.3 | 63.8 |
| 30 | 55.2 | 65.9 | 51.7 | 65.2 | 51.0 | 65.2 | 49.0 | 63.8 | 54.0 | 65.7 | 53.8 | 65.8 |
| 40 | 57.8 | 67.0 | 57.9 | 66.6 | 57.5 | 66.8 | 54.0 | 65.9 | 57.1 | 67.0 | 58.1 | 66.8 |
| 50 | 59.9 | 67.7 | 60.3 | 67.6 | 60.2 | 67.6 | 58.9 | 67.2 | 59.8 | 67.6 | 60.0 | 67.8 |
| 60 | 61.7 | 68.3 | 61.4 | 68.2 | 61.3 | 68.0 | 59.9 | 67.6 | 61.6 | 68.2 | 61.6 | 68.3 |
| 70 | 63.4 | 68.7 | 62.9 | 68.4 | 62.7 | 68.5 | 61.0 | 67.9 | 62.7 | 68.7 | 63.1 | 68.8 |
| 80 | 64.4 | 69.1 | 63.8 | 68.9 | 63.8 | 68.9 | 61.8 | 68.2 | 63.9 | 69.1 | 64.4 | 69.1 |
| 90 | 65.2 | 69.4 | 64.3 | 69.1 | 64.5 | 69.2 | 62.3 | 68.5 | 65.0 | 69.3 | 65.1 | 69.3 |
| 95 | 65.5 | 69.4 | 64.9 | 69.3 | 64.7 | 69.3 | 62.5 | 68.4 | 65.3 | 69.4 | 65.4 | 69.4 |
| 100 | 65.8 | 69.6 | 65.8 | 69.6 | 65.8 | 69.6 | 65.8 | 69.6 | 65.8 | 69.6 | 65.8 | 69.6 |

*method random* (3,4), meet both requirements of the "good" mode. Comparing these two criteria we have found that the *class random* criterion gave better results. For all projects, its quality value was maximal for the same lower sampling degree *R*=40%. In case of *method random* the maximal *EQ* were calculated for higher number of mutants: *R*=50-55% for different projects.

It appears that using mutant sampling as a cost reduction method of OO mutation testing, we should select 40% of mutants that could be generated for each class.

Examining the results for standard mutation (Fig. 7 - Fig. 12) we can observe that sampling criteria of *fully random* and *namespace random* do not meet both "good sampling" requirements, similarly as for OO operators. In addition, both criteria are also not fulfilled by the *method random* criterion. In case of *class random* the first requirement is not met.

Two criteria, namely *file random* and *mutation operator random*, gave results consistent with the requirements. However, the maximum of the quality metric was in the range of 35-40% selected mutants for the *file random* criterion, whereas about 30-35% for the *mutation operator random*. The second case required less mutants, therefore, the most beneficial results for standard operators could be obtained while sampling mutants according to *mutation operator* criterion with the sampling degree *R*=30-35%.

Reduced number of mutants and tests indicates at the lower complexity of mutation testing. In order to compare effective benefits we measured real times of mutant generation and test execution. In Table VI, we compare times of all mutants and times of sampling with parameter *R*=35% and *class random* criterion for OO mutation or *R*=30% and *operator random* in case of standard mutation operators, accordingly. Significant reduction in these times can be observed.

With respect to the average results for all investigated programs, it appears that sampling about 40% of mutants for each class for OO operators took 32% of time to generate the mutants. Mutation score was declined in 15% in reference to all mutants and all tests (85% of $MS_{orig}$). It is possible to use only about 10% of tests to obtain this mutation score.

Mutant sampling gives better results for standard mutation operators than for OO. While sampling of 30% of mutants for each operator, the mutation score was equal to 93% of the original one. Mutant generation time declined in 70%. It would be possible to use only 15% of tests to obtain this result.

### *B. Threats to validity*

The experiments were conducted on widely used, complex open-source programs, with 3-5 thousands of mutants per each. However, the conclusion validity can be limited by the small number of subjects. Moreover, only programs in C# were mutated. No detailed results are given for other OO languages, as Java or C++, although we could expect similar trends due to analogy in mutation operators.

The original tests associated with programs had insufficient code coverage; therefore, additional tests were developed. The code coverage did not reach 100% even with all tests. In experiments, only mutants covered by tests were taken into account. The calculation of MS can also be influenced by equivalent mutants, although the most of them was identified before the result evaluation.

The presented results depend on the coefficients $W_{MS}$, $W_T$, $W_M$ of the quality metric. Therefore, the experiment

TABLE V.
AVERAGE MUTATION RESULTS (MS IN [%]) OF MUTANT SAMPLING FOR MONO GENDARME

| R [%] | (1) Fully random | | (2) File random | | (3) Class random | | (4) Method random | | (5) Operator random | | (6) Namespace random | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | OO | St | OO | St | OO | St | OO | St | OO | St | OO | St |
| 5 | 20.1 | 48.2 | 16.0 | 45.7 | 15.3 | 45.4 | 15.0 | 39.6 | 20.8 | 47.6 | 21.0 | 48.3 |
| 10 | 31.5 | 57.9 | 27.0 | 56.9 | 25.9 | 57.4 | 21.7 | 54.3 | 29.3 | 57.1 | 29.5 | 58.2 |
| 20 | 39.2 | 65.5 | 38.2 | 64.9 | 38.6 | 65.6 | 30.1 | 64.2 | 38.4 | 65.4 | 38.5 | 65.2 |
| 30 | 44.0 | 68.6 | 40.9 | 68.2 | 42.0 | 68.1 | 34.2 | 68.2 | 43.1 | 68.7 | 43.6 | 68.8 |
| 40 | 46.8 | 70.3 | 45.7 | 70.3 | 45.7 | 70.2 | 42.6 | 70.1 | 46.6 | 70.2 | 47.2 | 70.3 |
| 50 | 49.0 | 71.4 | 49.1 | 71.6 | 48.8 | 71.7 | 47.1 | 71.3 | 48.3 | 71.5 | 49.2 | 71.3 |
| 60 | 50.9 | 72.2 | 50.4 | 72.3 | 49.8 | 72.4 | 48.3 | 72.2 | 51.1 | 72.3 | 50.6 | 72.2 |
| 70 | 52.4 | 72.8 | 51.3 | 72.8 | 51.4 | 72.8 | 49.9 | 72.6 | 52.4 | 72.8 | 52.3 | 72.9 |
| 80 | 53.9 | 73.2 | 52.2 | 73.2 | 51.7 | 73.2 | 51.2 | 73.0 | 53.7 | 73.3 | 53.8 | 73.2 |
| 90 | 54.9 | 73.6 | 53.1 | 73.5 | 52.9 | 73.5 | 51.6 | 73.3 | 54.8 | 73.6 | 54.6 | 73.6 |
| 95 | 55.4 | 73.7 | 52.8 | 73.6 | 52.9 | 73.6 | 52.0 | 73.4 | 55.4 | 73.7 | 55.4 | 73.7 |
| 100 | 55.9 | 73.9 | 55.9 | 73.9 | 55.9 | 73.9 | 55.9 | 73.9 | 55.9 | 73.9 | 55.9 | 73.9 |

outcomes were recalculated for another set of weight coefficients. According to a new set (0.8, 0.1, 0.1), mutation score is a more dominant factor in the metric in comparison to the case discussed above. We obtained results that have corresponded to this interpretation. The quality measures were the best for the same sampling criteria as chosen above but for the higher sampling degree. The percent of sampled mutants was equal to 90-100% for OO operators and 60-70% for standard ones. For these coefficients benefits of lower number of mutants or tests are very small, especially for object-oriented operators.

Another factor that influenced the construct validity was the sampling parameter ($R$). The experiments covered the whole scope of the parameter value (from 5% to 100%) with a small difference (per 5%). All calculations were also repeated ten times for different random sampling.

## IV. RELATED WORK

There are different methods to reduce a cost of mutation testing. Many of them focus on reduction of mutant number, including mutant sampling [1][3].

Experimental evaluation on mutant sampling with 22 standard mutation operators in Mothra resulted in mutation score drop in 16% assuming 10% of mutants were fully randomly sampled [5]. Our results were different, as in the quality metric we took into account not only a drop in the mutation score but also efficiency factors. However, if we compare *MS* only, the results for standard operators applied for C# programs are for the first random criterion very similar, i.e. $R$=10% gives 15% decline of a mutation score. With the same sampling degree but for OO operators MS decrease is substantially bigger - about 37%.

Other experiments have compared mutant sampling approaches to selective mutation of standard operators applied in C programs. Empirical results reported by [14] point at the preference of selective mutation over the fully random one. The opposite is claimed in [15], in which two sampling modes were considered: fully random - called here one-round random, and two-round random (first a mutation operator is selected than a mutant within this operator). The results showed that random sampling methods can be as effective as those based on operator selection, but are more stable and predictable. The results of this comparison cannot be simply applied to OO operators. It is known that standard operators can generate much more mutants and many of them can be surplus, but there are less tests killing such mutants or the tests are not adequate to kill OO mutants [4], [16].

An approach that would be an alternative to selective mutation and mutant sampling was also discussed in [17], but it was only illustrated by simulation results. Moreover, assumptions behind the idea were more suitable to standard mutation operators than object-oriented.

Mutant sampling method was also beneficially applied in VHDL description [18]. The sampling criterion was similar to the mutation operator sampling, but the percentage of selected mutants was independently established for each operator.

Sun [19] explored mutant reduction based on a program structure and different strategies of path analysis. Experiments on C programs showed that the best strategies were more effective than the random selection technique preserving a sufficiently high mutation score. However, some other strategies did not outperform random approach.

All discussed above results were devoted to standard mutation operators.

Before the experiments with CREAM were conducted, to the best of our knowledge, no results of OO sampling were performed, and no cost reduction on mutation of C# programs was investigated. Experiments following the

similar process were developed for selective mutation and mutant clustering of C# programs [4], [7].

Experiments on mutant sampling on 8 Java classes were conducted by Bluemke [22]. Fully random sampling with the sampling degree ranged from 60% to 10% were examined. Randomly sampling 60% or 50% of mutants in Java programs gave significant reduction in the cost of testing with acceptable mutation score and code coverage decline. This result has been averaged on all kinds of mutation operators. No quality measures were considered.

Java program were also a target of experiments reported by Ma [23]. The weak mutation technique, in which intermediate program results are taken into account, was combined with mutant clustering, in with a mutant is selected among a group of mutants of similar behavior. Only selected mutants were completely executed to obtain the strong mutation results. The number of mutants was significantly reduced. However, the experiments were limited to simple programs and only several standard mutation operators. Hence, no data about object oriented mutation were given.

Object oriented mutation operators for C++ has been recently investigated in experiments reported by Delgado-Perez [24]. They considered also random selection of operators, but not mutant sampling.

Our study differs also from those of other authors in application of the quality metric that takes into account not only a drop in mutation score but also efficiency measures - numbers of mutants and numbers of tests. The metric applied in experiments was proposed in [4], and used also in other experiments reported in [7].

Other metrics to mutation testing quality were discussed by Ester-Botaro in [25]. Some of them were an extension of a effectiveness metric previously proposed by one of the authors. They discuss quality of mutant and operators in order to omit those of a low quality. However, these metrics do not evaluate a cost of a mutation testing process.

Another approach has been recently investigated in [26], where mutation adequacy score was estimated taking into account several object-oriented metrics, which capture the structural complexity of a program.

TABLE VI.
BENEFITS OF MUTANT GENERATION TIME AND TEST EXECUTION TIME FOR MUTANT SAMPLING

| R [%] | Time of mutant generation (including compilation) [h:min:sec] | | Time of test execution [h:min:sec] | |
|---|---|---|---|---|
| | All | Sampling | All | Sampling |
| 1 OO | 06:26:11 | 01:48:39 | 06:32:37 | 00:09:09 |
| 2OO | 05:37:44 | 01:49:31 | 07:14:14 | 00:31:16 |
| 3OO | 03:49:32 | 01:23:41 | 02:02:29 | 00:11:24 |
| 1 St | 07:22:44 | 02:12:04 | 11:45:39 | 00:20:15 |
| 2 St | 10:36:60 | 03:10:19 | 15:44:19 | 01:29:15 |
| 3 St | 13:53:39 | 04:09:13 | 09:43:36 | 13:53:39 |

## V. CONCLUSION

The empirical study presented in this paper confirms the tendency that OO mutation operators undergo different characteristics than standard operators and therefore may require slightly different methods of cost reduction.

Moreover, the benefits of the methods previously studied for standard operators are lower in case of OO ones, probably due to a lower number of generated and unnecessary mutants.

Using the sampling approach, we can achieve some lowering the number of mutants and tests but also obtaining a relative decrease in mutation score accuracy. For the selected tradeoff, the mutation score was about 93% of that obtained with all mutants and all tests using standard operators, and about 85% for object oriented ones.

Comparison of mutant sampling of C# programs with other "do fewer" methods, such as selective mutation [4] and mutant clustering [7], does not support one definite leading method. The number of mutants and tests was lower for mutant sampling than for selective mutation and similar to those of clustering. On the other hand, the mutation accuracy was lower than in those methods. However, all differences are about few percent and could also be treated as a measurements' deviation. Moreover, sampling methods are superior because of their stability and simple implementation. Mutant clustering is computationally expensive, whereas selective mutation, especially in respect to object-oriented operators, is not so decisive and can depend on a program [4], [16].

The lessons learned is that instead of fully random sampling we would recommend to use different sampling criteria: *class random* for object-oriented operators and *mutation operator random* for standard ones. Both criteria can be easily implemented and both were the best for different tunings of the impact factors in the quality metric.

The percentage of selected mutants depends on the preferred tradeoff between mutation score decline and the efficiency measures (number of mutants and number of tests). For the ratio 6:2:2 of these three components the suggested sampling degree is about 40% for object oriented operators and 30-35% for standard ones.

It should be noted, that in practice, the number of mutants could be not the most important cost factor. Overall time of mutation testing is also strongly influenced by the number of tests to be performed. Therefore, comparing a process quality we should take into account different factors, as in the quality metric applied in the paper.

Concerning C# programs, improvement in mutation testing efficiency is provided by code mutation at level of the Common Intermediate Language of .NET. Another tool

[27], which satisfies this requirement and is tidily coupled with the MS Visual Studio, gives promising results and can be further enriched with some cost reduction methods.

## APPENDIX: QUALITY METRIC IN DEPENDENCE ON THE SAMPLING PARAMETER R

Legend: "- - -" dashed line Enterprise Logging, "…."dotted line Castle, "___" solid line MonoGendarme.

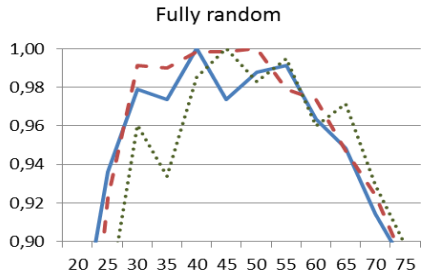


Fig. 1 OO mutation operators, fully random sampling

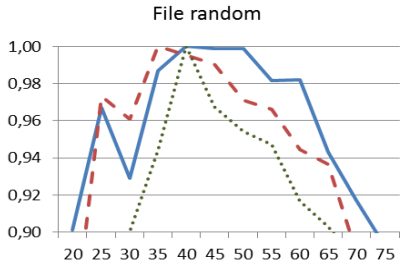


Fig. 2 OO mutation operators, file random sampling

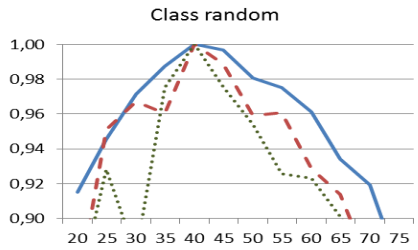


Fig. 3 OO mutation operators, class random sampling

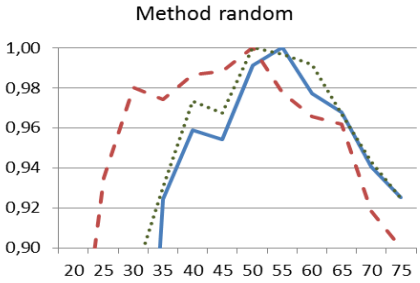


Fig. 4 OO mutation operators, method random sampling

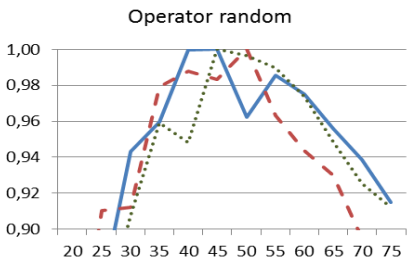


Fig. 5 OO mutation operators, operator random sampling

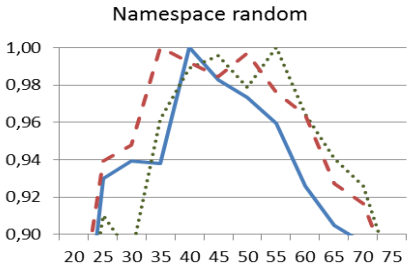


Fig. 6 OO mutation operators, namespace random sampling

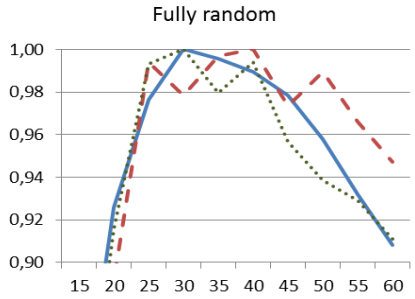


Fig. 7 Standard mutation operators, fully random sampling

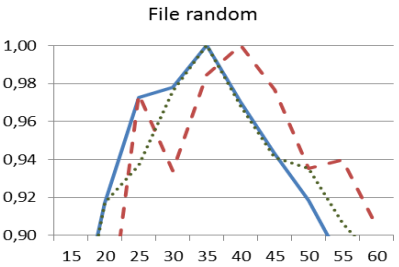


Fig. 8 Standard mutation operators, file random sampling

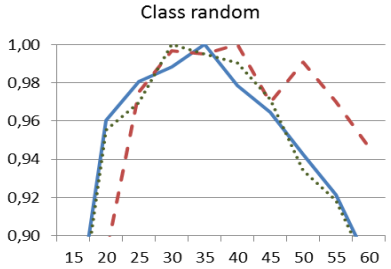


Fig. 9 Standard mutation operators, class random sampling

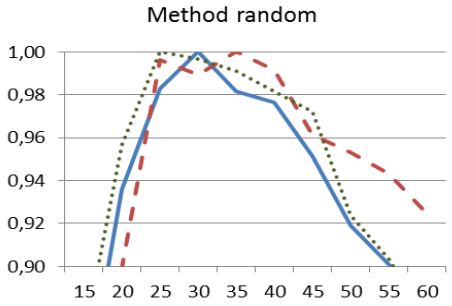


Fig. 10 Standard mutation operators, method random sampling

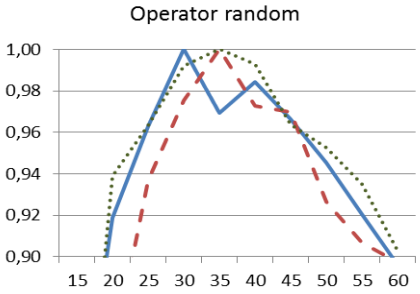


Fig. 11 Standard mutation operators, operator random sampling

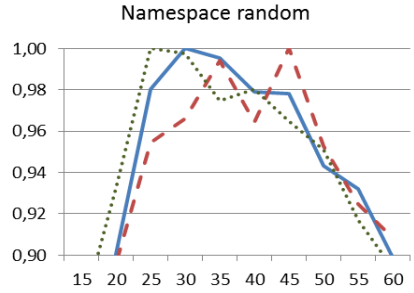


Fig. 12 Standard mutation operators, namespace random sampling

## REFERENCES


[1] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no.5, pp. 649—678, Sept-Oct. 2011, https://dx.doi.org/10.1109/TSE.2010.62

[2] A. Derezińska and A. Szustek, "Object-Oriented testing capabilities and performance evaluation of the C# mutation system," in *Proc. CEE-SET 2009*, Szmuc, T., Szpyrka, M., Zendulka, J. Eds., LNCS, vol. 7054, 2012, pp. 229–242, https://dx.doi.org/10.1007/978-3-642-28038-2_18

[3] M. P. Usaola and P. R. Mateo, "Mutation testing cost reduction techniques: a survey," *IEEE Software,* vol. 27, no. 3, pp. 80–86, May-June 2010, https://dx.doi.org/10.1109/MS.2010.79

[4] A. Derezińska and M. Rudnik, "Quality evaluation of Object-Oriented and standard mutation operators applied to C# programs," in *Proc. TOOLS Europe 2012*, C.A. Furia, S. Nanz Eds., LNCS, vol. 7304, Springer Berlin Heildelberg, 2012, pp. 42–57, https://dx.doi.org/10.1007/978-3-642-30561-0_5

[5] A. P. Mathur and W. E Wong, "Reducing the cost of mutation testing: an empirical study," *J. of Systems and Software*, vol. 31, no. 3, pp. 185–196, Dec. 1995, http://dx.doi.org/10.1016/0164-1212(94)00098-0

[6] J. Offut, G. Rothermel, and C.Zapf, "An experimental evaluation of selective mutation," in *Proc. 15th International Conference on Software Engineering*, IEEE Comp. Soc. Press, 1993, pp. 100–107, https://dx.doi.org/10.1109/ICSE.1993.346062

[7] A. Derezińska, "A quality estimation of mutation clustering in C# programs," in *New Results in Dependability and Computer Systems* W. Zamojski et al. Eds., AISC vol. 224, Springer, Switzerland, 2013, pp.183-194, https://dx.doi.org/10.1007/978-3-319-00945-2_11

[8] A. T. Acree, "On Mutation,", Ph.D. thesis, Georgia Institute of Technology, Atlanta, GA, 1980.

[9] T. A. Budd, "Mutation analysis of program test data," Ph.D. thesis, Yale University, New Haven, CT, 1980.

[10] G. J. Myers, *The Art of Software Testing*, John Wiley & Sons, 1979, 3rd. ed 2011

[11] M. Kryszkiewicz, "Fast algorithm finding minima in monotonic Boolean functions," Warsaw Univ. of Technology, ICS Res Rep. 42/93, 1993.

[12] A. Derezińska and A. Szustek, "Tool-supported mutation approach for verification of C# programs," in *Proc. International Conference on Dependability of Computer Systems,* W. Zamojski, et al. Eds., pp. 261–268, 2008, https://dx.doi.org/10.1109/DepCoS-RELCOMEX.2008.51

[13] CREAM, http://galera.ii.pw.edu.pl/~adr/CREAM/

[14] E. F. Barbosa, J.C. Maldonado, and A.M.R. Vincenzi, "Toward the determination of sufficient mutant operators for C," *Softw. Test. Verif. and Reliab*. vol. 11, pp. 113–136, June 2001, https://dx.doi.org/10.1002/stvr.226

[15] L Zhang, S-S., Hou, J-J. Hu, T., Xie, and H. Mei, "Is operator-based mutant selection superior to random mutant selection?" in *Proc. 32nd International Conference on Software Engineering, ICSE 2010*, 2010, pp. 435–444, https://dx.doi.org/10.1145/1806799.1806863

[16] J. Hu, N. Li, and J. Offutt, "An analysis of OO mutation operators," in *Proc. of 4th International Conference Software Testing Verification and Validation Workshops*, 6th Workshop on Mutation Analysis, IEEE Comp. Soc., 2011, pp. 334–341, https://dx.doi.org/10.1109/ICSTW.2011.47

[17] K. Adamopoulos, M. Harman, and R. M. Hierons, "How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution," GECCO'04, LNCS, vol. 3103, pp. 1338-1349. Springer, 2004, https://dx.doi.org/10.1007/978-3-540-24855-2_155

[18] M. Scholive, V. Beroulle, C. Robach, M. L. Flottes, and B. Rouzeyre, "Mutation sampling technique for the generation of structural test data," in *Proc. of the Conference on Design, Automation and Test in Europe*, DATE'05, vol. 2, pp.1022 – 1023. IEEE Comp. Soc., 2005.

[19] C. Sun, F. Xue, H. Liu, and X. Zhang, "A path-aware approach to mutant reduction in mutation testing," *Information and Software Technology*, vol. 81, pp. 65-81, Jun. 2017, https://dx.doi.org/10.1016/j.infsof.2016.02.006

[20] S. Segura, R. M. Hierons, D. Benavides, and A. Ruiz-Cortes, "Mutation testing on an object-oriented framework: An experience report," *Information and Software Technology*, 53(10), pp. 1124—1136, Oct. 2011, https://dx.doi.org/10.1016/j.infsof.2011.03.006

[21] L. Zhang, M. Gligoric, D. Marinov, and S. Khurshid, "Operator-based and random mutant selection: better together," in *28th IEEE/ACM Conference on Automated Software Engineering,* Palo Alto, USA, 2013, pp. 92-102, https://dx.doi.org/10.1109/ASE.2013.6693070

[22] I. Bluemke and K. Kulesza, "Reduction of computational cost in mutation testing by sampling mutants," in *New Results in Dependability and Computer System,* W. Zamojski et al. Eds., Springer, 2013, pp. 41-51, https://dx.doi.org/10.1007/978-3-319-07013-1_9

[23] Y.-S. Ma and S.-W. Kim, „Mutation testing cost reduction by clustering overlapped mutants," *J. of Systems and Software*, vol. 115, pp. 18-30, May 2016, http://dx.doi.org/10.1016/j.jss.2016.01.007

[24] P. Delgado-Perez, S. Segura, and S. Media-Bulo, „Assessment of C++ object-oriented mutation operators: A selective mutation approach," *Softw TestVerif Reliab*., 2017, https://dx.doi.org/10.1002/stvr.1630

[25] A. Estero-Botaro, F. Palomo-Lozano, I. Medina-Bulo, J. J. Dominguez-Jimenez, and A. Garcia-Dominguez, "Quality metrics for mutation testing with application to WS-BPL compositions," *Softw Test Verif Reliab,* vol. 25, no. 5-7, pp. 536-571, Aug.-Nov. 2015, https://dx.doi.org/10.1002/stvr.1528

[26] M. Moghadam and S. Babamir, "Mutation score evaluation in terms of object-oriented metrics," *4th International eConference on Computer and Knowledge Engineering (ICCKE*), 2014, Mashhad, Iran 2014, pp. 775–780, https://dx.doi.org/10.1109/ICCKE.2014.6993419

[27] A. Derezińska and P. Trzpil, "Mutation testing process combined with Test-Driven Development in .NET Environment," in *Theory and Engineering of Complex Systems and Dependability*, W. Zamojski et al. Eds., AISC vol. 365, Springer, pp. 131-140, 2015, https://dx.doi.org/10.1007/978-3-319-19216-1_13