A Quality Estimation of Mutation Clustering in C# Programs

Anna Derezińska

Institute of Computer Science, Warsaw University of Technology
Nowowiejska 15/19, 00-665 Warsaw, Poland
A.Derezinska@ii.pw.edu.pl

Abstract. Mutation testing tasks are expensive in time and resources. Different cost reduction methods were developed to cope with this problem. In this chapter experimental evaluation of mutation clustering is presented. The approach was applied for object-oriented and standard mutation testing of C# programs. The quality metric was used to compare different solutions. It calculates a tradeoff between mutations score accuracy and mutation costs in terms of number of mutants and number of tests. The results show a substantive decrease in number of mutants and tests while suffering a small decline of mutation score accuracy. However the outcome is not superior to other cost reduction methods, as selective mutation or mutant sampling.

1 Introduction

In mutation testing many faulty versions (so-called *mutants*) of a program under test are generated by application of mutation operators. If any test from a given test set detects an abnormal behavior of a mutant, the mutant is set to be *killed*. The ability of a test set to reveal faults specified by mutation operators is named *mutation score* (*MS*) and measured as a ratio of the number of killed mutants over the number of all non-equivalent mutants. An *equivalent* mutant generates the same outcomes as the original program and cannot be killed by any test. Mutation operators define various kinds of faults. Standard operators deal with expressions and structural features common to all general purpose programming languages, whereas object-oriented (OO) operators with flaws specific to OO languages [1].

An important obstacle of mutation testing approach is the computational expense. For one program many mutants can be generated and one mutant is run with many test cases. Several cost reduction techniques to mutation testing were proposed based on “do smarter”, “do faster” or “do fewer” approaches [1,2]. The mutation clustering belongs to a “do fewer” method that tries to execute fewer mutants against fewer test cases.

In mutation clustering a set of mutants is divided into disjoint subsets, so-called groups, based on the ability of tests to kill these mutants. Various groups of mutants can be killed by the same, or similar, subsets of tests. In the further testing

W. Zamojski et al. (Eds.): *New Results in Dependability & Comput. Syst.*, AISC 224, pp. 119–129.
DOI: 10.1007/978-3-319-00945-2_11

process, only one mutant representing a group is applied instead of all mutants from the group. Division of mutants can be realized using one of the clustering algorithms, such as agglomerative hierarchical or K-means clustering [3, 4] or by static domain analysis [5].

The analysis on mutation clustering was performed on C programs using standard mutation operators [3], and on an exemplary, simple Java program [5]. It was not shown how the approach will scale up for bigger-size and practically used programs. Another open question was whether the clustering method can give benefits to the object-oriented mutation testing, as according to the author's experience [6,7] the object-oriented mutation evaluates with different characteristics than mutation with standard operators. Mutation testing of C# programs is supported by the CREAM tool [6,8,9]. In order to effectively perform experiments on various cost reduction methods, an extension to CREAM was implemented [7,10].

The aim of the research presented in this chapter is examination whether the clustering method is worthwhile in mutation testing of C# programs, both in terms of standard and object-oriented operators. The tradeoff between the quality of mutation testing result (*MS*) and its cost (number of mutants and tests) is quantitatively evaluated with assessment of an original, tool supported metric [7].

The reminder of this chapter is organized as follows. Next Section describes related work. In Section 3 the main methodological issues are discussed. The experimental set-up is presented in Section 4. Section 5 gives experiment results and their analysis. Finally, Section 6 concludes the work.

2 Related Work

Promising results of mutation clustering with standard operators and C programs were presented in [3]. They showed, for example, that using a substantially reduced number of mutants (13%) and of tests (8%) we can obtain almost the same mutation score, i.e. 99%. In experiments with C# programs so good results were obtained neither with standard nor with OO operators.

Experiments in [3] focus on the assessment of potential clustering benefits, similarly as in this chapter. Therefore clustering was based on results of runs of all mutants against all test cases. Ji at al. proposed a practical approach in which clustering is based on static domain analysis [5]. The experiment proved that the method is applicable and dealt with a small Java program giving the encouraging results (e.g. 25% mutants with 62% tests gave 94% of mutation score).

The mostly studied "do fewer" method was selective mutation [11-14], in which only subset of mutation operators is used. Five standard operators were recognized as selective in experiments with Fortran programs [11]. The research on OO mutation was neither so promising nor so conclusive [7,14]. A method of mutant sampling, based on random selection of mutants, gave good results (10% of mutants with 16% loss of *MS* accuracy) for standard mutation in Fortran [15]. Sampling according to different criteria was studied in [10] for C# programs.

There are several tools for mutation testing of Java programs, but the only tools that support mutation testing of C# programs with some standard and OO operators are those implemented under the author's supervision: CREAM [6,8,9] and the prototype ILMutator [16]. The latter injects faults directly into Intermediate Language of .NET and therefore speeds up mutant generation. However the cost of test execution remains the same as using CREAM.

3 Methodology

In experiments discussed in this chapter the agglomerative clustering algorithm was applied [4]. Below its general idea is presented. Next, the experiment scenario and quality metric are discussed.

3.1 Clustering Algorithm

A clustering algorithm returns the division of mutants for a given set of mutants M and a set of test T that kill the mutants. The algorithm is characterized by a threshold parameter K reflecting a similarity of mutant groups. Two groups are similar with K degree, if the number of tests that kill at least one mutant from one group and kill none mutant from the second group is equal to K.

The general idea of the algorithm can be described in following steps:

1) First, for each mutant from the set M, a mutant group is initialized. Therefore there are $|M|$ groups of one element, where $|X|$ denotes cardinality of set X.
2) A temporary group similarity value is set to 0 ($i = 0$).
3) All pairs of current groups are compared. Two groups are merged if the similarity of the groups is less then the temporary group similarity value (i). If there are no more pairs of groups to be merged we go to the next step.
4) The algorithm stops if the current group similarity value reaches the algorithm threshold ($i = K$). Otherwise, the temporary similarity value is incremented (i++) and the algorithm is continued with the step 3.

For example, given a set of mutants $M = \{m1, m2, m3, m4\}$ killed by the test sets $\{t1,t2\}$, $\{t1,t2\}$, $\{t1\}$, $\{t2,t3\}$ accordingly, and the parameter $K=1$, we obtain the following two groups of mutants $\{m1, m2, m3\}$ and $\{m4\}$. The first group is killed by the test set $\{t1, t2\}$, whereas the second group is killed by tests $\{t2, t3\}$.

3.2 Experimental Scenario on Mutation Clustering

In experiments on cost reduction methods we answer a question how a reduced number of mutants is able to assess the test quality (MS) in comparison to all mutants that could be generated. Moreover we look for minimal test sets that could be as effective in revealing faults as the reference test set. The experiments on mutation clustering were designed according to the following scenario:

A) Using a given set of mutation operators, all first order mutants of a program under test are generated (mutant set M_{All}).
B) All mutants from M_{All} are run against all tests from a considered set T_{All}. The resulting mutant execution matrix states for each pair <mutant m, test t> whether the mutant m is killed by the test t or not.
C) A parameter K of the clustering algorithm is selected. Disjoint mutant groups are determined by the clustering algorithm for given mutants M_{All}, test set T_{All}, and parameter K.
C1) A subset of mutants $M_{C1} \subseteq M_{All}$ is created by selection of one representative mutant from each mutant group. Mutation score MS_{C1max}= MS (M_{C1}, T_{All}) is calculated assuming that mutants from this subset were tested by all tests.
C2) In order to optimize a test set, a collection L of test subsets of T_{All} is created. Any test set in L has a minimal number of tests and gives the mutation score equal to MS_{C1max} (from step C1). Tests sets meeting those requirements can be generated using prime implicant of a monotonous Boolean function [17]. The collection L includes either all test sets of this kind, or a limited number *TestSetLimit* of such sets. The value of *TestSetLimit* is a parameter of an experiment.
C3) For any test set included in L a mutation score is calculated as if all mutants from M_{All} were tested by the test set.
C4) The average mutation score MS_{avg} is calculated from the results of step C3.

Investigating an impact of the clustering threshold on the mutation results, we can repeat steps C(C1-C4) for different values of K. Next the final statistics and quality metrics are calculated.

3.3 Quality Metric

The primary metric used for evaluating results on mutation testing process is the mutation score (MS). The original mutation score MS_{orig}= MS (M_{All}, T_{All}) is calculated using execution results of all mutants from set M_{All} and all tests from set T_{All}.. If a reduced number of mutants ($M_i \subseteq M_{All}$) and/or a reduced number of tests ($T_i \subseteq T_{All}$) are taken into account, the mutation score can be less accurate that the original one. In order to estimate the mutation testing approach not only in terms of the mutation score accuracy but also the cost factors, the quality metrics were proposed [7]. Using the metrics it is possible to compare results of different programs and different experiments, as it is based on a normalization function and takes therefore values from 0 to 1.

The quality metric EQ applied in the analysis of mutant clustering is a weighted sum of three components (Eq. 1)

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

The metric is based on three dependent variables that assess a decline of mutation score accuracy (S_{MS}), a reduced number of tests required to kill mutants (Z_T), and a

reduced number of mutants (Z_M). Contributions of these factors to the metric are calibrated by weight coefficients $W_{MS,}$ W_T, W_M, which sum must be equal to 1. $I()$ denotes a normalization function. The normalization is performed for all results in an experiment. Detailed formulae of the variable computation are given in [7], where the quality metric was applied for quality evaluation of selective mutation.

4 Experimental Set-Up

The mutation testing process discussed in this chapter dealt with *first order mutation* - a mutant is created by introducing one fault specified by one mutation operator in a program under test, and *strong mutation* - a mutant is recognized to be killed if a result of at least one test differs from the result of the original program.

Three widely used, open-source programs related to different domains and various authors were used in the experimental study. The basic statistics of the programs are given in Table 1. The tests associated with the first project - Enterprise Logging were unit tests designed and run with MSTest, a part of the Microsoft Visual Studio, whereas tests of Castle and Mono-Gendarme were NUnit tests.

Table 1 Subject programs and their statistics

No	Program	LOC		Classes & Interfaces	
		with tests	without tests	with tests	without tests
1	Enterprise Logging http://entlib.codeplex.com	87552	57885	991	587
2	Castle http://www.castleproject.org	54496	41288	724	493
3	Mono-Gendarme http://www.mono-project.com/Gendarme	51228	25692	907	171
	Sum	193276	124865	2622	1251

The experiments were carried out with the CREAM (CREAtor of Mutants) tool a mutation system for C# programs mutated at the syntax tree level [6,8,9]. It is the most mature mutation system of C# applications. The latest version of the tool was extended with a wizard in order to efficiently perform experimental study on cost reduction techniques. The extension assists in creating mutants, executing tests, and evaluating test results in respect to three methods: mutation operator selection, mutant sampling and mutation clustering.

The experimental scenario from Sec. 3.2 and the whole analysis were preformed independently for two sets of mutation operators: 18 object-oriented and 8 standard ones implemented in CREAM v3 [7]. The standard operators cover the five operators distinguished to be selective [11].

5 Experiment Results and Quality Analysis

The basic mutation testing results of subject programs are summarized in Table 2.

The first row includes numbers of mutants referring only to the covered code. The programs were covered by their unit tests in 82%, 77% and 87% respectively. None of the mutants generated for uncovered code were killed. Therefore in the calculations of the mutation score only the covered code was taken into account.

For each program, mutants were run against all tests T_{All} associated with the program. The numbers of killed mutants are given in the second row.

Some generated mutants can be equivalent. Equivalent mutants were manually detected by analyzing mutants generated by selected operators (with the highest number of not killed mutants and those easily to be analyzed). The numbers of recognized equivalent mutants are listed in the third row. However, some equivalent mutants could remain undetected. Covered and not recognized as equivalent mutants were counted as a set of all mutants M_{All} generated by either OO or standard operators, respectively. Basing of this data the original mutation score MS_{orig} was calculated. It is given in the last row and will be counted as a reference value.

Table 2 Mutation results - number of mutants generated, killed, equivalent and mutation score

	1. Enterprise Logging		2.Castle		3. Mono-Gendarme	
	O-O	Standard	O-O	Standard	O-O	Standard
Generated covered mutants	1341	1683	1208	2379	998	4153
Killed mutants	558	1151	701	1611	478	3009
Equivalent mutants	438	60	143	60	143	79
Mutation Score (MS_{orig}) [%]	61,79%	70,92%	65,82%	69,56%	55,91%	73,86%

The test results of all mutants were used in further steps C(C1-C4) of the mutation clustering scenario (Sec. 3.2) performed under the following assumptions:

- the clustering parameter K varied from 0 to 19,
- *TestSetLimit* - the number of minimal test sets in collection L was set to 15.

Average mutation score (step C4) calculated for different values of the parameter K=1..19 is shown in Table 3. This value reflects an average mutation result that could be obtained if we used not all mutants but only its subset - representatives of groups determined with a given K parameter. In general, higher values of K result in the drop of mutation score, although the functions are not strictly monotonous. This effect is caused by selection of one mutant representing a group.

If no clustering is made (K=0), the values are slightly higher than for clustering with K=1 and equal to the reference values MS_{orig} given in Table 2.

Table 3 Average Mutation Score in dependence on the clustering parameter *K* in [%]

Clustering parameter	1. Enterprise Logging		2.Castle		3. Mono-Gendarme	
	O-O	Standard	O-O	Standard	O-O	Standard
1	61.03%	70.18%	63.89%	67.43%	53.57%	69.43%
2	52.88%	63.96%	54.82%	65.20%	43.76%	65.33%
3	49.45%	62.27%	53.08%	63.04%	40.67%	60.82%
4	40.76%	59.79%	46.57%	61.52%	34.39%	54.97%
5	44.30%	59.12%	46.95%	59.27%	34.63%	55.48%
6	38.29%	49.39%	44.30%	59.47%	34.12%	49.40%
7	36.30%	45.48%	42.00%	54.25%	35.00%	46.14%
8	33.92%	44.85%	39.23%	53.88%	31.44%	47.09%
9	30.34%	43.61%	40.72%	55.98%	29.70%	45.77%
10	33.04%	42.98%	37.86%	56.70%	28.60%	42.97%
11	26.22%	37.56%	38.34%	54.21%	24.35%	44.43%
12	30.74%	41.94%	37.50%	53.10%	23.87%	40.88%
13	27.98%	43.77%	37.61%	50.47%	22.30%	39.29%
14	27.80%	34.36%	36.46%	54.42%	26.23%	36.03%
15	28.45%	29.64%	34.79%	48.46%	22.13%	35.72%
16	25.24%	42.82%	36.16%	52.99%	19.42%	35.20%
17	26.01%	39.02%	33.55%	51.76%	20.88%	33.89%
18	28.52%	28.43%	37.06%	51.16%	19.56%	33.71%
19	24.02%	30.04%	33.15%	47.57%	20.16%	33.88%

Quality analysis was aimed at assessing a tradeoff between the decline of mutation score (visible in Table 3) and a possible cost reduction counted in terms of mutant and test number. Based on experiment results, the quality metric *EQ* (Sec. 3.3) was calculated for different values of the clustering parameter. Table 4 comprises quality values calculated assuming the weight coefficients $W_{MS,}$ W_T , W_M equal to 0.6, 0.2, 0.2 accordingly, i.e. the mutation score accuracy amounts to 60% in the quality measure whereas efficiency factors to 40% (20% for the number of mutants and 20% for the number of tests). The clustering parameter *K* varies from 0 to 7, as the mutation score was too inaccurate for the higher thresholds.

For OO operators, the best quality of projects 1 and 2 was reached for the clustering parameter *K*=1. This means that mutants in a cluster are killed by test sets including only one different test case. The OO quality of 3rd project was the highest with no clustering, although the quality for *K*=1 was also close to 1.

Table 4 Quality Metrics EQ in dependence of the clustering parameter *K*

Clustering parameter	1. Enterprise Logging		2.Castle		3. Mono-Gendarme	
	O-O	Standard	O-O	Standard	O-O	Standard
0	0.89	0.73	0.94.	0.89	1.00	0.75
1	1.00	1.00	1.00	0.93	0.97	0.86
2	0.98	0.90	0.73	1.00	0.79	1.00
3	0.82	0.91	0.71	0.87	0.65	0.86
4	0.52	0.98	0.49	0.91	0.47	0.73
5	0.87	0.98	0.55	0.72	0.51	0.80
6	0.51	0.63	0.41	0.84	0.59	0.55
7	0.47	0.47	0.31	0.32	0.67	0.43

Quality metric for standard operators applied to projects 2 and 3 reached maximum when *K*=2. In case of project 1, the maximum is when *K* equals 1, but other values (*K*=2,3,4,5) gave also good results (above 0.9). It should be noted that for higher values of the parameter (*K*=3,…7) the results of standard operators were in the most cases significantly better (0.1-0.3 higher) than the OO results

While generalizing results, the potential data (mutation score, number of mutants and number of required tests) are compared with the original values without clustering (Table 5). The clustering parameter was assumed to be 1 for OO operators and 2 for standard ones. Results for OO operators averaged for all projects showed that while using 32% of all mutants and 17% of tests, we could obtain 97% of the original mutation score. For standard operators the results of 19% of mutants and 22% of tests could give *MS* with 91% of the original accuracy.

Table 5 Clustering results for OO and standard mutation

Program		Object-oriented (cluster *K* = 1)			Standard (cluster *K* = 2)		
		Mutation Score [%]	Mutant number	Test number	Mutation Score [%]	Mutant number	Test number
1.Enterprise Logging	Original	61.79%	903	1148	70.92%	1623	1148
	Cluster.	61.03%	295	139	63.96%	221	110
2. Castle	Original	65.82%	1065	642	69.56%	2316	642
	Cluster.	63.89%	333	154	65.20%	681	145
3. Mono-Gendarme	Original	55.91%	855	899	73.86%	4074	899
	Cluster.	53.57%	282	140	65.33%	545	312
Average change [%]		97.2 %	32.3%	17.2%	90.8%	18.8%	22.3%

Time of mutation testing should be decreased when the reduced number of mutants and tests are applied. Effective times of mutant generation and test execution are given in Table 6 and compared with times necessary to generate all mutants

Table 6 Times of mutant generation (including compilation) and of test execution [h:min:sec]

Program	Object-oriented (cluster $K = 1$)		Standard (cluster $K = 2$)	
	Mut. gener. time	Test exec. time	Mut. gener. time	Test exec. time
1 All	06:26:11.2	06:32:36.6	07:22:44.2	11:45:39.2
1 Cluster.	02:07:34.0	00:20:24.0	01:01:44.0	00:07:58.7
2 All	05:37:43.9	07:14:14.2	10:36:59.7	15:44:18.9
2 Cluster.	01:50:54.0	00:56:05.6	01:49:21.0	01:27:42.8
3 All	03:49:32.0	02:02:28.7	13:53:38.9	09:43:36.0
3 Cluster.	01:05:49.0	00:12:45.4	01:54:29.0	00:24:52.2

and execute all mutants with all tests. On average, time of generating a reduced number of mutants took about 30% and 15% of the original time, and time of execution of all test 9% and 5%, for OO and standard mutants respectively.

The programs were quite complex and widely used; however, conclusion validity of experiments is limited due to a small number of programs (three). All programs were of open-source origin that could object external validity.

Mutation score measured with test sets distributed with the projects was low. To alleviate this threat to construct validity additional tests were designed, but the results were still below 100%. Construct validity can also be influenced by metrics and parameter selection. Therefore the analysis was performed for a wide range of K parameter. It also was repeated for another set of weight coefficients: $W_{MS,}$ W_T , W_M equal to 0.8, 0.1, 0.1. In this case mutation score accuracy was more important (0.8) and the quality was maximal when K=0 for both OO and standard operators.

6 Conclusions

It was shown, that potential profits of mutation clustering for C# programs could be considerable. While using only 32% or 19% of all mutants and 18% or 22% of tests, the mutation score could be of 97% or 91% close to the original one, for OO and standard mutation operators respectively. In comparison, analogous results for mutant sampling were about 33%, 30% of mutants, 10%, 15% of tests resulting in 85% and 93% of mutation score accuracy [10]. Another method for reduction of mutant and test number - selective mutation gave better accuracy but with a smaller decline of mutant number and analogous number of tests [7].

However, mutation clustering is more difficult to be implemented than selective mutation or mutant sampling, because we generate unnecessary mutants. In a practical approach to clustering, applying statically domain analysis [5], all mutants should be generated but we could benefit from reduced number of test runs with a reduced number of mutants. Concluding, if the potential lowering of

mutation testing complexity and accuracy of mutation results are comparable it would be worthwhile implement methods that are easier to be generalize.

Combining those methods with other approaches to cost reduction, e.g. omitting of redundant mutants [18] or test prioritization [19], remains an open issue.

Acknowledgments. I am very thankful to my student M. Rudnik for extending the CREAM tool and performing mutation testing experiments.

References

[1] Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Transactions on Software Engineering 37(5), 649–678 (2011), doi:10.1109/TSE.2010.62

[2] Usaola, M.P., Mateo, P.R.: Mutation testing cost reduction techniques: a survey. IEEE Software 27(3), 80–86 (2010), doi:10.1109/MS.2010.79

[3] Hussain, S.: Mutation Clustering. Ms. Thesis, King's College London, Strand, London (2008)

[4] Jain, A.K., Murty, M.N., Flynn, P.J.: Data Clustering: A Review. ACM Computing Surveys 31(3), 264–323 (1999)

[5] Ji, C., Chen, Z.Y., Xu, B.W., Zhao, Z.H.: A novel method of mutation clustering based on domain analysis. In: Proc. of 21st Inter. Conf. on Softw. Eng. & Knowledge Eng., pp. 422–425 (2009)

[6] 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. Springer, Heidelberg (2012)

[7] 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 2012. LNCS, vol. 7304, pp. 42–57. Springer, Heidelberg (2012)

[8] Derezińska, A., Szustek, A.: Tool-supported mutation approach for verification of C# programs. In: Zamojski, W., et al. (eds.) Proc. of Inter. Conf. on Dependability of Computer Systems, DepCoS-RELCOMEX 2008, pp. 261–268 (2008), doi:10.1109/DepCoS-RELCOMEX.2008.51

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

[10] Derezińska, A., Rudnik, M.: Empirical evaluation of cost reduction techniques of mutation testing for C# Programs, Warsaw University of Technology, ICS Res. Rep. 1/2012 (2012)

[11] Offut, J., Rothermel, G., Zapf, C.: An experimental evaluation of selective mutation. In: Proc. of 15th Inter. Conf. on Software Engineering, pp. 100–107 (1993)

[12] Zhang, L., Hou, S.-S., Hu, J.-J., Xie, T., Mei, H.: Is operator-based mutant selection superior to random mutant selection? In: Proc. of the 32nd International Conference on Software Engineering, ICSE 2010, pp. 435–444 (2010), doi:10.1145/1806799.1806863

[13] Kaminski, G., Praphamontripong, U., Ammann, P., Offutt, J.: A logic mutation approach to selective mutation for programs and queries. Inform. and Softw. Technol. 53, 1137–1152 (2011), doi:10.1016/j.infsof.2011.03.009

[14] Hu, J., Li, N., Offutt, J.: An analysis of OO mutation operators. In: Proc. of 4th Inter. Conf. Softw. Test. Verif. and Validation Workshops, pp. 334–341 (2011), doi:10.1109/ICSTW.2011.47

[15] Mathur, A.P., Wong, W.E.: Reducing the cost of mutation testing: an empirical study. J. of Systems and Softw. 31, 185–196 (1995)
[16] Derezińska, A., Kowalski, K.: Object-oriented mutation applied in Common Intermediate Language programs originated from C#. In: Proc. of 4th International Conference Software Testing Verification and Validation Workshops, pp. 342–350 (2011), doi:10.1109/ICSTW.2011.54
[17] Kryszkiewicz, M.: Fast algorithm finding minima in monotonic Boolean functions, Warsaw Univ. of Technology, ICS Res. Rep. 42/93 (1993)
[18] Just, R., Kapfhammer, G.M., Schweiggert, F.: Do redundant mutants affects the effectiveness and efficiency of mutation analysis? In: Proc. IEEE 5th Inter. Conf. on Software Testing, Verification and Validation, pp. 720–725 (2012), doi:10.1109/ICST.2012.162
[19] Zhang, L., Marionov, D., Zhang, L., Khurshid, S.: Regression mutation testing. In: Proc. of Int. Symp. on Software Testing, ISSTA 2012, pp. 331–341 (2012)