Quality Evaluation of Object-Oriented and Standard Mutation Operators Applied to C# Programs

Anna Derezińska and Marcin Rudnik

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

Abstract. Mutation testing is a kind of fault injection approach that can be used to generate tests or to assess the quality of test sets. For object-oriented languages, like C#, both object-oriented and standard (traditional) mutation operators should be applied. The methods that can contribute to reducing the number of applied operators and lowering the costs of mutation testing were experimentally investigated. We extended the CREAM mutation tool to support selective testing, sampling and clustering of mutants, and combining code coverage with mutation testing. We propose an approach to quality evaluation and present experimental results of mutation operators applied to C# programs.

Keywords: mutation testing, object-oriented mutation operators, C#.

1 Introduction

Mutation testing is a fault-injection technique that can be used for assessment of test set quality and support for test case generation [1]. Once a defined fault is introduced in a program, a mutated program (*mutant*) is created. A program modification is determined by a *mutation operator*. Within this paper we deal with first order mutation, i.e. one mutation operator is applied in one place. If a mutant behavior differs from that of the original program while running against a test case, the mutant is said to be *killed* by this test. The test is effective at killing the mutant. A quality of a test set is a mutation score MS calculated as the ratio between the number of killed mutants over all the generated but not equivalent mutants. An *equivalent* mutant has the same behavior as the original program and therefore cannot be killed by any test.

Mutation testing process is counted as a very cost-demanding testing activity. The cost is determined by the number of generated mutants, the number of tests involved in the process and their ability to test mutants, the number of equivalent mutants and their recognizing, a kind of a mutation tool support, etc.

Important factors of mutation testing are mutation operators that reflect possible faults made by programmers and therefore should deal with different constructs in programming languages. In C# programs, as for any general purpose language, standard (i.e. structural, intra-class or statement-level) operators can be applied, e.g. dealing with logical, arithmetical, relational operators like those defined in Fortran or C. Moreover, object-oriented (or inter-class) operators should also be used. Operators

C.A. Furia and S. Nanz (Eds.): TOOLS Europe 2012, LNCS 7304, pp. 42–57, 2012.
© Springer-Verlag Berlin Heidelberg 2012

of object-oriented features were primarily defined for the Java language [2-4]. Applicability of those operators was studied for C# programs, and analogous operators of the same or similar scope were proposed [5]. Its set was extended with new operators, for example dealing with exception handling, or devoted to the programming constructs specific to C# but not present in Java, like delegates or properties [6]. Empirical evaluation of object-oriented and other advanced operators in C# programs was conducted on above 13 thousands of mutants [6-11].

However, a quality of object-oriented mutations, both in Java or C#, remains still an open question in the relation to the cost estimation. A problem is, which object-oriented operators we really need, and which of them can be omitted without loosing the ability to qualify a given test set. Should we reduce the cost by selecting operators, random sampling of mutants, or other reduction techniques? These issues were studied for structural languages [12-15] and partially for Java programs [16-18].

According to our experience [6-11], object-oriented operators generate fewer mutants than the standard ones, and are more dependant on the concerned programs. Therefore we extended the CREAM mutation testing tool for C# programs with the facility to carry out experiments into cost reduction techniques. They cover operator selecting, mutant random sampling and clustering [1, 19-22]. Using the tool some experiments on fist-order mutation in C# programs were conducted. They showed that relations between different cost techniques are not necessarily the same as for the standard mutation in C [15]. The main contributions of the paper are:

- Evolution of the mutation testing tool for C# programs, and incorporated processes of an empirical and statistical analysis of mutation results.
- New quality metrics for assessment of a tradeoff between mutation score accuracy and mutation costs in terms of number of mutants and number of tests.
- The first experiments on the selective mutation of C# programs performed and analyzed for 18 object-oriented (OO in short) and 8 standard mutation operators.
- The first general estimation of results for experiments on mutant sampling and clustering for object-oriented mutation.

The paper is organized as follows: Section 2 summarizes briefly the main features of the CREAM mutation testing tool. In Section 3 we give details about selected investigation processes incorporated in the tool and the quality metrics. Section 4 describes an experimental set-up and results of the conducted experiments. Finally, the remaining sections present related work and conclusions.

2 CREAM Mutation Testing Tool for C# Programs

CREAM (CREAtor of Mutants) was the first mutation testing tool supporting object-oriented mutation operators for C# programs [7,8,23]. It is a parser-based tool. A fault defined by a mutation operator is introduced into a parser output after analysis of a C# project. Then the C# source code is reconstructed from the modified syntax tree. It can be compiled, so creating a mutated program that can be run against a test set.

Currently, the next, third version of the tool is ready to use. It was extended to support more mutation operators, to keep-up with new versions of the C# language and cooperate with new tools in order to work on emerging real-word applications. It can create mutants for the whole code or only for the code covered by the test cases, if

required. Moreover, it was equipped with a wizard aimed at evaluation of detailed statistics and supporting experimental studies on mutation operator assessment. The most important functionalities of the current version of CREAM are as follows:

1. It supports parser-based generation of first order mutants of C# programs with 18 object-oriented operators and 8 selected standard operators, listed in Tab. 1.
2. It runs mutants against test suites and evaluates test results. Unit tests can be compatible with the NUnit tool [24] or with MSTest (another tool built in Microsoft Visual Studio).
3. It optionally takes into account code coverage results while creating mutants. The coverage data can be delivered by NCover [25] (.xml files), Microsoft Visual Studio (.coverage files) or Tester [9] (.txt files).
4. It optionally stores mutants in the local or remote SVN repository [26] in order to reduce an occupied disk space [9].
5. It automates analysis of generated mutants according to cost reduction techniques: mutation operator selection, mutant sampling and clustering.
6. It evaluates statistics of many experiments, and enables presentation of output data in cooperation with a Data Viewer tool.

Table 1. Mutation operators: standard and object-oriented supported in CREAM v.3

No	Type	Abbreviation	Name
1	Standard	ABS	Absolute Value Insertion
2	Standard	AOR	Arithmetic Operator Replacement (+, -, *, /, %)
3	Standard	ASR	Assignment Operator Replacement (=, +=, -=, /=, *=)
4	Standard	LCR	Logical Connector Replacement (&&, ll)
5	Standard	LOR	Logical Operator Replacement (&, l, ^)
6	Standard	ROR	Relational Operator Replacement (<, <=, >, >=, ==, !=)
7	Standard	UOI	Unary Operator Insertion (+, -, !, ~)
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

3 Investigation Process of Mutation Operators

In this section we explain the basic setup of an experimental process. The empirical evaluation of mutant features can be considered as several experimental scenarios: three relating to selective mutation, six to mutant sampling, and one to mutant clustering. Mutant sampling refers to random selection of a given % of all mutants or of mutants uniformly distributed for all classes, files, methods, operators or namespaces. The examination of mutation results is done independently for object-oriented operators and standard ones. For the brevity reasons we only describe the general experimental scenario of all types of experiments and the details of selective mutation. The details of sampling and clustering scenarios are omitted [27].

3.1 Generic Scenario of Experiments

A) In the first step, common for all experiments, all mutants of a program under test are generated for a given set of operators. In this case all standard and all object-oriented operators available in CREAM v.3 were used independently (Tab. 1). This original set of all mutants is called M_{All}.

B) Secondly, all mutants are run against a whole set of tests T_{All} considered as a basic, complete set of tests taken into account in the experiments for a given program. Results of all mutants and all tests are stored in a file and can be examined during different experiments, or viewed by a user. Taking into account these results a mutation score can be calculated. Further we refer to this value as to the "original" mutation score $MS_{orig.} = MS\ (M_{All}, T_{All})$.

C) Next, the subsequent steps (C1-C4) and D are repeated many times in accordance to different parameters specific to each kind of experiments (e.g. numbers of mutation operators, number of kinds of mutant sampling, etc.).

C1) A subset of mutants $M_{C1} \subseteq M_{All}$ is selected according to given criteria of the experiment. This set determines the maximal mutation score generated using all tests $MS_{C1max} = MS\ (M_{C1}, T_{All})$.

C2) A list L of subsets of T_{All} is created. Usage of any test set in L against M_{C1} gives the mutation score equal to MS_{C1max}. But each test set of L is minimal, i.e. all tests are necessary. All test sets of this kind can be generated using prime implicant of a monotonous Boolean function [27]. The number of all test sets is finite but can be high. Therefore the cardinality of the list L is limited, $|L| \leq TestSetLimit$. It is fixed as a parameter of an experiment that restricts its complexity. $|X|$ is cardinality of set X.

C3) For a set $M_{C3} \subseteq M_{All}$, mutation scores MS_{C3j} are calculated using consecutively each minimal test set in L: $MS_{C3j} = MS(M_{C3}, T_j)$, where $T_j \subseteq T_{All}$, $T_j \in L$, $j=1..|L|$. The considered mutant set M_{C3} depends on the type of the experiment (Sec. 3.2).

C4) The average mutation score $MS_{avg} = (\sum_{j=1..|L|} MS_{C3j}) / |L|$ is calculated for all minimal test sets in L. The average number of test sets in L is $NT_{avg} = (\sum_{j=1..|L|} |T_j|) / |L|$.

D) In mutant sampling, calculation of average statistics for many random runs of steps C1-C4 for a given experiment parameter. Basing on data from C4 new average values MS_{avg}, NT_{avg} are calculated over those repeated runs for this parameter.

E) Calculation of final statistics and normalization of results (see Sec. 3.3).

3.2 Experimental Flows on Selective Mutation

In selective mutation, only a subset of mutation operators is used. Considering different policies of operator selection, and determination of minimal test sets and sets of mutants used for evaluation of a final mutation score, three kinds of experiments are supported in CREAM.

Experiment 1. Mutation operators that generate the biggest number of mutants are excluded. Therefore, omitting the fewer number of operators the biggest number of mutants could be not used.

In these experiment, steps C1-C4 are repeated for different numbers of excluded operators ($i = 0..k$, where k is a number of all operators, in this case 8 for standard and 18 for object-oriented ones). In general, for a given value of i it could be the binomial coefficient C(k,i) of various subsets including i operators. But we exclude i operators generating the biggest number of mutants. In almost all cases there was exactly one such subset with i operators, otherwise one such subset was randomly selected.

In step C1, M_{C1} is the subset of M_{All} containing the mutants generated by the operators not excluded in the current experiment run.

In step C3, the mutation scores are calculated for all mutants, i.e. $M_{C3} = M_{All}$.

Step D is not used.

Experiment 2. One mutation operator is excluded. The selection is performed for each mutation operator separately. In this way all mutation operators are examined in accordance to their influence on the mutation score result.

The number of repetition of steps C1-C4 is equal to the number of considered mutation operators. Step D is not used.

In step C1, we determine M_{C1} as the subset of M_{All} containing the mutants generated by the operators not excluded in the current experiment run.

In step C3, $M_{C3} = M_{All}$.

Experiment 3. One mutation operator is excluded, similarly as in the second type of the selective experiment (steps C1 and C2 are the same). But, in this case the mutation scores in step C3 are calculated for the set of mutants M_{C3} generated by the operator excluded in the current experiment run ($M_{C1} \cup M_{C3} = M_{All}$). Step D is not used.

The motivation of this experiment is assessment of an operator quality. A “good” operator is an operator for which no mutants are generated by other operators and are killed by the same tests. A “poor” operator can be counted as a redundant one, tests that kill other mutants can also kill mutants generated by this operator.

3.3 Quality Metrics

The cost of mutation testing process is influenced by different factors, mainly the cost of mutant generation, of running mutants against tests, dealing with equivalent mutants and analyzing test results. Reduction of the cost can cause decrease of mutation score accuracy. Therefore, we proposed metrics to assess a tradeoff concerning the loss of MS accuracy on the one hand and profits of using a smaller

number of mutants and smaller number of tests on the other hand. It assists in comparing results for different programs and different experiments.

The quality metric *EQ* takes into account three variables:

S_{MS} - reflects a loss of Mutation Score adequacy (MS) in an experiment,

Z_T - approximates a profit of a cost decrease due to a reduced number of tests required for killing mutants in an experiment,

Z_M - assesses a profit of a cost decrease due to a reduced number of mutants considered in an experiment.

These variables are calculated as given in Equations (1)-(3).

The first variable S_{MS} is evaluated in a different way in accordance to the experiment type. It is calculated as a ratio of the current average mutation score to the original mutation score if we look for a possible smallest difference to MS_{orig}. Otherwise, if we are interested in the biggest difference, S_{MS} is equal to 1 minus the ratio mentioned in the previous case (Eq. 1). The average mutation score MS_{avg} is calculated for all minimal test sets in step C4 (or in D if applicable).

$$S_{MS} = \begin{cases} MS_{avg} / MS_{orig} & \text{in } 1^{st} \text{ selective experiment, sampling, clustering} \\ 1 - (MS_{avg} / MS_{orig}) & \text{in } 2^{nd} \text{ and } 3^{rd} \text{ selective experiment} \end{cases} \quad (1)$$

Variable Z_T is calculated as 1 minus a ratio of an average number of tests for a current experiment divided by the number of all tests considered for a program, if the divident is bigger than zero. Otherwise Z_T is equal to zero (Eq. 2). The average number of tests NT_{avg} is defined in step C4 (or in D if used for many random cases).

$$Z_T = \begin{cases} 1 - (NT_{avg} / |T_{All}|) & \text{if } NT_{avg} > 0 \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

Value Z_M is equal to 1 minus a ratio of a mutant number currently taken into account in the experiment ($|M_{C1}|$ from step C1) to a maximal number of all mutants generated for the program (in the set M_{All}), if the current mutant number is bigger than zero. Otherwise Z_M is set to zero (Eq. 3).

$$Z_M = \begin{cases} 1 - (|M_{C1}| / |M_{All}|) & \text{if } |M_{C1}| > 0 \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

Next, the variables obtained for different parameters of an experiment are normalized. The normalization function *NORM(x)* represents a normalized value of variable *x* over a set of its values *X* (Eq. 4).

$$NORM(x) = (x-MIN(X)) / (MAX(X)- MIN(X)), \quad where \quad x \in X \quad (4)$$

In result, the normalized variable *x* will be distributed within the <0,1> interval and can be further processed in an comparable way. The normalization is calculated for a set of results determined by experiment parameters. For example, in the second selective experiment on object-oriented operators the set of variables correspond to

exclusion of one selected operator. In this case cardinality of the set *X* equals the number of operators (18).

The quality metric *EQ* is based on a weighted sum of three components (Eq. 5).

$$EQ(W_{MS}, W_T, W_M) = NORM(W_{MS}*NORM(S_{MS}) + W_T*NORM(Z_T) + W_M*NORM(Z_M)) \quad (5)$$

The weight coefficients W_{MS}, W_T, W_M state for parameters of the analysis and are determined according to the importance assigned to particular components of the metric. The sum of coefficients must be equal to 1.

The whole metric is also normalized over the set of values calculated for different parameters of an experiment, similarly as the variables.

4 Experiments

In this section we describe the subject programs and their results of mutation testing. Outcomes of experiments on selective mutation are also discussed.

4.1 Investigated Programs

Objects of experiments were three, commonly used open-source programs. They were selected to cover different types of complexity, application domain, and origin.

1. Enterprise Logging [http://entlib.codeplex.com] - a module from the "pattern & practices" library developed by Microsoft. It is used for logging information about code faults.
2. Castle [http://www.castleproject.org] - a project supporting development of advanced applications in .NET. Four modules were used in experiments: Castle.Core, Castle.DynamicProxy2, CastleMicroCernel and Castle.Windsor.
3. Mono Gendarme [http://www.mono-project.com/Gendarme] - a tool for inspection of programs written in Mono and .NET environments. It looks for flaws not detected by a compiler.

The following measures of the programs are summarized in Table 2.

1. *Files* - number of file modules with the source code included in a program.
2. *Statements* - number of statements in a program.
3. *Lines* - number of all lines in a project including comments.
4. *Percent comment lines* - % of lines with comments among all program lines.
5. *Percent documentation lines* - % of lines with documentation among all program lines.
6. *Class, Interfaces, Structs* - number of all such items defined in a program.
7. *Methods per Class* - average number of methods per class (a ratio of the number of all methods to the number of all defined classes, structs and interfaces).
8. *Calls per method* - average number of calls of other methods in a given method.
9. *Statements per method* - average number of statements in a method.
10. *Maximum complexity* - maximal number of conditional decisions in a method.
11. *Average complexity* - average number of conditional decisions in methods.
12. *Depth of inheritance* - maximal number of inheritance levels in a project.

Table 2. Details of tested programs (measured with SourceMonitor [28] and Microsoft Visual Studio)

No	Measure	1.Enterprise Logging		2.Castle		3.Mono Gendarme	
		with tests	without tests	with tests	without tests	with tests	without tests
1	Files	662	497	533	403	291	170
2	Statements	33451	17427	20284	14001	21739	9715
3	Lines	87552	57885	54496	41288	51228	25692
4	% comment lines	8.1	9.1	13.7	13.8	18.9	21.9
5	% document. lines	19.2	29.0	11.4	14.6	9.9	19.5
6	Classes, Interfaces,	991	587	724	493	907	171
7	Methods per class	5.4	5.9	5.0	5.4	3.8	5.1
8	Calls per method	3.1	1.3	2.7	2.2	2.0	2.7
9	Statem. per method	3.3	2.3	3.2	2.9	3.7	7.9
10	Max. complexity	14	14	25	25	53	28
11	Average complex.	1.3	1.5	1.6	1.8	2.0	4.0
12	Depth of inherit.	6	6	4	4	10	3

4.2 Mutant Generation and Execution

The tested programs were distributed with unit tests. For two programs additional test cases were prepared in order to increase their code coverage results (Table 3). The first program has unit tests compatible to MSTest and its code coverage was evaluated using functionality build in Microsoft Visual Studio. Remaining two programs have unit tests for NUnit [24] and were examined with the NCover coverage tool [25].

Table 3. Code coverage results (measured with NCover [25] and Microsoft Visual Studio)

	1. Enterprise Logging	2. Castle	3. Mono Gendarme
	MSTest tests	NUnit tests	NUnit tests
Number of original tests	1148	578	784
Number of additional tests	0	64	115
All test cases	1148	642	899
Line coverage [%]	82	77	87

If demanded, CREAM v3 can generate mutants for these code statements that were covered by tests from a test suit under consideration, when the appropriate coverage data are provided. According to our experiences, if MS was evaluated, it was useless to mutate the code not covered by these tests. Just in case, all possible mutants of the programs discussed in this paper were generated and run against tests. But none of uncovered mutants was killed by any test. Therefore, as generated mutants (column *Gen* in Table 4.) are only counted these mutants that are created by modification of covered code lines. Only this sort of mutants is used further in the calculation of mutation results. Based on the uncovered code we obtained 265 standard and 336 object-oriented mutants for Enterprise Logging, 448 and 367 for Castle and 392, 449 for MonoGardarme, accordingly. These uncovered mutants were discarded.

Table 4 presents mutation results for each standard and object-oriented operator implemented in CREAM v3. The full names of the operators are given in Tab. 1. Columns Kill include numbers of mutants killed using all tests defined in Tab. 2.

Table 4. Mutation results (mutants generated, killed, equivalent, and mutation score in [%])

Opera	1. Enterprise Logging				2. Castle				3. Mono Gendarme			
tor	Gen	Kill	Eq	MS	Gen	Kill	Eq	MS	Gen	Kill	Eq	MS
ABS	114	7	60	13%	102	9	60	23%	116	3	79	8%
AOR	328	322	-	99%	68	19	-	28%	88	67	-	76%
ASR	160	97	-	61%	98	43	-	44%	85	54	-	64%
LCR	34	27	-	79%	196	138	-	70%	417	270	-	65%
LOR	2	0	0	0%	2	0	0	0%	16	14	-	88%
ROR	220	141	-	64%	645	427	-	66%	900	575	-	64%
UOI	795	537	-	68%	1070	842	-	79%	2342	1920	-	83%
UOR	30	20	-	67%	198	133	-	67%	189	106	-	56%
Sum	1683	1151	60	71%	2379	1611	60	70%	4153	3009	79	74%
DMC	0	0	0	-	0	0	0	-	0	0	0	-
EHR	9	6	0	67%	8	3	4	75%	5	5	-	100%
EOA	22	0	21	0%	23	2	13	20%	7	2	1	33%
EOC	98	43	18	54%	494	209	119	56%	536	159	124	39%
EXS	22	1	10	8%	11	3	0	27%	3	0	1	0%
IHD	0	0	0	-	0	0	-	-	0	0	0	-
IHI	1	0	0	0%	0	0	-	-	0	0	0	-
IOD	21	20	-	95%	13	10	-	77%	10	9	-	90%
IOK	20	19	-	95%	13	9	-	69%	10	8	-	80%
IOP	20	7	11	78%	7	2	4	67%	34	22	-	65%
IPC	45	35	-	78%	39	31	-	79%	0	0	-	-
ISK	51	32	-	63%	18	11	-	61%	30	30	-	100%
JID	80	32	24	57%	143	106	-	74%	155	135	-	87%
JTD	458	52	353	50%	48	38	3	84%	17	0	17	0%
OAO	164	115	-	70%	212	117	-	55%	142	66	-	46%
OMR	17	16	-	94%	54	50	-	93%	0	0	-	-
PRM	17	11	1	69%	16	12	-	75%	15	10	-	67%
PRV	296	169	-	57%	109	98	-	90%	34	32	-	94%
Sum	1341	558	438	62%	1208	701	143	66%	998	478	143	56%

CREAM prevents in some cases, especially for OO operators, from creating of equivalent mutants, but still many such mutants can be obtained. Therefore, some not killed mutants were examined manually. First, a preliminary mutation indicator was calculated for each operator and a program (i.e. the number of killed mutants divided be the number of generated mutants). If the indicator was below 50% for an OO operator, or below 40% for a standard one, mutants generated by this operator were examined, whether they are equivalent or not. These thresholds were selected after the empirical evaluation of data. In addition, we checked those mutants that were easily to be verified.

Mutants examined of being equivalent are denoted in Table 4. In column Eq a number of detected equivalent mutants is given ("-" states for not examined mutants).

Finally, a mutation score (column MS) was evaluated, as a ratio of killed mutants to generated but not equivalent mutants.

4.3 Experiments on Selective Mutation

The experiments were conducted according to assumptions given in Sec. 3, and for the limit of minimal test sets *TestSetLimit* equal to 100. The required quality condition is that a decrease of the mutation score accuracy is acceptable while there are considerable benefits in the cost reduction in terms of the lowering of the mutant number and the number of tests required for killing those mutants. The quality metrics *EQ* given in tables (Tab. 5, 6) were calculated for 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).

1st Experiment on Selective Mutation - Exclusion of the Most Popular Operators
The experiment investigates how many mutation operators that generate the biggest number of mutants (and which of them) can be omitted. Selection of less than 8 standard, or less than 8 object-oriented operators to be excluded was unique in all cases (comp. Sec. 3.2). In general, excluding mutation operators that generate the biggest numbers of mutants results in the decrease of the mutation score even for one operator (Table 5.), regardless standard or object-oriented operators were concerned. In the first column a number of excluded operators is given. Columns MS1 include average mutation scores (in %) calculated under these conditions.

Considering potential profits in a reduced number of mutants and tests, the quality metric was calculated (*EQ1*). Selecting the quality value above 90% for object-oriented operators, we obtained different sets of operators to be excluded. The common result for all programs was elimination of two operators EOC and OAO.

In the case of the standard operators, the results for different programs are more similar to each other than for object-oriented ones. A maximal quality value was obtained for one or two excluded operators. Assuming a quality value about 90% the common two operators to be excluded are UOI and ROR .

2nd Experiment on Selective Mutation - Exclusion of One Mutation Operator
In this experiment each time one mutation operator was omitted. Average mutation scores obtained while omitting mutants generated by one of operators are given in Tab. 6 (column MS2 in [%]). Omitting one mutation operator gives in many cases similar results in comparison to all operators (row None, i.e. none operator omitted).

Quality measure *EQ2* has value close to 100% when an operator attributes to the MS (is a selective one), while values close to 0% when the operator could be omitted. On the contrary to the previous experiment, we look for operators that should not be excluded from the analysis, because it causes observable lowering of a mutation score The following operators give *EQ2* above 15% for at least one program and could stay: 7 OO operators PRV, OAO, JTD, JID, EOC, IPC, IOP, and 3 standard ones UOI, ROR, LCR. It can be seen that the result partially contradicts the previous experiment.

Table 5. Average results for excluding the most popular mutation operators: mutation score (MS1) and quality metric (EQ1)

	1. Enterprise Logging				2. Castle				3. Mono Gendarme			
	MS1 [%]		EQ1 [%]		MS1 [%]		EQ1 [%]		MS1 [%]		EQ1 [%]	
	OO	St	OO	St	OO	St	OO	St	OO	St	OO	St
0	62	71	86	91	66	70	71	84	56	74	87	61
1	57	65	100	100	56	65	86	93	45	67	100	99
2	48	55	91	94	52	63	92	100	34	59	89	100
3	43	48	85	89	51	58	100	97	24	33	70	57
4	39	24	94	60	45	37	91	75	20	24	65	39
5	38	22	99	59	41	28	91	65	18	20	56	32
6	35	13	96	48	35	24	78	61	12	8	33	2
7	33	0	94	0	30	0	67	0	9	7	25	0
8	33		98		29		66		9		25	
9	25		74		25		56		4		5	
10	23		64		24		53		3		0	
11	9		2		17		34		0		0	
12	9		3		15		26		0		0	
13	7		0		8		4		0		0	
14	0		0		7		0		0		0	

3rd Experiment on Selective Mutation - Mutation Operator Quality

This experiment evaluates a quality of each implemented mutation operator. In this case quality metric *EQ3* of a low value (close to zero) denotes an operator that could be omitted. The operator generates some mutants that are redundant (i.e. tests that kill mutants of other operators are also able to kill those mutants). In the contrast to the previous metric *EQ2,* this metric is less sensitive to the number of generated mutants.

However, it should be noted that this qualification does not take into account the ability to generate equivalent mutants. For example ABS operator has *EQ3* equal to 100% (is selective and generated necessary mutants), but also generated many equivalent mutants that were distinguished and removed during the preliminary analysis (Sec. 4.2). One of the following operators could be selected: standard ABS, LCR, UOI, ROR, UOR for *EQ3* > 25% and object-oriented EHR, EOC, EXS, OAO, IPC, JTD, PRV for *EQ3* > 50%. The obtained values are in many cases different for various programs, which is observable especially for the object oriented operators.

Comparison of Experiment Results. The results based on the above experiments are summarized in Table 7. Program identifiers and types of mutation operators (standard or object-oriented) are denoted in the first column. Number of mutants, number of tests and mutation scores calculated in four cases are compared. In the first, reference case, mutants are generated for all considered operators and all tests are used (columns All). Remaining results (columns Ex1, 2, 3) refer to cases decided on the basis of the above experiments and their quality metrics. If we exclude the most popular operators selected in experiment 1st, or use operators chosen in experiment 2nd,

or finally select operators according to *EQ3*, then we would obtain less accurate mutation score but also use fewer mutants and fewer tests, as given in columns Ex1-3, accordingly. The tradeoff of the accuracy and efficiency is visible, but in general the best results are in the last two cases (Ex2, Ex3), i.e. mutation scores close to the maximal ones for the significantly lower numbers of mutants and tests.

Table 6. Results for omitting one mutation operator (MS and quality metrics in [%])

Omitted	1. Enterprise Logging			2. Castle			3. Mono Gendarme		
operator	MS2	EQ2	EQ3	MS2	EQ2	EQ3	MS2	EQ2	EQ3
ABS	70.7	3	87	69.6	0	0	73.9	0	0.7
AOR	70.9	8	12	69.6	1	15	73.9	2	6
ASR	70.7	5	13	69.4	2	12	73.8	2	0.4
LCR	70.7	2	18	68.8	19	54	71.7	30	100
LOR	70.9	0	0	69.6	0	0	73.9	1	0
ROR	70.3	12	19	68.9	24	35	73.3	13	31
UOI	64.4	100	100	65.1	100	100	66.8	100	92
UOR	70.9	0	0	69.4	6	26	73.7	5	20
None	70.9			69.6			73.9		
DMC	61.8	0	0	65.8	0	0	55.9	0	0
EHR	61.3	5	57	65.7	1	34	55.4	4	69
EOA	61.8	0	0	65.8	1	0.2	55.9	0	1
EOC	58.7	50	89	55.0	100	100	44.1	100	100
EXS	61.7	3	85	65.7	3	67	55.9	0	0
IHD	61.8	0	0	65.8	0	0	55.9	0	0
IHI	61.8	0	0	65.8	0	0	55.9	0	0
IOD	61.7	2	6	65.8	0	0	55.9	2	1
IOK	61.7	11	12	65.8	1	1.3	55.9	2	1
IOP	61.8	0	0	65.8	1	41	54.6	16	47
IPC	60.1	17	30	63.6	17	78	55.9	0	0
ISK	61.3	11	21	65.8	1	1.5	55.9	1	5
JID	61.3	15	19	65.1	14	23	53.2	32	33
JTD	60.0	31	49	65.5	5	13	55.9	0	0
OAO	52.1	100	100	63.8	26	41	51.3	41	68
OMR	60.9	10	41	64.9	11	25	55.9	0	0
PRM	61.8	6	2	65.6	3	20	55.8	3	15
PRV	56.6	71	58	65.1	24	33	53.7	15	51
None	61.8			65.8			55.9		

4.4 Threats to Validity

Conclusion validity of the experiments is limited by a number of investigated programs. Three programs were not small, quite representative and of different origin, but may not reflect all programming tendencies in usage of new programming concepts of the C# language. Therefore, for example, no mutants for the DMC operator dealing with delegates were created, which is a specialized concept of C#.

Table 7. Mutation results and benefits for three experiments on mutation operator selection

Prog.	Mutation Score [%]				Number of mutants				Number of tests			
Oper.	All	Ex1	Ex2	Ex3	All	Ex1	Ex2	Ex3	All	Ex1	Ex2	Ex3
1 OO	61.8	42.5	57.1	59.7	903	363	710	711	1148	63	105	115
1 St	70.9	59.2	70.3	70.6	1623	578	1015	1103	1148	47	114	120
2 OO	65.8	52.0	61.6	62.4	1065	478	887	795	642	81	132	135
2 St	69.6	58.0	68.3	69.2	2316	403	1715	1950	642	60	124	143
3 OO	55.9	39.1	55.1	49.7	855	267	777	595	899	78	132	94
3 St	73.9	58.8	71.4	73.3	4074	643	3242	2987	899	157	273	319

Another factor influencing the reasoning behind the experiments is existence of equivalent mutants. The manual analysis significantly lowered this threat, but it cannot guarantee that all equivalent mutants were detected.

Construct validity concerns the quality metrics used for evaluation of experimental results. Their interpretation is in accordance to weight coefficients subjectively selected by the authors. However, they suggest only tendencies in usage of different operators. The data calculated for other coefficients (0.8, 0.1, 0.1) gave analogous results. The final results (Table 7.) are expressed in terms of strict measures, such as mutation score, number of mutants or number of tests.

In order to minimize a threat to external validity, the programs used in experiments were parts of big, commonly used projects. Though, all of them were open-source projects and might have slightly different features than the commercial ones.

5 Related Work

Research on object-oriented mutation operators was conducted on Java and C# programs. Previous experiments on object-oriented operators of C# [5-10] were summarized in [11]. It concerned 29 object-oriented and specialized mutation operators defined for C#, and indicated on the difficulty to generalize results of the object-oriented operators. One operator (e.g. PNC, JID) can generate many mutants for one program, but only few for another program.

Application of object-oriented mutation operators to Java was studied in series of experiments [15-18, 22,29,30] with MuJava [31] that implements the most comprehensive set of 28 object-oriented operators, and MuClipse [32] - the plug-in for Eclipse adopted from MuJava.

An overview of cost reduction techniques, including selective mutation, mutant sampling and clustering can be found in [1,19]. Selective mutation was studied for structural languages [12-14], giving a recommendation of five standard operators in [12] that were also applied in CREAM. Ten operators for C were selected in empirical studies performed using the Proteum tool [13]. Comparison between two approaches: operator-based mutant selection and random mutant selection did not confirm a superiority of the first one [15], but this result only referred to standard operators.

Selectiveness of operators was also investigated for Java programs [16,17]. General conclusions were similar to those of C#. Object-oriented mutants are killed by a lower number of tests than standard mutants, but a significant decrease in the

number of mutation operators is no so visible as for standard operators. In [17] it was recommended not to use OAC, PCI, and one of EAM, EMM operators, but those operators were not selected for implementation in CREAM.

Apart from CREAM, only the ILMutator prototype implements 10 object-oriented and C# specific mutation operators [10]. It introduces mutations into Intermediate Language of .NET for programs originated from C#, hence is more time effective in generating mutants than CREAM. Other mutation tools for C# like Nester [33] and PexMutator [34], do not support any object-oriented operators.

A problem of code converge in relation to the ability of killing mutants was addressed in the work of [35]. Uncovered mutants were manually identified as a type of equivalent mutants, whereas CREAM can automatically decide not to generate an uncovered mutant, if appropriate coverage data are provided. In experiments reported in [32] sufficiently high statement coverage (80%) was an initial requirement, but did not directly influence mutant generation. In the contrast to other tools PexMutator is aimed at creating test cases in order to kill mutants. It extends the Pex tool that creates tests in order to obtain the high code coverage.

6 Conclusions

The main lessons learned after investigation of three different open-source C# projects are the following.

The simplest and most beneficial way of mutation cost reduction is introducing mutation only to a covered code. Using tests that cover on average 82% of the code 75% of all mutants were generated giving no loss in the mutation score (MS) accuracy. The conclusion is obvious, but it practically means that coverage and mutation tools should be combined. In CREAM the mutants can be generated for the whole code, or for the fragments covered by a given test set. It is especially important for a project under development, when only parts of the code are currently examined. Though, the cost will be not reduced if the code is covered in 100%.

The remaining results gave no explicit conclusion about the general superiority of one applied cost reduction technique over the other ones. However, it was possible to obtain a small decline of a mutation score (99% of MS_{orig} for standard mutation and 93% for object-oriented) with a significant gain in lowering mutation costs: the number of mutants (81% of standard mutants used, 74% OO mutants) and the number of tests (22% for standard and 14% for OO). We proposed metrics to evaluate a tradeoff between these factors. They could also be used to compare tradeoffs in other mutant selection experiments and adjusted to higher order mutation [18].

In all investigated approaches to mutation operator selection and mutant sampling the MS accuracy of object-oriented mutants was worse (from few to 10%) than the corresponding accuracy of standard mutants. We stated that omitting a selected mutation operator was more beneficial than excluding mutants generated by the most popular operators (about 10% better MS). However, similarly to the previous studies on the object-oriented mutation the detailed results depend on the programs under concern. Especially programs using specialized programming constructs can give different results (e.g. DMC - operator of delegates was not used in these programs).

The same approach to quality evolution was also applied to mutant sampling. The best quality tradeoff was obtained when 35% of mutants where randomly selected for each class giving about 85% of the original MS for the object-oriented mutation and

using 10% of tests. In the case of standard operators, 30% of mutants selected for each operator and 15% of tests resulted in 93% of MS_{orig}. In general, the random mutant sampling allows to obtain significant reduction in number of mutants and tests, but the loss of MS accuracy was a bit higher than in the operator selection experiments.

The detailed results of mutant clustering are also behind the scope of this paper. However, in general we obtained 97% of the original MS for object-oriented mutants using 32% of mutants and 17% of tests. Whereas for standard mutations, it was only 91% MS for 19% of mutants and 15% of tests. These results were less promising than the results of clustering for standard mutation in C [21]. Moreover, experiences of clustering are difficult to reproduce to other projects.

The experiments on the cost reduction techniques can be performed on other kinds of C# projects using the wizard built-in the CREAM mutation testing tool.

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)
2. Chevalley, P.: Applying Mutation Analysis for Object-Oriented Programs Using a Reactive Approach. In: Proc. of the 8th Asia-Pacific Software Engineering Conference, ASPEC, pp. 267–270 (2001)
3. Kim, S., Clark, J., McDermid J.A.: Class Mutation: Mutation Testing for Object-Oriented Programs. In: Conference on Object-Oriented Software Systems, Erfurt, Germany (2000)
4. Ma, Y.-S., Kwon, Y.-R., Offutt, J.: Inter-class Mutation Operators for Java. In: Proc. of International Symposium on Software Reliability Engineering, ISSRE 2002. IEEE Computer Soc. (2002)
5. 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)
6. Derezińska, A.: Quality Assessment of Mutation Operators Dedicated for C# Programs. In: Proc. of the 6th International Conference on Quality Software, QSIC 2006, pp. 227–234. IEEE Soc. Press (2006)
7. Derezińska, A., Szustek, A.: Tool-supported Mutation Approach for Verification of C# Programs. In: Zamojski, W., et al. (eds.) Proc. of International Conference on Dependability of Computer Systems, DepCoS-RELCOMEX 2008, pp. 261–268. IEEE Comp. Soc. (2008)
8. 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)
9. Derezińska, A., Sarba, K.: Distributed Environment Integrating Tools for Software Testing. In: Elleithy, K. (ed.) Advanced Techniques in Computing Sciences and Software Engineering, pp. 545–550. Springer, Dordrecht (2009)
10. 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, 6th Workshop on Mutation Analysis, pp. 342–350. IEEE Comp. Soc. (2011)
11. Derezińska, A.: Classification of Operators of C# Language. In: Borzemski, L., et al. (eds.) Information Systems Architecture and Technology, New Developments in Web-Age Information Systems, pp. 261–271. Wrocław University of Technology (2010)

12. Offut, J., Rothermel, G., Zapf, C.: An Experimental Evaluation of Selective Mutation. In: Proc. of 15th International Conference on Software Engineering, pp. 100–107. IEEE Comp. Soc. Press (1993)
13. Barbosa, E.F., Maldonado, J.C., Vincenzi, A.M.R.: Toward the Determination of Sufficient Mutant Operators for C. Journal Software, Testing, Verification, and Reliability 11, 113–136 (2001)
14. Namin, S., Andrews, J.H.: On Sufficiency of Mutants. In: Proc. of 29th International Conference on Software Engineering, ICSE 2007 (2007)
15. 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)
16. Ma, Y.-S., Kwon, Y.-R., Kim, S.-W.: Statistical Investigation on Class Mutation Operators. ETRI Journal 31(2), 140–150 (2009)
17. Hu, J., Li, N., Offutt, J.: An Analysis of OO Mutation Operators. In: Proc. of 4th International Conference Software Testing Verification and Validation Workshops, 6th Workshop on Mutation Analysis, pp. 334–341. IEEE Comp. Soc. (2011)
18. Kaminski, G., Praphamontripong, U., Ammann, P., Offutt, J.: A Logic Mutation Approach to Selective Mutation for Programs and Queries. Information and Software Technology, 1137–1152 (2011)
19. Usaola, M.P., Mateo, P.R.: Mutation Testing Cost Reduction Techniques: a Survey. IEEE Software 27(3), 80–86 (2010)
20. Mathur, A.P., Wong, W.E.: Reducing the Cost of Mutation Testing: An Empirical Study. Journal of Systems and Software 31, 185–196 (1995)
21. Hussain, S.: Mutation Clustering. Ms. Th., King's College London, Strand, London (2008)
22. Ji, C., Chen, Z.Y., Xu, B.W., Zhao, Z.: A Novel Method of Mutation Clustering Based on Domain Analysis. In: Proc. of 21st International Conference on Software Engineering & Knowledge Engineering, SEKE 2009, pp.422–425 (2009)
23. CREAM, http://galera.ii.pw.edu.pl/~adr/CREAM/
24. NUnit, http://www.nunit.org
25. NCover, http://www.ncover.com
26. Subversion svn, http://subversion.tigris.org
27. Derezińska, A., Rudnik, M.: Empirical Evaluation of Cost Reduction Techniques of Mutation Testing for C# Programs, Warsaw Univ. of Tech., Inst. of Computer Science Res. Rap. 1/2012 (2012)
28. Source Monitor, http://www.campwoodsw.com/sourcemonitor.html
29. Lee, H.-J., Ma, Y.-S., Kwon, Y.-R.: Empirical Evaluation of Orthogonality of Class Mutation Operators. In: Proc. of 11th Asia-Pacific Software Engineering Conference. IEEE Comp. Soc. (2004)
30. Ma, Y.-S., Harrold, M.J., Kwon, Y.-R.: Evaluation of Mutation Testing for Object-Oriented Programs. In: Proc. of 28th International Conference on Software Engineering, pp 869–872. IEEE Comp. Soc. Press (2006)
31. Ma, Y.-S., Offutt, J., Kwon, Y.-R.: MuJava: an Automated Class Mutation System. Software Testing, Verification and Reliability 15(2) (June 2005)
32. Smith, B.H., Williams, L.: A Empirical Evaluation of the MuJava Mutation Operators. In: Proc. 3rd International Workshop on Mutation Analysis Mutation 2007 at TAIC.Part 2007, Cumberland Lodge, Windsor UK, pp. 193–202 (September 2007)
33. Nester, http://nester.sourceforge.net/
34. Pexmutator, http://www.pexase.codeplex.com
35. Segura, S., Hierons, R.M., Benavides, D., Ruiz-Cortes, A.: Mutation Testing on an Object-oriented Framework: an Experience Report. Information and Software Technology, 1124–1136 (2011)