Object-Oriented Testing Capabilities and Performance Evaluation of the C# Mutation System*

Anna Derezińska and Anna Szustek

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

Abstract. The main purpose of mutation testing approach is to check a test suite quality in terms of the adequacy to killing programs with inserted programming faults. We present advances in the C# mutation testing system that supports object-oriented mutation operators. The system enhancements related to functional requirements (mutation operators, avoiding generation of invalid and partially of equivalent mutants) and non-functional ones (speed-up using a new parser and reflection, space reduction storing mutant updates). Mutation testing of six widely used open source programs is discussed. The quality of the tests supplied with these programs was experimentally determined. Performance measures were evaluated to assess system enhancements (2-4 faster mutants creation, 10-100 times disk space reduction, tradeoff of time overhead for storing mutants of different size in a local or remote repository).
Keywords: mutation testing, object-oriented mutation operators, C#, system evolution

1 Introduction

Software testing is a critical part of software development. Mutation is the technique of generating faulty variants of a program [1]. It can be applied for assessing the fault detection ability of a test suite or comparing testing strategies. According to the experimental research [2], generated mutants can be treated as representatives of real faults. Mutation analysis can be performed "whenever we use well defined rules defined on syntactic descriptions to make systematic changes to the syntax or to objects developed from the syntax" [3].

A transformation rule that generates a *mutant* from the original program is called a *mutation operator*. So-called standard (or traditional) mutation operators introduce small, basic changes that are possible in typical expressions or assignment statements of any general purpose language [1]. Mutation operators dealing with many specific programming features, including object-oriented ones, were also developed. If for any program P the result of running its mutant P' is different from the result of running P for any test case of the test suite T, then the mutant P' is said to be *killed*. A mutant that cannot be killed by any possible test suite is counted as an *equivalent* one. The

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

adequacy level of the test set T can be computed in terms of the number of mutants killed by T.

Advanced mutation operators of C# language were adopted similarly to object-oriented operators of Java language [4,5,6,7] or developed for the specific language features and studied in experiments [8,9]. Based on the gathered experiences about evaluation and specification of object-oriented mutation operators, the first CREAM (CREAtor of Mutants) system supporting few of them for C# programs was implemented. Prior works [10,11] have presented the general architecture overview of the system and results of the first experiments. Suffered from several limitations, the system evolved to CREAM2, enhancing its mutant generation capabilities and performance issues.

The crucial problem in mutation testing approaches are considerable time and resource constraints. Due to lack of mature mutation tool support, also very little empirical evaluation has been done in the industrial development of C# programs. In this paper, we discuss advances in the CREAM2 system, including code parsing improvements, preventing generation of invalid and partially of equivalent mutants, cooperation with the distributed Tester environment [12], storing mutant's updates in the SVN repository [13]. We conducted experiments with several open source programs to evaluate and verify an extended set of mutation operators, to evaluate quality of test suites, and compare the results with the previous mutation system and standard mutations.

The remaining paper is organized as follows. In the next Section related work is discussed. Section 3 contains the outline of the improvements of the CREAM2 system. Section 4 presents an experimental study on object-oriented mutation of C# programs. We finish with assessment of performance results and conclusions.

2 Related Work

Mutation testing approach was developed primarily for structural languages, like Fortran with the mostly known Mothra tool [1]. Further, similar standard mutation operators were applied for other languages, as C language: Proteum [14,15], MiLu [16], and also object-oriented languages Java, C++, C#.

Standard mutations introduced into Java Byte Code are supported in different tools, like Jumble [17], MuGamma [18]. Selected traditional and some object-oriented mutations in Java were also implemented in Judy [19] and Javalanche [20] tools. Test cases considered for Java programs were commonly unit tests suitable for JUnit environment [17,19,21,22], or similar but specialized like in MuJava [23]. Several testing systems for C++ language uses standard mutation testing also in commercial products, as Insure++ from Parasoft. Simple changes to Java source code, without parser involvement, were implemented also in Jester environment [21]. The ideas of Jester system were transformed to Python and C# languages. Nester tool [24] supports the standard mutations of C# language. The improved version of Nester makes only one compilation run for all mutants. Afterwards, it is decided during test execution which mutant should run.

Mutation testing of object-oriented program features (mostly inter-class relations) was exhaustively studied for Java programs [4,5,6,7,23,25,26]. Standard and object-oriented operators for Java were applied in MuJava [23] and MuClipse [26] - an Eclipse plug-in based on MuJava. In MuJava, mutants are generated either as a parameterized

program (so-called meta-mutant) that is further recompiled and executed as a mutant according to a given parameter, or a mutation is introduced directly in the Byte Code.

The first tool supporting selected object-oriented mutations for C# was the CREAM system [10,11]. The Nmutator tool announced in 2002 was supposed to introduce object-oriented mutations into C# programs, but there are no evidences that this intention was fulfilled. Research in [27] mentioned object-oriented features of C# but concentrated on other problems and does not develop any tool at this area. Besides the CREAM system the only known system dealing with object-oriented mutation operators for C# programs is the ILMutator prototype [28]. It provides object-oriented mutations into the intermediate code derived from compiled C# programs. It implements so far six types of changes of intermediate code corresponding to object-oriented mutation operators on the C# source code level. Further development of the tool is in progress.

There are still many challenges that need to be solved in order to effectively bring mutation testing into industrial practice. Representativeness of mutation faults in comparison to real faults was studied in case of standard mutation operators and C programs [2]. Experiments gave promising results, but analogous facts for mutation faults injected into other languages and using more specialized operators have been not yet sufficiently verified [22].

Generation and execution of many mutants is generally very expensive. Time and space performance can be limited by selection of the most relevant mutation operators, via selective mutation [29], sufficient or orthogonal operators [15,25]. Performance can be also improved by introduction of faults into intermediate language forms like Java Byte Code, or Intermediate Language of .NET, or usage of repositories if many mutants of the same program are stored.

Another crucial problem is detection of equivalent mutants. In an experiment 40% of all mutations turned out to be equivalent [20]. Existence of equivalent mutants is, in general, an undecidable problem. Therefore, applied algorithms cannot properly classify all mutants in all cases. Different techniques to recognize equivalent mutants in the structural programs were proposed, like constraint based testing (CBT), program optimizing rules, genetic algorithms, program slicing. Dealing with equivalent mutants in object-oriented programs is more complicated and is still not enough tool-supported. Therefore, it is important to prevent creating equivalent mutants. Firstly, the mutation operators should be carefully selected, avoiding those that can lead to many equivalent mutants. Secondly, generation of mutants can be restricted, omitting those cases that might produce equivalent mutants, even if under such restrictions certain proper mutants will be not generated.

3 Advances in the CREAM2 System

Basic principles of the CREAM2 system are similar to those of the previous version [10,11]. The system is aimed at object-oriented and other specialized programming flaws. O-O mutation operators, in opposite to simple traditional ones, have to deal with the structural program dependencies. The mutant generator uses a parser tree of the analyzed code. According to the language grammar and the rules defined in a mutation operator, the places in the tree are identified where the operator could be applied. If

additional correctness conditions are satisfied at any such place, the tree is modified reflecting a fault introduced by the operator (see an example below). The mutant is a program corresponding to the modified tree. Many mutants can be created from the modifications applied in different places according to the rules defined by the same mutation operator.

Based on the experiment results, the next version of the CREAM system has been developed. In the program evolution the following goals were achieved:

- extension of functionality (new mutation operators, collecting of timing features, calculation of statistics, better interface),
- enhancement of program expansibility,
- improvements of mutant generation and its speed-up by usage of a new parser library and reflection mechanism,
- decrease of memory requirements on storing mutated programs,
- correlation of mutation with the code coverage output.

About forty object-oriented and other advanced mutation operators were specified for C# programs and their usefulness evaluated in preliminary experiments [8,9]. Based on this data, in the CREAM2 system 13 object-oriented operators were implemented (Table 1).

Table 1. Selected object-oriented mutation operators.

No		Operators
1	EHR	Exception handler removal
2	EOA	Reference assignment and content assignment replacement
3	EOC	Reference comparison and content comparison replacement
4	IHD	Hiding variable deletion
5	IHI	Hiding variable insertion
6	IOD	Overriding method deletion
7	IOK	Override keyword substitution
8	IOP	Overridden method calling position change
9	IPC	Explicit call of a parent's constructor deletion
10	ISK	Base keyword deletion
11	JID	Member variable initialization deletion
12	JTD	This keyword deletion
13	PRV	Reference assignment with other compatible type

Mutation operators mimic different faults introduced by developers using various of programming techniques. Correctness conditions are applied in order to assure that

a valid mutant is generated, i.e. an injected fault would not be detected by a compiler. They are also partially aimed at preventing creating too many equivalent mutants. In this case, the measured mutation score indicator is closer to the exact mutation score.

For example, the EOA operator simulates a fault that a copy of an object is assigned instead of a desired object.

```
     public class Car : ICloneable{
          public object Clone(){...}
      ...}
      Car s1 = new Car();
      Car s2;
Original code:              Mutated code:
s2 = s1;                    s2 = s1.Clone() as Car;
```

In case of EOA operator, only code extracts of the following syntax were mutated:

```
      <Expression> = <Identifier>
      <Variable>   = <Identifier>
```

If the right side of the assignment were equal to a different expression than an identifier of a variable, as for example result of a method call or a complex expression, the EOA operator would be not applied to this statement. In this way we prevent generating a code that might be not correctly compiled.

Expansibility mechanism of the system was improved. CREAM2 can be extended with new mutation operators implemented in appropriate add-ins without modification of other code. The same technique can be used for substitution of an existing operator with its better version. This feature was verified by users developing traditional mutation operators, applied in further experiments:

- AOR - Arithmetic Operator Replacement (binary: +, -, * ,/, %, unary:+, -, and pre/post: v++, ++v, v–, –v)
- COR - Conditional Operator Replacement (binary: `&&, ||, &, |, ^`, and unary: `!`)
- LOR - Logical Operator Replacement (binary: `&, |, ^`, and unary: `~`)
- ROR - Relational Operator Replacement (`>, >=, <, <=, ==, !=`)
- ASR - Assignment Operator Replacement (short-cut: =, +=, -=, *=, /=)

In the previous system, a class hierarchy was created and used to examine inheritance dependencies between classes, identification of member types, etc. Such information was helpful during identification of possible mutation places in the parser tree. The approach had some limitations, as classes from external libraries could not be taken into account in the considered class hierarchy. Lack of access to embedded types and library classes impeded implementation of more sophisticated object-oriented operators, like IHI or EOA. For example, the IHI operator requires information about members inherited from the base class. If the base class belonged to a library, information was inaccessible. The problem was solved in CREAM2 by usage of reflection mechanism for creation and inspection of the type hierarchy. More advanced mechanisms of type checking minimized the possibility of creation of invalid mutants.

In CREAM2, instead of C# parser *kscparse* used previously, a new parser from the *NRefactory* library [30] was applied. Therefore those files that include C# 2.0 constructions could be also mutated. The new parser enables faster movement on a parser tree and its modification. In result, a number of ill-defined mutants decreased and mutants could be created more effectively.

Experiments performed with the previous system were limited by requirements on disk space for storing code of many mutants. A mutated project differs only in few code lines from the original one. Solving this problem, we applied the SVN (Subversion) server [13] managing program versions. An original project is stored as the first version in a repository and each mutant is one of next versions. Therefore, only differences between a mutant and its origin are stored.

The program repository was also used as a buffer for simple integration of the mutant generator with other testing tools. CREAM2 cooperated with Tester - a distributed environment integrating software testing tools [12]. It can organize a hybrid testing process consisting of steps performed by differed testing tools and statistic evaluation of various test results. One of integrated tools was the IBM Rational PureCoverage tool. Code coverage results of a mutated program can be read by CREAM2. Taking into account coverage data can be used as a primarily condition for assessing test suites. It bounds a number of not killed mutants that have to be classified of being equivalent or not, because mutations introduced to areas not covered by tests are often not killed by these tests [9].

New versions of C# language (2.0 and 3.0) provide new programming constructions, e.g. generic types, nullable types, extensions to static types, properties, indexers, covariance, delegates etc., that were considered for new advanced mutation operators. A set of such operators were proposed and examined in preliminary experiments. The results were not very promising, because the new features concentrate mainly on simplification of code writing and either their misuse was easily detected by a compiler or many equivalent mutants were generated. Therefore these new operators were not included to the current CREAM2 version.

4 Experiments on Operators Evaluation and Tests Qualification

Capabilities of the CREAM2 system were verified in experiments performed on a set of widely used, open-source programs, which are distributed together with their unit test suits. The basic complexity measures of the programs and their tests are summarized in Table 2.

The experiments were performed according to the following scenario:

1. An original project with its unit test suite was added to the local or remote project repository.
2. The project was run against its tests. Code coverage of the program was evaluated. Program test results (an oracle) and coverage results were stored in the test results data base of the Tester system [12].
3. CREAM2 generated set of mutants of the project according to selected mutation operators and coverage data. Updates of compiled valid mutants were stored in the project repository.

Table 2. Complexity measures of mutated programs.

No	Programs	Number of		Source code [LOC]		Project
		Classes	Tests	pure	with tests	total [MB]
1	Adapdev.NET	252	14	68158	68315	23.0
2	Castle.Core	57	165	6190	8737	1.5
3	Castle.DynamicProxy	73	78	6933	10594	2.0
4	CruiseControl.NET	352	1279	31344	62412	32.0
5	NCover	38	72	4347	7403	18.4
6	NHibernate.IEsiCollections	8	153	21047	22293	0.4

4. Mutants were tested with their test suites using NUnit [31]. Generated results were compared against the given oracle, in order to decide whether a mutant was killed or not, and stored in the data base.
5. Mutation results were evaluated using stored data.

Number of mutants created by CREAM2 for different object-oriented operators are shown in Table 3. The first column identifies programs described in Table 2. Results for the IHD operator are omitted, because no mutants were generated. Usage of the repository for storing mutants allow to significantly raise limits on number of performed mutants. Therefore all possible mutants could be created, as for example above seven thousand mutants of *Adapdev.NET* (number 1).

Table 3. Number of mutants in programs per operators.

No	Operators												Sum
	EHR	EOA	EOC	IHI	IOD	IOK	IOP	IPC	ISK	JID	JTD	PRV	
1	89	123	1839	2	88	52	50	23	94	738	2365	1963	7425
2	0	5	74	0	10	10	0	13	2	22	42	51	229
3	0	9	41	0	9	9	1	4	4	38	5	49	169
4	7	49	398	12	65	53	10	22	26	443	964	470	2519
5	0	17	61	0	3	3	1	1	6	55	77	31	255
6	0	0	28	0	2	2	0	0	0	4	45	2	83
Sum	96	203	2441	14	177	129	62	63	132	1300	3498	2566	10680

In Table 4 and 5 mutation results of the programs and the object-oriented mutation operators are presented. Numbers of mutants killed by the unit tests were in many cases not very high. Mutation score was calculated as a ratio of the number of mutants killed over the total number of generated valid mutants. Precisely, it is the lower bound on mutation score, called in [19] as mutation score indicator. The exact mutation score should be calculated over the number of all non-equivalent mutants. In the CREAM2 mutant generator much effort was spent to create only valid mutants and if possible avoid equivalent mutants. However, equivalent mutants can be also created. In simple cases they could be eliminated by hand. But, for the most discussed programs, when the numbers of not killed mutants were higher (like for the PRV operator), it was impossible to identify precisely all equivalent mutants.

Mutation results of all programs except *NCover* were low (16-37%) and showed a poor quality of the considered test suites. But, tests of *NCover* (no 5) killed all mutants of six operators and the most of mutants for the rest of them, giving a total mutation score equal to 97.6%. It is characteristic that in general a test suite good at killing mutants is dealing quite well with mutants generated by the given different mutation operators.

Table 4. Number of killed mutants.

No	Operators												Sum
	EHR	EOA	EOC	IHI	IOD	IOK	IOP	IPC	ISK	JID	JTD	PRV	
1	29	40	539	2	15	14	18	10	47	291	690	384	2079
2	-	0	9	-	0	0	-	3	0	7	4	14	37
3	-	0	9	-	4	4	0	1	0	33	3	34	88
4	2	47	19	1	4	3	0	2	0	29	212	260	532
5	-	17	60	-	3	3	1	1	5	54	74	31	249
6	-	-	24	-	0	0	-	-	-	1	5	1	31

The ability of tests to kill mutants generated by different mutation operators was compared. It was measured, how many tests of the suite killed a mutant (Table 6). In case of two biggest programs (1 and 4) any mutation was killed usually by one test. Test cases of these programs were diverse and devoted to disjoint subjects. In the remaining programs a test was killed on average by about ten to several dozen of tests. It points out the existence of many redundant test cases as far as the given programming flaws are concerned. On the other hand, there are many mutants not killed by the tests from these test suites. For program number 5, having the best mutation score, the numbers of tests killing mutants are except the JID operator on a moderate level of about few tests. High mutation results for different mutation operators were obtained by a limited number of tests, but effective and focused on different subjects. We can observe that a number of tests killing mutants was in general a feature of a test suite. There was no

Table 5. Mutation score in [%].

No	Operators												Sum
	EHR	EOA	EOC	IHI	IOD	IOK	IOP	IPC	ISK	JID	JTD	PRV	
1	32.6	32.8	29.3	100	17.1	26.9	36	43.5	50.0	39.4	29.2	19.6	28.0
2	-	0	12.1	-	0	0	-	23.1	0	31.8	9.5	31.8	16.7
3	-	0	21.9	-	44.4	44.4	0	25.0	0	86.8	60.0	73.9	37.6
4	25.0	4.2	19.0	7.14	6.1	5.6	0	9.1	0	6.6	21.9	55.7	21.4
5	-	100	98.3	-	100	100	100	100	83.3	98.2	96.1	100	97.6
6	-	-	85.7	-	0	0	-	-	-	25.0	11.1	50	37.3

significant dependency between this measure and the kind of the operator. The averaged percentage of tests that killed a mutant are given in the last column.

Table 6. Number of tests killing mutants.

No	Operators												Avg.	Avg.
	EHR	EOA	EOC	IHI	IOD	IOK	IOP	IPC	ISK	JID	JTD	PRV		[%]
1	1	1	1.1	28.6	1	1.1	1.1	1.1	1.1	1.1	1.1	7.9	1.1	7.6
2	-	0	5.2	-	0	0	-	5	0	15	20.8	31.8	10.8	6.5
3	-	0	25.8	-	40	40	0	40	0	40	12.3	40	37.6	48.2
4	1	1	0.8	1	1	1	0	1	0	1.0	1.0	1.1	1.1	0.1
5	-	1.5	2.1	-	1.4	1.4	1.4	1.4	1.9	71	5.3	8.4	13.2	18.4
6	-	-	3.5	-	0	0	-	-	-	73	73	15	17.3	11.3

Mutation results of the object-oriented operators were compared with results of standard operators. Five standard mutation operators (see Sec. 3) were applied for two programs *Adapdev.NET* and *NCover* and a subset of all possible mutants were generated. No mutants were created for the ASR operator because no short-cuts (like +=) were used in these programs. Mutation score was calculated for given numbers of mutants (Fig. 7). All mutants were killed by the test suites in almost all cases. Good results of the test suite were expected for *NCover* as the tests were also good at detecting various flaws injected by the object-oriented operators. However, good results (99%) were obtained also for *Adapdev.NET*, while mutation score for the object-oriented operators was about 28%. In general, simple faults simulated by standard operators are more eas-

Table 7. Mutation results for standard mutation operators.

No	Programs	Number of mutants					Mutation score [%]				
		AOR	COR	LOR	ROR	Sum	AOR	COR	LOR	ROR	Sum
1	Adapdev.NET	60	46	40	40	186	100	98	100	100	99
5	NCover	40	8	0	80	128	100	100	-	100	100

ily to detect by tests. It is not sufficient to have test suites adequate to standard mutations in order to detect faults misusing object-oriented rules.

The experiment results have to be considered in the context of possible threats to validity. A big number of mutants helped to cope with threats to the statistical conclusions validity. However, for some mutation operators, e.g. IHI, these numbers were quite small, because such operator deals with less frequent programming constructions. Mono-operation bias is a threat as the experiment was conducted on a single development project. Therefore several, different programs were used in the experiments. The selected programs and their test suites are commonly used and neither prepared nor adapted for the experiments. It reduces threats to external validity concerning possibility of results generalization.

5 Performance Evaluation

In experiments we measured the impact of the applied technology on the system performance. Time and storage constrains were compared for different solutions and to the previous version of the mutation system.

One of improvements of the CREAM2 system was substitution of the parser module. Usage of a new parser speeded up the generation of mutants, as routines manipulating on parser trees were more effective. Average time of creation of mutants' source code was about 2-4 times shorter than in the previous system.

Repository for storing mutants was applied in order to reduce disk storage requirements. However, storing mutants in the repository takes more time than direct storing of whole mutants. Repository can be saved locally on the same computer as the mutants generation system, or on another remote computer. In dependence on the program characteristics, time of generation of mutants for a local repository was longer about 61-121% than the time of direct storing whole mutants in the same computer. While using a remote repository, the corresponding time was longer of about 39-91%. The average times of generating and storing a mutant in different locations for selected programs are given in Table 8. The time of generation to a repository depends on a program size. For bigger programs (like programs no 1, 4 and 5 - Table 2), time of using a remote repository was shorter than using a local one. The repository server had to analyze many files in order to identify and store the update. Concurrent work of the server and the generator was beneficial and exceeded the transmission overhead. For medium-size programs (as 2 and 3), times of using both kinds of repository were comparable. In

Table 8. Average time of generating and storing a mutant in [s].

No	Programs	On local disk	In local repository	In remote repository
1	Adapdev.NET	27.98	61.9	47.71
2	Castle.Core	3.20	6.24	6.09
3	Castle.DynamicProxy	3.77	6.22	6.04
4	CruiseControl.NET	32.98	33.64	25.13
5	NCover	14.47	26.19	20.08
6	NHibernate.IEsiCollections	2.80	5.59	9.30

case of the smallest program (number 6) usage of the local repository took less time, because the server of the repository had smaller requirements on processor time and the transmission overhead was noticeable.

It should be notated that, although discrepancies of average times are not very high, difference of times required for processing many mutants can be substantial. For example, generation and execution of all mutants of *Adapdev.NET* took 29 hours more using the local repository than the remote one.

The main goal of application of the SVN repository was decreasing memory requirements on storing mutants. Mutated programs are very similar to each other. In a repository only the updates to the original programs have to be stored. An empty repository needs about 80 kB and this is a small overhead in comparison to program sizes. Amounts of disk memory are compared in Table 9. Numbers in the last column refer to the repository including given numbers of mutated programs. The previous column shows how many disk space would be necessary to store the same mutated projects directly, without using the repository. The results showed that required memory was reduced in size from 10 to 100 times according to different programs and numbers of mutants.

6 Conclusions and Future Work

This paper reports on experiments on the object-oriented mutation testing of C# programs performed by the CREAM2 tool and its evolution to the current version. Application of advanced mutation operators implies more computational effort than the standard operators. But, according to experiments, test cases adequate to standard mutations are not enough capable to expose additional faults dealing with object-oriented constructions. Therefore, automated usage of advanced mutants for test qualification would be profitable.

Performed experiments showed that the new version of the CREAM2 system can generate object-oriented mutants more precisely than the previous one. The problem of big amounts of disk space was solved by usage of SVN repository for storing mutants.

Table 9. Comparison of memory requirements.

No	Programs	Number of mutants	Disk storage	Repository storage
1	Adapdev.NET	7425	167 GB	3 GB
2	Castle.Core	229	204 MB	34 MB
3	Castle.DynamicProxy	169	481 MB	31 MB
4	CruiseControl.NET	2519	79 GB	1 GB
5	NCover	255	5000 MB	90 MB
6	NHibernate.IEsiCollections	83	71 MB	6 MB
	Sum	10680	252 GB	4 GB

It gave storage reduction of about 10 to 100 times. A bottleneck of experiments remains time requirements. However, using improved organization and automation of the whole testing process, all implemented object-oriented mutants of the given programs, i.e. above ten thousand of mutants, were generated, tested and analyzed. The long execution time could be tolerable when the whole process can be realized without user supervision. Essential reduction of time requirements can be obtained by elimination of mutants compilation. It is straightforward for standard mutation operators, but not for the advanced ones. First experiments with the prototype ILMutator system [28] gave promising results. It inserts mutations that simulate objects-oriented mutations of C# language direct into the Intermediate Language of .NET.

The most crucial problem of the mutation testing automation is still exclusion of equivalent mutants. Although the accurate recognition of all equivalent mutants would be in general impossible, but acceptable results could be obtained by limitation of their number to small amounts that can be neglected. There are two complementary approaches: preventing of equivalent mutant generation and detection of equivalent mutant in a set of non killed mutants. We concentrated so far on the first approach, while identification of equivalent mutants for advanced mutants of C# needs future work.

Another important issue concerning testing automation is execution of test runs and concluding on the killing of mutants. In the reported experiments, NUnit was used to run mutants with unit tests and to generate tests results compared to an oracle. In general, different test cases and test environments can be apply together with mutation testing. In the future, capture and replay environments with testing scripts including verification points and other verification mechanisms can be combined with the mutant generation system.

Acknowledgments

The authors would like to thank Krzysztof Sarba, for his help in performing experiments.

References

1. Voas, J.M., McGraw, G.: Software Fault Injection, Inoculating Programs Against Errors. John Wiley & Sons Inc. (1998)
2. Andrews, J. H., Briand, C., Labiche, Y., Namin, A.S.:, Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. IEEE Transactions on Software Engineering 342(8), 608–624 (2006)
3. Offut, J.: A Mutation Carol Past, Present and Future. Proc. of the 4th International Workshop on Mutation Analysis, Mutation'09, Denver, Colorado (2009)
4. Chevalley, P.: Applying Mutation Analysis for Object-oriented Programs Using a Reflective Approach. In: Proc. of the 8th Asia-Pacific Software Engineering Conference, ASPEC, 267–270 (2001)
5. Chevalley, P., Thevenod-Fosse P.: A Mutation Analysis Tool for Java Programs. Journal on Software Tools for Techn. Transfer (STTT) 5(1), 90–103 (2003)
6. Kim, S., Clark, J., McDermid J.A.: Class Mutation: Mutation Testing for Object-oriented Programs. In: Proc. of Conference on Object-Oriented Software Systems, Erfurt, Germany (2000)
7. Ma, Y-S., Kwon, Y-R., Offutt, J.: Inter-class Mutation Operators for Java, Proc. of International Symposium on Software Reliability Engineering, ISSRE'02, IEEE Computer Soc. (2002)
8. Derezińska, A.: Advanced Mutation Operators Applicable in C# programs. In: K. Sacha (ed.) Software Engineering Techniques: Design for Quality, IFIP, vol. 227, pp. 283–288. Springer, Boston (2006)
9. Derezińska, A.: Quality Assessment of Mutation Operators Dedicated for C# Programs. In: Proc. of the 6th International Conference on Quality Software, QSIC'06, pp. 227–234. IEEE Computer Soc. Press, California (2006)
10. Derezińska, A., Szustek, A.: CREAM - a System for Object-oriented Mutation of C# Programs. In: Szczepański, S., Kłosowski, M., Felendzer, Z. (eds.) Annals Gdansk University of Technology Faculty of ETI, No 5, Information Technologies, vol.13, pp. 389–406. Gdańsk (2007)
11. Derezińska, A., Szustek, A.: Tool-supported Mutation Approach for Verification of C# programs. In: W. Zamojski, J. Mazurkiewicz, J. Sugier, T. Walkowiak (eds.) Proc. of International Conference on Dependability of Computer Systems, DepCoS-RELCOMEX 2008, pp. 261–268. IEEE Comp. Soc. USA (2008)
12. Derezińska, A., Sarba, K.: Distributed Environment Integrating Tools for Software Testing, In: T. Sobh (Ed.) Advances in Computer and Information Sciences and Engineering, Springer (2009) (to appear)
13. Subversion svn, http://subversion.tigris.org
14. Delmaro , M., Maldonado, J.: Proteum - a Tool for the Assessment of Test Adequacy for C Programs. In: Proc. of Conference on Performability in Computing Sys., PCS96, pp. 79–95 (1996)
15. Namin, S., Andrews, J.H.: On Sufficiency of Mutants. In: Proc. of the 29th International Conference on Software Engineering, ICSE'07 (2007)
16. Jia, Y., Harman, M.: MiLu: a Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language. TAIC-Part (2008)
17. Irvine, S.A. et al.: Jumble Java Byte Code to Measure the Effectiveness of Unit Tests, Mutation'07 at TAIC.Part'07, 3th Inter. Workshop on Mutation Analysis, pp. 169–175. Cumberland Lodge, Windsor UK (2007)
18. Kim, S.-W., Harrold, M.J. Kwon, Y.-R.: MuGamma: Mutation Analysis of Deployed Software to Increase Confidence and Assist Evolution. In: Proc. of the 2nd Workshop on Mutation Analysis, Mutation 2006, Releigh, North Carolina, Nov. (2006)

19. Madeyski, L.: On the Effects of Pair Programming on Thoroughness and Fault-finding Effectiveness of Unit Tests. In: Münch, J., Abrahamsson, P.J. (eds.) Profes 2007. LNCS, vol. 4589, pp. 207–221. Springer, Heidelberg (2007)
20. Grün, B.J.M, Schuler, D., Zeller, A.: The Impact of Equivalent Mutants. In Proc. of the 4th International Workshop on Mutation Analysis, Mutation'09, Denver, Colorado (2009)
21. Moore, I.: Jester a JUnit Test Tester. eXtreme Programming and Flexible Process in Software Engineering – XP2000 (2000)
22. Do, H., Rothermel, G.: A Controlled Experiment Assessing Test Case Prioritization Techniques via Mutation Faults. In: Proc. of the 21st IEEE International Confe on Software Maintenance, ICSM'05, IEEE Comp. Soc. (2005) 411–420
23. Ma, Y-S., Offutt, J., Kwon, Y-R.: MuJava: an Automated Class Mutation System, Software Testing, Verification and Reliability 15(2) (2005)
24. Nester, http://nester.sourceforge.net/
25. Lee, H.-J., Ma, Y.-S., Kwon, Y.-R.: Empirical Evaluation of Orthogonality of Class Mutation Operators. In: 11th Asia-Pacific Software Engineering Conference, IEEE Computer Society (2004)
26. Smith, B.H., Williams, L.: A Empirical Evaluation of the MuJava Mutation Operators. Mutation'07 at TAIC.Part'07, 3th International Workshop on Mutation Analysis, pp. 193–202. Cumberland Lodge, Windsor UK (2007)
27. Baudry, B., Fleurey, F., Jezequel, J.-M., Traon, Y.L.: From Genetic to Bacteriological Algorithms for Mutation-based Testing. Software Testing, Verification and Reliability 15(2), 73-96 (2005)
28. Kowalski, K.: Implementing Object Mutations into Intermediate Code for C# programs, Bach. Thesis, Inst. of Comp. Science, Warsaw Univ. of Technology (2008) (in polish)
29. Offut, J., Rothermel, G., Zapf, C.: An Experimental Evaluation of Selective Mutation. In: Proc. of the 15th International Conference on Software Engineering, pp. 100-107 (1993)
30. NRefactory, http//codeconverter.sharpdeveloper.net/Convert.aspx
31. NUnit, http//www.nunit.org