Schedae Informaticae Vol. 25 (2016): 227–236
doi: 10.4467/20838476SI.16.017.6198

Mutation Churn Model

Michał Mnich, Adam Roman, Piotr Wawrzyniak
Faculty of Mathematics and Computer Science
Jagiellonian University
ul. Łojasiewicza 6, 30-348 Kraków, Poland
e-mail: {*michal.mnich, adam.roman, piotr.wawrzyniak*}*@uj.edu.pl*

Abstract. Mutation testing is considered as one of the most effective quality improvement technique by assessing the strength of the actual test suite. If no test is able to kill a given mutant, this means that the tests are not strong enough and we need to write additional one that will be able to kill this mutant. However, mutation testing is very time consuming. In this paper we investigate if it is possible to reduce the scope of the mutation analysis by running it only on the new or changed part of the code. Using data from the real open-source projects we analyze if there is a relation between mutation scope reduction and effectiveness of the mutation analysis.

Keywords: Software testing, Mutation testing

1. Introduction

Mutation testing is a technique for measuring the quality of existing tests. For a given program, a number of its mutants is generated. Each mutant represents some defect artificially injected by so-called mutation operator. All mutants, together with the original program are tested by the same test suite. A mutant is killed if at least one test gives different results for this mutant and the original program. Depending on the ratio of the killed mutants (so-called mutation ratio) we infer about the quality

Received: 19 September 2016 / Accepted: 25 September 2016

of our tests. If no test was able to kill a given mutant, this means that the tests are weak and we should add at least one more test that is able to kill this mutant.

Mutation testing is considered as one of the most effective testing methods. Unfortunately, there are several problems regarding this technique:

- It is very time-consuming – usually there is a large number of mutants and a large number of tests. All mutants need to be compiled. After that, each mutant needs to be tested till some test kills it (or till all the tests are performed and none of them is able to kill this mutant). This takes much time.
- If a mutant introduces an infinite loop, we do not know if we should stop the application performance or if it just takes a lot of time to finish the work.
- Some mutants may be equivalent to the original code. In general, checking if a mutant is equivalent to the original program is undecidable.

Of course there are some methods that try to overcome these difficulties at least to some degree, but in general the above mentioned problems are still too important to ignore them.

Mutation testing has quite a long story. The paper [1] is considered as a seminal paper on this subject. The foundations of mutation testing theory are presented in a book of Ammann and Offutt [2].

Nowadays it is very common to utilize agile techniques and new software engineering practices like continuous integration. Such approach heavily uses regression testing. Regression test suites may be very large. Performing mutation testing on such large set of tests is very time consuming. In the literature several methods were proposed to overcome this issue. One of the simplest approaches is to reduce tests. Another one is test case prioritization, where we try to define a test execution for each mutant, such that the test that kills a given mutant should be executed as early as possible. See [3] and [4] for the survey on this topic.

When using continuous integration we usually deal with many software versions. Software may be released every week, every day, or even after any single change was committed to the repository. If we perform mutation analysis for our code, the following question arises: having a new release, is it enough to perform mutation testing only on the changed part of the code, or should we do the mutation testing for the whole code, even for the part that has not been changed? This question is of big importance, as it may allow us to heavily reduce the time spend on the mutation process.

At first glance it may seem that we do not need to perform the mutation analysis for the unchanged code, but the following simple example shows that is may not necessarily be true. In the example we have two versions of the code being a mutant of the original program. The unchanged part in the first version was killed by one test. In the second version the same test reached this part of the code, but was not able to kill this mutant.

```
function f(int x) { // version N
  x := x - 1
  if (x >= 1 || x <= -1) // unchanged part
    return 1
```

```
  else
    return 0
}

function f(int x) { // mutant of the version N
  x := x - 1
  if (x > 1 || x <= -1) // mutation
    return 1
  else
    return 0
}

function f(int x) { // version N+1
  x := 1 - x  // changed code
  if (x >= 1 || x <= -1) //unchanged part
    return 1
  else
    return 0
}
```

Suppose we have a test for $x = 2$. In version N for this test the program $f(2)$ returns 1. The mutation changed the relational operator from $\geq$ to $>$. The same test will kill this mutant. In the version $N + 1$ the first instruction was changed to `x:=1-x`. The rest of the code remains unchanged. Our test reaches the predicate in the *if* statement, but will not be able to kill the mutant.

The fundamental question is as follows: what is the probability of such situation in the real projects? If this probability is small, it may be reasonable to perform mutation analysis only on the unchanged parts of the code in the new release. Otherwise, we should do the full analysis.

In order to answer this question, we designed and performed an experiment on two open-source projects. In Section 2. we describe the mutation testing process and the mutation operators used in our research. In Section 3. we present the experiment and its results. We end with conclusions presented in Section 4.

2. Mutation testing

Mutation testing was originally developed by DeMillo in 1978 [1]. In a mutation testing we are given a source code of the original program P, called *System Under Test* (SUT) and a related test suite T. A tester creates many copies of the SUT, but in each copy she introduces some small syntactic changes. These changed copies are called *mutants*. Each change should represent a simulated developer's mistake. All changes are introduced by using the predefined *mutation operators* which transform the code with a well-defined rules. It is important that after such a change the mutant should be able to be compiled and run.

The next step is to test both original program and all the mutants using the given test suite. Let us denote by M and t the mutant and test, respectively. By $P(t)$ (resp. $M(t)$) we denote the result of the execution of P (resp. M) for a test $t \in T$. If, for a given M, there exists t such that $P(t) \neq M(t)$ we say that M was *killed* by t. This means that our tests are able to detect a defect injected into M. If, from the other hand, no test is able to kill M, that is, $M(t) = P(t)\ \forall t \in T$, it means that our tests are unable to detect this defect. In such case a tester should add a new test to T which would be able to kill M. The fraction of killed mutants is called a *mutation score*.

Mutation testing can be therefore considered as a process of testing the tests, not the program itself. It checks the quality of a given test suite T. Mutation testing has an indirect influence on the quality of the SUT, because it requires that new tests, able to kill the remaining, live mutants, should be added. This increases the ability of the test suite to detect new defects.

The effectiveness of the mutation process depends significantly on the type of mutation operators used. If the operators are trivial, the mutants will be weak and almost all tests will be able to kill them all. Mutations produced by the operators should reflect the real mistakes done by developers. In the literature many types of mutation operators were proposed, see for example [5]. Mutation operators can produce not only simple syntactic mutants (like changing one relational operator into another), but also mutants reflecting the object-oriented types of faults [6].

Mutation analysis can also be used as a method for predicting the field defects. This is done by applying the modification of a capture-recapture method: if a test suite T was able to find N out of D (unknown) real defects in the SUT and the mutation score (using the same test suite T) was S, we may claim that this score can be also applied to the real defects, so $S \approx N/D$. Therefore, we estimate $D \approx N/S$. Another possible utilization of mutation testing is fault localization [7].

3. Experiments and results

For our experiment we used a modified version of the PIT 1.1.7 tool [8], a software for performing mutation testing in Java. PIT has several optimizing features that allow us to perform mutation analysis as quickly as possible. One of them works as follows: when the first test kills a given mutant M, no other tests are run for M. We disabled this optimization feature and forced PIT to run *all* tests that could potentially kill a given mutant. This was done in order to check how many tests kill each mutant. Basing on PIT reports we created the unique names for all mutants and tests, based mostly on method/test names and classpaths. Using those unique names we manually verified the changes in tests and methods between different versions of the SUT to identify new, modified and removed tests/methods.

The manual part was necessary, as the following situations might occur: the mutant seemed to be the same (but the source code has changed); there were changes in the tests (important from the business logic or just cosmetic); a mutant was killed

Table 1. Basic characterizations of applications under test.

Application	Version	LOC	# of methods	# of tests
JUnit	4.9	15440	106	409
JUnit	4.11	23794	111	506
JUnit	4.12	26079	135	727
EasyMock	3.2	12659	213	747
EasyMock	3.3	13158	233	764
EasyMock	3.3.1	13240	237	765

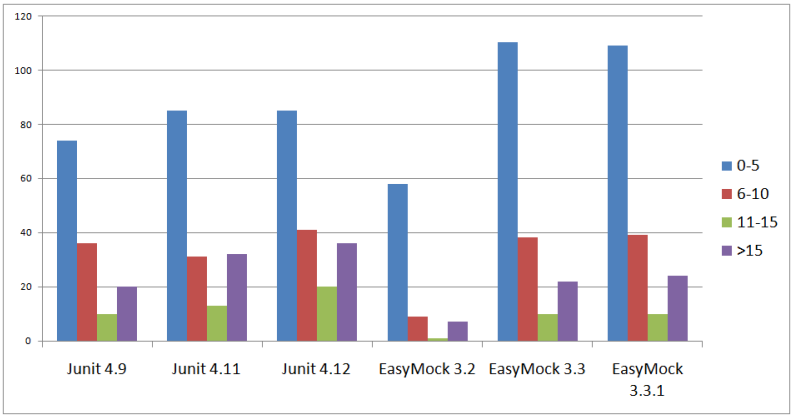

Figure 1. Distribution of number of methods in classes for all applications under test.

in one version and survived in the next one; etc. In the next step of our experiment we compared mutants between versions identifying new, changed and removed ones. Finally, we analyzed the changes of mutants statuses identifying the sources of changes.

The important fact is that we performed our experiment on projects, where during the development no mutation testing has been used. Because of that we where able to decide which mutants are relevant for our research. If a mutation testing was used, the test suite modification would have biased the results of our experiment.

For the experiment we chose three releases of the two open-source projects: JUnit and EasyMock. Both applications where taken from the official repositories on GitHub. We skipped JUnit 4.10 because it is very similar to version 4.11. In the view of the differences between 4.11 and 4.12, version 4.10 could be treated like a beta version of 4.11. Table 1 gives the basic characterizations of the examined projects.

Fig. 1 presents the distribution of the number of methods within a class for each project and for all versions.

In Table 2 the basic metrics for all programs are shown. We used the following, standard metrics:

Table 2. Basic metrics.

		CC	CK metrics			
Application	Version	Mean ± Sd	WMC	DIT	NOC	LCOM
JUnit	4.9	1.22 ±0.72	4.55 ±8.1	1.50 ±0.9	0.42 ±3.27	0.04 ±0.15
JUnit	4.11	1.22 ±0.67	4.59 ±8.75	1.45 ±0.85	0.38 ±3.21	0.04 ±0.16
JUnit	4.12	1.17 ±0.48	4.27 ±0.19	1.40 ±0.80	0.34 ±2.94	0.04 ±0.16
EasyMock	3.2	1.46 ±1.25	9.55 ±15.37	1.21 ±0.56	0.12 ±0.71	0.09 ±0.21
EasyMock	3.3	1.46 ±1.23	9.01 ±15.11	1.22 ±0.56	0.14 ±1.1	0.09 ±1.21
EasyMock	3.3.1	1.46 ±1.13	8.89 ±15.01	1.24 ±0.58	0.14 ±0.92	0.09 ±1.21

- CC (Cyclomatic Complexity) – measures the complexity of the code structure in terms of the number of decision points (CC = number of decision points in the code + 1). The lower, the better.
- WMC (Weighted Methods per Class) – number of methods defined in the class. The lower the WMC is, the better, as it is a well-known fact that high WMC leads to more faults.
- DIT (Depth of Inheritance Tree) – the maximal length of the inheritance path from the class to the root class. DIT measures the depth of a class hierarchy. The deeper the tree is, the more methods and variables it is likely to inherit, making the code more complex, so this value should be kept small.
- NOC (Number of Children) – number of immediate sub-classes of a class. NOC measures the breadth of a class hierarchy. A high value of NOC may indicate: high reuse of a base class; that base class may require more testing; improper abstraction of the parent class; misuse of sub-classing.
- LCOM (Lack of Cohesion of Methods) – it is calculated as follows: take each pair of methods in the class. If they access disjoint sets of instance variables, increase P by one. If they share at least one variable access, increase Q by one. $LCOM = P - Q$ if $P > Q$ and 0 otherwise. If $LCOM = 0$ the class is cohesive. Otherwise, it may require to be split into two or more classes, since its variables belong to disjoint sets. Classes with high LCOM have been found to be more error-prone.

From the data in Table 2 we can see that in case of JUnit there is a tendency to keep the low level of the complexity. Despite the increase in the number of classes the mean value of CC decreases from one version to another. We can also observe the decrease in the standard deviation of CC. This suggests that the source code was refactored. In case of EasyMock the mean cyclomatic complexity is stable with

Table 3. Changes in unit tests.

Version to version	Tests added	Tests deleted	Tests modified
JUnit 4.9 to 4.11	118	21	30
JUnit 4.11 to 4.12	285	64	41
EasyMock 3.2 to 3.3	19	3	14
EasyMock 3.3 to 3.3.1	1	0	0

Table 4. Overall statistics for mutation analysis.

Application	Mutants			
	Generated	Killed	Survived	LOC-to-mutants
JUnit 4.9	1673	954 (57%)	719	9.23
JUnit 4.11	1903	1091 (57%)	812	12.50
Junit 4.12	2351	1494 (64%)	857	11.09
EasyMock 3.2	1374	1148 (84%)	226	9.21
EasyMock 3.3	1424	1196 (84%)	228	9.24
EasyMock 3.3.1	1424	1196 (84%)	228	9.30

a slight decrease of its standard deviation. This may also suggest that EasyMock's code was refactored.

In case of WMC metric the standard deviation is high in comparison to the mean value. This means that the number of methods varies heavily from one class to another, both in JUnit and EasyMock. DIT, in both applications and in all versions, remain on a low level, between 1 and 2. This shows that the class hierarchy is flat in all versions.

Table 3 shows the statistics on how the number of tests was changing from one version to another.

Table 4 presents the overall statistics for the mutation analysis. It shows, for each application version, the total number of mutants generated and the number of mutants killed and survived. The last column presetnts the LOC-to-mutants ratio. This metric says what, on average, is the number of lines per one mutant. If mutants are generated by the same tool in the same way for all versions of a given application then an immediate increase of this metric might suggest a substantial change in the code structure. For example, there may be added a large portion of code that is non-mutable with respect to a given mutation tool and mutation operators.

The great increase of the LOC-to-mutants ratio from JUnit 4.9 to 4.11 is due to a very big change in the code. First, the code size increased significantly from 15 KLOC to almost 24 KLOC. Second, most of the added code was a code of interfaces and some other non-mutable elements.

Table 5 presents the detailed statistics on mutation ratio broken by mutation operator type. In our experiment we used 6 different mutation operator types available in the PIT tool:

- CBM – Conditionals Boundary Mutator. Replaces the relational operators $<$, $\leq$, $>$, $\geq$ with their boundary counterpart ($\leq$, $<$, $\geq$, $>$ resp.).

Table 5. Detailed mutation analysis by mutation operator type. G = generated, K = killed, JU = JUnit, EM = EasyMock.

App	CBM		IM		VMCM		RVM		MM		NCM	
	G	K	G	K	G	K	G	K	G	K	G	K
JU 4.9	59	25	40	19	475	238	587	366	53	12	459	297
JU 4.11	67	27	45	21	574	287	634	399	58	19	525	338
JU 4.12	78	47	60	39	586	291	808	549	61	23	680	488
EM 3.2	65	57	40	35	318	276	479	374	26	22	446	384
EM 3.3	64	56	39	34	332	289	503	397	29	25	457	395
EM 3.3.1	64	56	39	34	332	289	503	397	29	25	457	395

Table 6. Different types of succession types.

Type	JUnit	EasyMock
-K	45	16
-S	3	2
KS	2	0
SK	2	0

- IM – Increments Mutator. Mutates increments, decrements and assignment increments and decrements of local (stack) variables. Replaces increments with decrements and vice versa, for example $i++$ to $i--$.
- VMCM – Void Method Call Mutator. Removes method calls to void methods.
- RVM – Return Values Mutator. Mutates the return values of method calls. Depending on the return type of the method, different mutation is used. For example, for boolean type it replaces true with false, for Object typr it replaces non-null value with null etc.
- MM – Math Mutator. Replaces binary arithmetic operations for either integer or floating-point arithmetic with another operation. For example, it changes $+$ with $-$, $*$ with $/$, & with $\|$ etc.
- NCM – Negate Conditionals Mutator. Mutates all conditionals found: it changes $==$ with $!=$, $\leq$ with $>$ and $\geq$ with $<$.

In Table 6 we present the quantitative analysis of the different sequences of the mutation result types (for siplicity we call these types *successions*). Each value represents the sum of a given type of succession from the first to second and from the second to third version for both applications. We consider four interesting types of succession.

- -K: in version N there was no mutant and in version $N+1$ it appeared and was killed,
- -S: in version N there was no mutant and in version $N+1$ it appeared and survived,

- KS: in version N a mutant was killed, but in version $N + 1$ it survived,
- SK: in version N a mutant survived, but in version $N + 1$ it was killed.

There are also two other types of succession: KK and SS, but they are not interesting for our research. KK simply means that a mutant was killed in both versions, SS – that a mutant survived in both versions. The non-zero number of SS successions means that mutation analysis was not used in case of both JUnit and EasyMock. This makes these applications good candidates for our research, because using the mutation analysis would result in test suite improvement and hence would distorted the experiment results.

Types -K and -S represent the situations where the new mutants were introduced due to code change. New mutants could be introduced only when a given part of a code was changed or when a new part of code was written. Notice that in case of JUnit there were 48 new mutants in total and the test suite was able to kill most of them (45, which is about 93%). Also in case of EasyMock almost all new mutants were killed (16 out of 18, which is about 89%).

The most interesting cases are KS and SK types. KS means that a previously killed mutant survived in the next version. This is an undesirable situation if we want to focus only on mutation analysis for the changed part of a code. In case of EasyMock no mutants fall into KS category. In case of JUnit we observed 2 such situations. Notice that this is rather negligible number when compared to the total number of over 3500 mutants generated in versions 4.9 and 4.11. We may conclude that the (undesirable) KS succession occurs incidentally.

Let us briefly discuss these two cases mentioned above. In JUnit from 4.11 to 4.12 a KS succession took place when a metod used by the mutated method was changed. The test and the main method remained the same. The second example of KS was observed in JUnit from 4.9 to 4.11. It occurred by moving a method to another one with some small modifications and with no test updates.

The SK succession type means that the previously survived mutant was killed in the next version of the SUT. This is a desirable (although very rare, when we do not use mutation testing to improve our test suite) situation, as we get more killed mutants for free. In case of JUnit there were 2 such cases. We did not observe SK succession in case of EasyMock. In JUnit all SK successions were observed only when a new test was added.

4. Conclusions

In this paper we investigated the effectiveness of mutation analysis performed for a dynamically changing code. This model suits well for agile and open-source projects and also for projects that use continuous integration. By performing a mutation analysis on several, consecutive versions of two applications: JUnit and EasyMock, we counted the number of so-called succession types of mutation analysis results.

The low number of mutants following the KS type of succession suggests that performing a mutation analysis on the unchanged part of a code is not effective, as a mutant killed in the previous version will be killed in the next version with a very high probability. Therefore, when testing the new version, we may restrict our mutation analysis only to the changed or added part of the source code. This approach allows us to significantly reduce the cost of mutation analysis while keeping the risk level related to mutation testing almost unchanged.

The results of our research suggest that it may be reasonable to utilize the above mentioned approach in applications like PIT. If an application for mutation testing is able to analyze the differences between two versions of the source code, it may run the analysis only on the changed or added part of the source code, reducing the time of an analysis. In such case the probability that we miss the generation of a mutant unable to be killed by a given test suite is very low.

5. References

[1] Millo R.A.D., Lipton R.J., Sayward F.G., *Hints on test data selection: help for the practicing programmer.* IEEE Computer, 1978, **11**(4), pp. 34–41.

[2] Ammann P., Offutt J., *Introduction to Software Testing.* Cambridge University Press, Cambridge, UK, 2008.

[3] Zhang L., Marinov D., Khurshid S., Faster mutation testing inspired by test prioritization and reduction. In: *ISSTA 2013 – Proceedings of the International Symposium on Software Testing and Analysis*, 2013, pp. 235–245.

[4] Usaola M.P., Mateo P.R., *Mutation testing cost reduction techniques: A survey.* IEEE Software(, 2010, 27, pp. 80–86.

[5] Kim S., Clark J.A., McDermid J.A., *Investigating the effectiveness of object-oriented testing strategies using the mutation method, software testing.* Verification and Reliability, 2001, 11 (3), pp. 207–225.

[6] s. Ma Y., r. Kwon Y., Offutt J., Inter-class mutation operators for java. In: *Proceedings of the 13th International Symposium on Software Reliability Engineering*, IEEE, 2002, pp. 352–363.

[7] Papadakis M., Traon Y.L., *Metallaxis-fl: mutation-based fault localization.* Software Testing, Verification and Reliability, 2015 (25), pp. 605–628.

[8] *PIT Mutation Testing.* http://pitest.org/, Accessed: 2016-12-30.