Key words:
mutation testing, mutation operators,
object-oriented mutation, C#

Anna DEREZIŃSKA*

CLASSIFICATION OF ADVANCED MUTATION OPERATORS OF C# LANGUAGE

Mutation testing can be used to evaluate the quality of test suites and support generating of test cases. Faults injected in mutation testing are defined by mutation operators. Mutation testing is a laborious approach; therefore selection of good mutation operators is of high importance. In this chapter the experimental and analytical investigation on object-oriented and other advanced mutation operators applied for C# programs are summarized. According to experimental results, object-oriented operators have a real added value in comparison to traditional mutation operators. A classification of about thirty advanced mutation operators is presented, according to an expected number of generated mutants, prediction about ability to being killed and tendency to create non-equivalent mutants. The judgment about the mutant equivalence is very costly and such mutants cannot be used for assessment of tests. Operators that create many mutants killed easily by any test operators are also not useful in qualification of tests. The paper introduces three metrics based on the number of generated mutants, killed mutants and equivalent mutants, to classify mutation operators. Using the metrics the operators are classified into general categories. The presented results could be used in the further improvement of mutation tools for C# language and selecting sufficient set of mutation operators.

1. INTRODUCTION

Software testing is indispensable in improving some attributes of software quality. Moreover, software tests and testing procedures should meet high quality requirements. Mutation testing helps to complete the later task. It is used to determine the effectiveness of test cases i.e. how good they are in detecting faults.

---

* Institute of Computer Science, Warsaw University of Technology, Nowowiejska 15/19, 00-665 Warsaw, Poland.

This work was supported by the Polish Ministry of Science and Higher Education under grant 4297/B/T02/2007/33.

A mutated program (*mutant*) is a program that has been purposely altered from its original version. These program changes, i.e. kinds of injected faults, are specified by *mutation operators* in order to automate the mutant generation process. A mutant that run against a test case is *killed* if the test detects a difference in its behavior to one of the original program. Two programs of the identical behavior are called *equivalent*.

So-called standard (or traditional) mutation operators define small, basic changes that are possible in typical expressions or assignments of a general purpose language [24]. Testing object-oriented programs requires taking into account object-oriented features and other programming characteristics that might be misused during software development. Therefore for object-oriented languages sets of adequate mutation operators were proposed (Java [10, 13], C# [1]).

Mutation testing is known to be very labor consuming. Hence, prediction about usefulness of the operators is worthwhile. The result presented in this chapter was based on many experiments performed using developed mutation tools [4,5,6] or simple scripts supporting mutant generations [1,2]. Some categorization concerning equivalent mutants was also based on an analytical examination of a potential mutated code.

The chapter discuses three measures, the number of generated mutants, killed mutants and equivalent mutants, to classify OO mutation operators. They can be used to compare mutation operators, as it takes both the cost (number of mutants and equivalent mutant) and the value (how hard to kill) factors into consideration. Such measure can be straightforward calculated from an experiment on a given program. A problem is answering a question whether it can be generalize into a coarse categories for many various programs in order to catch a general tendency for selection of operators to be used in further experiments.

2. CHALLENGES IN C# MUTATION

Object-oriented operators were primarily proposed for Java programs [10,13]. Adapting object-oriented operators of Java and extending with new language features, an advanced set of about 40 mutation operators of C# was proposed and partially evaluated in some experiments [1,2,4-6].

The evolution of the C# language brings still new programming's constructs that could be used. Examination of new features emerging in versions 2.0 and 3.0 of C# (VS 2005, 2008) gave mostly discouraging results for creating mutation operators based on the most of these features, with exception of Language Integrated Query [3].

Practical usage of the mutation approach requires a tool support. A comprehensive summary of mutation ideas and developed tools can be found in [9]. Object-oriented mutation of Java programs is supported by MuJava tool [14] and Muclipse[22] – a plugin for MuJava. The Java object-oriented operators were evaluated in several experiments [11,12,14,22,23].

Traditional mutation operators working on C# programs were supported in Nester tool [18], and five operators (sufficient according to Offutt) has been recently incorporated into PexMutator [21].

The first tool supporting selected object-oriented mutations for C# was the CREAM system [5]. It is a parser-based mutation tool. In the next version of the system some improvements in its performance were achieved [6].

Development of object-oriented mutations in the intermediate code derived from compiled C# programs is proposed in a prototype tool ILMutator [4]. It implements six selected operators. Modification of a program is performed through manipulation in its metadata and intermediate code , what enables omitting mutant recompilation.

Except of performance issues dealing with time of mutant generation and mutant testing, a crucial problem remains recognition of equivalent mutants. Determination about program equivalence is a generally undecidable problem [20].

However, non-killed mutants can be examined in order to detect at least a subset of equivalent mutants. Possible partial solutions are based, for example, on constraint solving [20] or program slicing [24]. It was experimentally showed [8] that the higher difference in execution of an original and mutated program measured in terms of the code coverage the less likely to be an equivalent mutant.

Generation of equivalent mutants can also be partially avoided in a tool. Restrictive mutant generation was done in CREAM, although many equivalent mutants can be still generated.

3. COMPARISON OF ADVANCED MUTATION OPERATORS OF C#

An important issue is determining a characteristic of mutation operators in order to choose the best ones to be applied. A "good" operator should generate valid and non-equivalent mutants, and be effective in qualification of tests. A desired feature of an operator is the ability to mimic typical errors of program developers. Although, an operator can also be designed in order to enforce higher coverage of a testing program, especially covering hardly accessible program branches.

Application of certain mutation operators can often provide many equivalent mutants. The judgment about the equivalence is very costly and such mutants cannot be used for assessment of tests. Some mutation operators can generate mutants that are killed very easily by any test. Such operators are not useful in qualification of tests, although they can mimic typical errors of developers.

We would like to point at the usefulness of the object-oriented mutation operators. The preliminary results with these mutation operators compared to traditional ones do not confirm the superiority of simple mutation over the more complex. We got for example 28% mutation score for faults injected by object oriented mutation operators (12 operators used by CREAM v2) in comparison to 99% for selected traditional operators (AOR, COR, LOR, ROR) and the same test suite [6]. The tests detecting simple faults seem not to be sufficient to detect object-oriented programming flaws in C#.

In this section a classification of advanced mutation operators is discussed according to an expected number of generated mutants, prediction about ability to being killed and tendency to create non-equivalent mutants.

3.1. MUTATION METRICS

Let assume that a program *P* is tested with a test suite *T*. For a given set of mutation operators, the basic numbers characterizing a mutation process are:

$|M|$ – number of mutants generated for the program *P*,

$|K|$ - number of mutants killed by at least one test of the suite *T*,

$|E|$ – number of equivalent mutants among non-killed mutants,

where symbol |X| is a multiplicity of set X.

The main measure commonly used in mutation testing is a *mutation score*. It depicts the ability of the test suite to find faults on the program It is calculated as a ratio of the number of mutants killed over the number of all non-equivalent mutants.

$$MS(P,T) = \frac{|K|}{(|M|-|E|)} \qquad (1)$$

However, as we already stated, it can be difficult to identify an exact number of equivalent mutants when not all equivalent mutants are distinguished. Therefore, an approximate number E'≤E is often used in practice. When equivalent mutants are not determined, the mutation score is in fact substituted by its lower bound on mutation score, called in [15] as mutation score indicator (2).

$$MSI(P,T) = \frac{|K|}{|M|} \qquad (2)$$

Calculating mutation score it does not matter how many test cases killed a mutant. Test cases can have different effectiveness in detecting errors injected by mutations. One of possible effectiveness metrics was introduced in [2] and is calculated as a ratio of test cases that killed given mutants to all test runs performed on these mutants (3). Only non-equivalent mutants are taken into account.

$$EF(P,T) = \frac{\sum_{m \in M} | Km |}{(| M | - | E |) * | T |} \tag{3}$$

where K_m are test cases killing mutant m.

It can be also interpreted as a product of mutation score and an average number of test cases killing dead mutants divided by the number of test cases [7].

All above measures can be used for all mutation results, or individually for a subset of mutants generated, for example, by a given mutation operator. They can be applied after some mutation experiments with a given program, or accordingly with a set of programs and their suites. In all case the definite results (i.e. *M, K, E, T,* etc) are necessary to make any calculation.

The problem focused in this chapter is how to generalise results of the performed experiments in order to predict the characteristics of various mutation operators. Each assessment would be imprecise from a definition and a definite judgment can be not obtained in all cases. Nevertheless, in some cases hints and general trends can be anticipated. The approximation will be based, as typical for many software engineering estimations, on very coarse categories, like low, high, medium. In order to classify a mutation operator to a certain category, three metrics will be used.

The first metric deals with an expected number of mutants generated by an operator. It should be noted that a number of mutants can also be roughly assess using a static analysis of a program. For example, a number of usage of an arithmetic operator in expressions determines the number of mutants changing the operator. Similarly, some object-oriented complexity metrics (like depth of inheritance tree, number of children of a class), and other values charactering mechanisms used in a program (number of exception handling operations, number of delegates) can indicate at the order of magnitude of mutant numbers.

Moreover, very coarse categorization can be made without any source analysis using the less precise, but the simplest program metrics - LOC (number of code lines). Such approach was used because of primary interest in general trends of operators regardless on the applied programs.

Expected number of generated mutants *GM* can be calculated in the following way:

$$GM(P,T) = \frac{| M | * C1}{LOC * C2} \tag{4}$$

Value of *GM* can be interpreted as follows:

$GM > 1$ implies that a given operator generates a high number of mutants,

$0.2 \leq GM < 1$ denotes a medium number of mutants,

and $GM < 0.2$ a low one.

Assuming some values of coefficients (C1=10000, C2=50), the following numbers of mutants were understood as distinguishing criteria:

High - more than 50 mutants for ten thousand code lines (on average),

Medium - between 10 and 49 mutants for ten thousand code lines,

Low - below 10 mutants for ten thousand code lines.

The next metric predicts a number of killed mutants over all mutants generated for a given operator. It is calculated in the same way as mutation score indicator (3). But here we do not discuss exact results of a particular experiment with a given test suite. Instead, the averaged values could be used for the categorization of mutation operators. The value of *MSI* can be interpreted in the following way:

$MSI \geq 0.5$ implies a *High* number of killed mutants of a given operator,

$0.2 < MSI < 0.5$ denotes a *Medium* number of killed mutants,

$MSI \leq 0.2$ a *Low* number.

It should be noted, that category "High" is a very broad one, including cases when for example 60% of mutants are killed and 90% as well. This is because such a general category should be a feature of a discussed operator. More precise results given by the mutation score and, for example close to 100%, can be a characteristic of a particular test suite analyzed for a given program. We also neglect in this calculation possibility of existence of equivalent mutants. It is also because it is aimed at rough categorization. But if an exact mutation score is available it can be considered.

The third measure estimates a number of equivalent mutants that could be generated. Two categories are provided *Low* and *High*. The high number category is assigned according to an analytical examination of an operator and recognition of equivalent mutants. The second possible criterion is an experimental result that refers to a high number (more than 75%) of not killed mutants whereas a test suite could not be easily improved.

3.2. CLASSISFICATION OF OPERATORS

Evaluation of applicability of object-oriented operators in C# programs is summarized in Tab. 1. The table contains a subset of mutation operators adopted or proposed for C# [1]. The operators are identified by their abbreviations. The names of the operators can be found in the appendix in Tab. 2.

The presented results are conclusions of operators analysis and experiments performed on mutants generated by dedicated scripts, or manual injection [1,2] or using tools: CREAM [5,6] and ILMutator [4]. Experiments performed with about 13 thousand of mutants were taken into account. In the experiments we used, apart from the programs listed in the above mentioned publications, the following programs: a library for mathematical calculations Math.Net.Iridium [16], classes Money and MoneyBag - examples from the NUnit framework [19] and MiscUtil project [17].

In Tab. 1 three columns of expected number of generated mutants correspond to Low, Medium and High category of *GM*. The next three columns recapitulates estimation of predicted number of killed mutants. Two last columns give an approximation about generating equivalent mutants by an operator.

Not all operators are marked for each measure. Expected number of killed mutants is not assessed for operators IHD, IHI, IOR, PRM because no definite tendency for different programs was found. In the cases of two operators (PNC and JID) numbers of generated mutants have so diverse measures in different experiments, that two categories with a question mark are indicated.

The less precise estimation is given for the last measure of expected number of equivalent mutants, where only two categories are marked. The classification is left empty for operators IHI, IHD, IOD, IOR, ISK, PMD, JTD, JID, JDC, EOC, EHR, DMC, DEH, PRM and OPD, when an estimation is not adequate to any of them, i.e. could be a medium one or there were no enough evidence to state a general judgment.

Several object-oriented operators could be compared to those of Java. Some results are consistent with tendencies shown for C#, as for example a high number of mutants for PRV, low for IOR [23], also low for IHD and ISK [14]. Other results might be inconsistent as a low number of mutants for EOC [14], or others in [23]. In general, the numbers of object-oriented mutants were to low in [11,23], or even equal zero [22], to be statistically significant and to allow generalization and comparison with above estimation. Other mutation operators, as dealing with delegates of C# (DMC, DMO, DEH), properties (PRM) or exceptions (EHR) had no Java corresponding results to be compared.

Tab. 2. Classification of object-oriented mutation operators of C#

Operator		Expected number of mutants							
		generated			killed			equivalent	
		Low	Medium	High	Low	Medium	High	Low	High
1	AMC			x	x			x	
2	IHD	x							
3	IHI	x							
4	IOD		x			x			
5	IOP	x			x				x
6	IOR	x							
7	ISK	x					x	x	
8	IPC		x		x				x
9	PNC	x?	x?		x				x
10	PMD	x			x				
11	PPD			x	x				x
12	PRV			x	x				x
13	OMR			x		x		x	
14	OMD			x	x				
15	OAO	x			x				

16	OAN	x			x				
17	JTI	x			x				x
18	JTD			x		x			
19	JID		x?	x?		x			
20	JDC	x				x		x	
21	EOA		x		x				x
22	EOC			x		x			
23	EHR	x			x				
24	DMC	x					x	x	
25	DMO	x			x				x
26	DEH	x					x	x	
27	PRM			x					
28	IOK		x			x			x
29	OPD			x	x			x	

Although we focused on result generalization, a number of potential mutants depends strongly on a kind of a program. The application of some operators is more or less probable in dependence to the program type, application domain, programming habits, etc. For example, the operators dealing with delegates will create scarcely or even any mutants if the delegate mechanism is not used in a program. A program with a limited exception handling produces less exception-based mutants than a program with an exception handling. The same observations are for operators dealing with class hierarchy, usage of certain keywords like *override*, etc. Using such operators we look for the specific faults. It can be viewed as a natural selection and has also some advantages. In the natural way, the set of generated mutants and further scope of test investigation to the considered program are adjusted.

There are several threats to validity faced by the analysis. Results of many various programs, originating in various sources, written by different programmers, were concerned in order to deal with a threat of mono-operation bias. A threat to statistical conclusion validity could be alleviated by the high number of mutants and test cases. The most serious threat to external validity concerns a quite arbitrary selection of coefficients and criteria thresholds in operators' classification. They were tuned according to available results, but nevertheless it should be treated as an exemplary proposal and not steadily fixed values. Another threat is a possibly of ambiguous results, therefore classification of some categories (e.g. as in column of equivalence) were not provided.

4. CONCLUSIONS

In the chapter a methodology of a classification of object-oriented mutation operators and its application to operators of C# programs was discussed. It can be beneficial in selection of mutation operators that are implemented during development of C# mutation tools (CREAM, ILMutator) and in planning of experiments. Usage of diverse object-oriented mutation operators also points at a natural selection of mechanisms that were applied in a program under test and should be covered by its tests.

Based on the developed tools further experiments on quality of advanced mutation operators of C# are planed. They are going to deal with mutant clustering approaches for selection of mutants subset as well as higher order mutation testing.

Acknowledgments The author would like to thank M. Rudnik for his cooperation in evaluation of mutation operators.

APPENDIX

Tab. 2. Object-oriented mutation operators of C# - subset [1]

1	AMC	Access modifier change
2	IHD	Hiding variable deletion
3	IHI	Hiding variable insertion
4	IOD	Overriding method deletion
5	IOP	Overridden method calling position change
6	IOR	Overridden method rename
7	ISK	Base keyword deletion
8	IPC	Explicit call of a parent's constructor deletion
9	PNC	New method call with child class type
10	PMD	Member variable declaration with parent class type
11	PPD	Parameter variable declaration with child class type
12	PRV	Reference assignment with other compatible type
13	OMR	Overloading method contents change
14	OMD	Overloading method deletion
15	OAO	Argument order change
16	OAN	Argument number change
17	JTI	This keyword insertion
18	JTD	This keyword deletion
19	JID	Member variable initialization deletion
20	JDC	C# supported default constructor create
21	EOA	Reference assignment and content assignment replacement
22	EOC	Reference comparison and content comparison replacement
23	EHR	Exception handler removal
24	DMC	Delegated method change
25	DMO	Delegated method order change
26	DEH	Method delegated for event handling change
27	PRM	Property replacement with member field
28	IOK	Override keyword substitution
29	OPD	Overriding property deletion

REFERENCES

[1] DEREZIŃSKA A., *Advanced mutation operators applicable in C# programs*, In: K. Sacha (ed.) IFIP Vol. 227, Software Engin. Techniques: Design for Quality, Springer, Boston, 2006, 283-288.

[2] DEREZIŃSKA A, *Quality assessment of mutation operators dedicated for C# programs*, In: 6th Inter. Conf. on Quality Soft., Beijing, China, IEEE Comp. Soc. Press, California, 2006, 227-234.

[3] DEREZIŃSKA A., *Analysis of emerging features of C# language towards mutation testing.* In: Models and methodology of system dependability, J. Mazurkiewicz, at allski (eds.), Monographs of system dependability, Vol. 1, Wrocław, Publish. Wrocław Univ. of Technology, 2010, 47-59.

[4] DEREZIŃSKA. A., KOWALSKI K., *Advances in mutation testing of C# programs*, ICS Research Report 4/2010, Institute of Computer Science, Warsaw University of Technology, Warsaw, 2010.

[5] DEREZIŃSKA A., SZUSTEK A., *Tool-supported mutation approach for verification of C# programs*. In: W. Zamojski, J. Mazurkiewicz, J. Sugier, T. Walkowiak (eds.) Inter. Conf. on Dependability of Computer Systems (DepCoS-RELCOMEX 2008), IEEE Computer Soc., 2008, 261-268.

[6] DEREZIŃSKA A., SZUSTEK A., *Object-oriented testing capabilities and performance evaluation of the C# mutation system.* In: preprint of Proc. 4th IFIP TC2 Central and Eastern European Conf. on Software Engineering Techniques CEE-SET'09, Krakow, Poland, 12-14 October 2009, 270-283.

[7] ESTERO A., PALOMO F, MEDINA I., Quantitative evaluation of mutation operators for WS-BPL compositions, In: Proc. of Workshop on Mutation Analysis, Muatation'10, April 2010.

[8] GRUN B. J. .M., SCHULER D., ZELLER A., *The Impact of Equivalent Mutants*. In: 4th Inter. Workshop on Mutation Analysis, Denver, Colorado, IEEE Comp. Soc. 1-4 Apr. 2009, 192-199.

[9] JIA Y., HARMAN M., *An analysis and survey of the development of mutation testing*, IEEE Transactions of Software Engineering, 2010 (to appear), http://www.dcs.kcl.ac.uk/pg/jiayue/repository/

[10] KIM S., CLARK J., McDERMID J. A., *Class Mutation: mutation testing for object-oriented programs*. In: Proc. of Net.ObjectDays Conf. on Object-Oriented Soft. Systems, Erfurt, Germany, 2000.

[11] LEE H-J., MA Y-S, KWON Y-R, *Empirical evaluation of orthogonality of class mutation operators*, In: 11th Asia-Pacific Software Engineering. Conf, IEEE Comp. Soc. 2004.

[12] MA Y.-S., HARROLD M. J., KWON Y.-R., *Evaluation of Mutation Testing for Object-Oriented Programs*. In: Proc. of the 28th Int. Conf. on Soft. Eng. Shanghai, China, 20-28 May 2006, 869-872.

[13] MA Y-S., KWON Y-R., OFFUT A.J., *Inter-class mutation operators for Java*. In: Proc. of 13-th Inter. Symp. on Software Reliability Engineering, ISSRE'02, IEEE Computer Soc., 2002, 352-363

[14] MA Y-S., OFFUTT A. J., KWON Y-R., *MuJava: an automated class mutation system*, Software Testing, Verification & Reliability, Vol. 15, No 2, June 2005, 97-133.

[15] MADEYSKI L., On the effects of pair programming on thoroughness and fault-finding effectiveness of unit tests. In: Munch, J., Abrahamsson, P.J. (eds.) Profes 2007. LNCS, vol. 4589, Springer, Heidelberg .2007, pp. 207-221.

[16] Math.Net.Iridium, htp://mathnet. opensourcedotnet.info/

[17] MiscUtil, http://www.yoda.arachsys.com/csharp/miscutil/

[18] Nester, http://nester.sourceforge.net/

[19] NUnit, http//www.nunit.org

[20] OFFUTT A. J., PAN J., *Automatically detecting equivalent mutants and infeasible paths,* Software Testing, Verification and reliability, Vol. 7, No. 3, 1997, 165-192.

[21] Pexmutator : C# code mutator, http://www.pexase.codeplex.com

[22] SMITH B. H., WILLIAMS L., *Should software testers use mutation analysis to augment a test set?* Journal of Systems and Software, No 82, 2009, 1819-1832.

[23] UMAR M., *An evaluation of mutation operators for equivalent mutants,* Msc Thesis Dep. of Comp. science, King's College, London, 2006.

[24] VOAS J.M., McGRAW G., *Software Fault Injection, Inoculating Programs Against Errors*, John Wiley & Sons Inc., 1998.