CS4110: Mutation Testing

Due on Monday, February 29, 2016

Anelia Dimitrova 4501667, Ioana Leontiuc 4515218, Mengmeng Ye 4407326

Contents

Mutation Testing basics

Mutation Testing is a method of evaluating the quality of the current test suite, as presented in the literature survey [Jia and Harman(2011)]. This method was proposed in 1970.

Mutant is a faulty version of the original program. They are based on syntactic changes (i.e. changing the AND operator into OR). They can be first order, formed by one change or n-th order mutants, by applying n changes. [Jia and Harman(2011)]

Killing mutants All mutants are run through the tests suite which the original program passes. If the test suite is good the mutant should fail at one of the tests. By failing a test, the mutant is said to be killed. The mutant fails the test by outputting a different result than the original program. [Jia and Harman(2011)] Studies have shown that by killing first order mutants, 99% of all second and third order mutants were also killed. The average survival ratio for first order mutants is 1/n and 1/n*n for second order mutants. [Jia and Harman(2011)]

Survivor mutants After the test suite has been executed there are still some mutant left. We refer to them as survivors. The surviving mutant indicate that there are more tests than need to be added in the suite. This is an iterative process until most of the mutants have been killed.

Equivalent Mutants Not all survivors can be killed. This is caused by mutants that output the same result as the original version. These are called Equivalent mutants. This is one of the setbacks of mutation testing, preventing it from being used more often. [Jia and Harman(2011)] The detection of these currently has to be made by humans. Because of them the mutation score can never be 100%. Studies shows that 10% of all mutants were equivalent. One attempt for detecting these mutant was made by using a fitness function that has poor values for equivalent mutants. By having a poor fitness function the mutants would be killed during the co-evolution process. This way only the mutants that are good at detecting faulty tests are kept for further study.

Mutation Score This metric is used to characterize the test suite in the end. This is the ratio between the number of the number of detected faults over the total number of the seeded faults. [Jia and Harman(2011)]

Traditional Mutation Testing

Traditional Mutation Testing is concerned with simple mutant considering them to be enough to simulate all faults. This is based on the Competent Programmer Hypothesis and the Coupling Effect.
The Competent Programmer Hypothesis assumes that programmers develop almost correct versions of the wanted system. Therefore the existing fault are simple and syntactic.
The Coupling Effect considers that a complex mutant can be broken down into simple mutants. If the simple mutant are resolved than the complex mutant is implicitly killed as well.[Jia and Harman(2011)]

We consider traditional mutation testing to be useful in isolated cases. We consider that test suites should also detect misunderstanding of the programmer regarding the requirements. These misunderstandings could be implemented in a correct syntactical way, however the system still does not behave as intended.

Mutation Testing Effectiveness

Are mutants a valid substitute for real faults in Software Testing? In this paper
[Just et al.(2014)Just, Jalali, Inozemtseva, Ernst, Holmes, and Fraser]

it is investigated whether mutants in Software Testing are a valid substitute for real faults that developers have fixed after that. For their research, they used 357 real faults in 5 open source applications, comprising a total of 321 000 lines of code. They also use both automatically generated and developer written test suites. They have three main research questions:

- Are real faults coupled to mutants generated by commonly used mutation operators?
- What types of real faults are not coupled to mutants?
- Is mutant detection correlated with real fault detection?

They use Major mutation framework to create mutant versions and perform mutation analysis. The framework provides some main mutation operators: Replace constants, Replace operators, Modify Branch Conditions and Delete statements. Classes were mutated only of the version 2 of the code and only if the same were modified by the bug fixer. For the developer written and automatically generated tests, the framework computes a mutation score. A test detects a mutant if the test outcome indicates a fault. Regarding the research questions they show results as follows:

- 73% of the read faults are coupled to the mutants, generated by commonly used mutation operators. When controlling for code coverage, on average 2 mutants are coupled to a single real fault. Also the Conditional operator replacement, relational operator replacement and statement deletion mutants are more often coupled to real faults than other mutants
- 27% of the real faults are not coupled to the mutants, generated by the commonly used mutation operators. 17% of the real faults, incl. algorithmic changes or code deletion are not coupled into any mutants. Other real faults that are not coupled are: Similar method called, context sensitivity, numerical analysis errors, specific literal replacements.
- Mutant detection is positively correlated with real fault detection, independently of code coverage. This correlation is stronger than the one between statement coverage and real fault detection. ...

Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria Mutation generation generates possible mutants based on a set of mutation operators
[Andrews et al.(2006)Andrews, Briand, Labiche, and Namin], which facilitate the analysis of fault detection. The reason is, in reality, an appropriate size with real faults are hard to find and hard to prepare appropriately, or the faults are not numerous enough. The paper investigate if empirical results obtained this way lead to valid, representative conclusions.

Research Question

The main question of the paper is: Is mutation analysis sufficient as a predictor of faults?

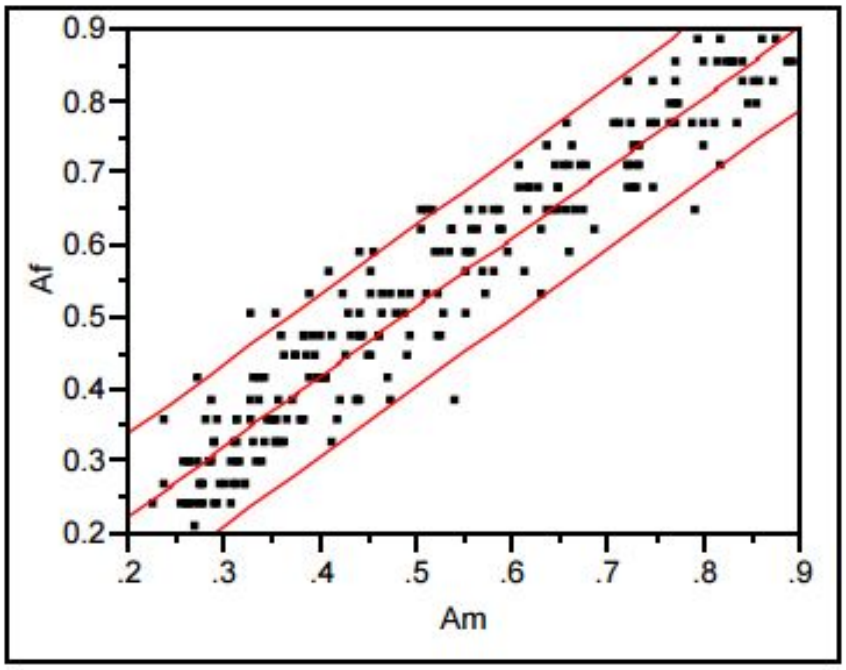
Experiments

As an experiment they used a program with 34 faulty versions. On the original version, do mutation generation: 4 classes of mutation operator, replace constant, replace arithmetic/logic operator, negate decision in if/while, delete a statement. 11379 mutants are generated, 10% used, 736 acted actually faulty behavior. Test suites generate from test pool based on the wanted coverage rate. Random suites are also generated.

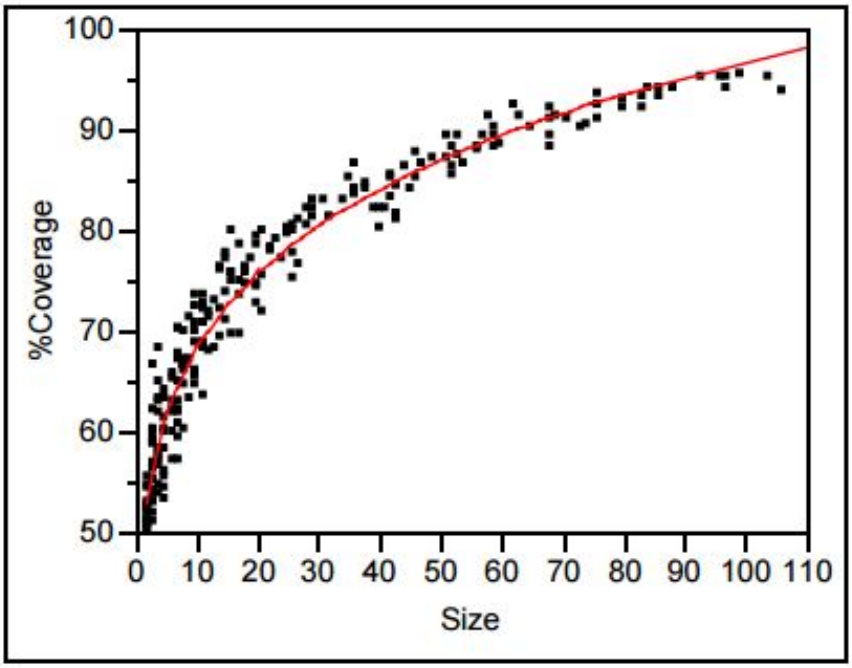
Results

Their results are as follows:

1. Mutation score predicts actual fault of detection rates



2. The higher the coverage criteria is, the more expensive it is (test suits size)



Cost-effectiveness of coverage Criteria, more cost, more efficient (more faults detected), however the size of test suites may not be a perfect representation of cost.

3. The test suites generated according to a coverage rate are more effective than the random generated suites (cost-effectiveness view)

4. Size and coverage rate both contribute to the mutation detection ratio. ...

All-Uses vs Mutation Testing: An Experimental Comparison of Effectiveness In this paper [Frankl et al.(1997)Frankl, Weiss, and Hu] the authors perform an experiment to compare the effectiveness of mutation testing and all-uses test data adequacy criteria at various coverage levels for randomly generated test sets. We choose to insert this paper because it shows how mutation testing perform better than other testing methods even though it is more expensive.
The experiment was done in order to let testers make an informed decision based on performance and costs when choosing the testing method.
The nine programs selected for this experiment were ones with subtle natural faults that can be exposed by relatively few test cases. There were added tests for the inequivalent mutants that were not killed previously. The results are mixed. However mutation testing had better result overall. The highest coverage level for 5 out of 7 subjects. Determining equivalent mutants was harder than identifying unexecutable duas. Mutation testing was also costly because of the human effort needed to check the equivalent mutants. Mutation testing has a small probability of achieving 100% coverage. This means that randomly selecting mutants is not an effective method. However the better mutants are more expensive.

Problems of Mutation Testing

High computational cost this is due to the high number of mutants against a test case[Jia and Harman(2011)].

Human effort required for the human oracle problem and the equivalent mutant problem described previously. The oracle problem refers to checking if the output is the same as for the original version. Equivalent mutant stay alive but human analysis is required for them to be labeled.

Mutant Reduction

In an attempt to solve one of the main problems for mutation testing, several method were proposed to minimize the number of necessary mutants.

Sampling

implies randomly selecting x% of the total possible mutants. Research shows that this apporach is valid with x% higher than 10%.

Clustering

chooses a subset of mutant selected from different clusters. Each cluster contains mutants than can be killed by a similar set of test cases. Research shows that the mutation score is maintained.

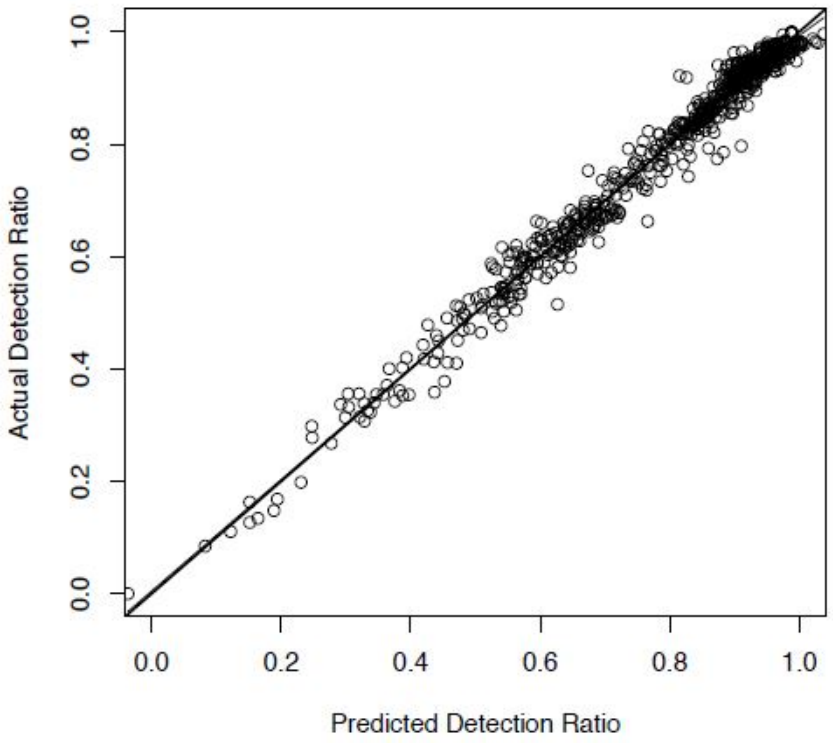
Selective

implies reducing the number of operators applied. Some operators generate more mutants than others. The selection is based on test effectiveness.
This method can be applied at unit, integration or specification level, as well as multiple programming languages. The number of potential faults is too big to be entirely reproduced. Therefore the mutants must be selected.
Usually the changes made to the original version include modifying variables and expressions by replacement, insertion or deletion of operators. [Jia and Harman(2011)]

Sufficient Mutation Operators for Measuring Test Effectiveness Mutation analysis can be used to measure test effectiveness [Siami Namin et al.(2008)Siami Namin, Andrews, and Murdoch]. However, it can be expensive to calculate, due to the large number of different mutation operators that have been proposed for generating the mutants. This paper talks about a small set of mutation operator can be sufficient enough to measure test effectiveness.

Research Question

There are 118 possible mutation operators for generating mutations, however it is very expensive to use such a big amount of operators, is it possible to find a few amount of operators which are sufficient enough to measure test effectiveness?



Experiments

The experiment when as follows: Choose Siemens Programs, which has a large pool of diverse test cases. Authors dont consider equivalent mutants, only non-equivalent. 108 mutation operators in total, 100 test suites for each program. Least Angle Regression (LARS) are used to generate subsets of operators (out of the 108), compare the predicted mutation adequacy ratio to the real mutation adequacy ratio (by building a linear model), subset operators close the model with a R-value of 0.98 or more. 28 operators are used, 7.4% of the original size.

Result:

Name	Description	Coefficient	NumMut
IndVarAriNeg	Inserts Arithmetic Negation at Non Interface Variables	0.162469	126
IndVarBitNeg	Inserts Bit Negation at Non Interface Variables	0.168583	121
RetStaDel	Deletes return Statement	0.051984	67
ArgDel	Argument Deletion	-0.016532	12
ArgLogNeg	Insert Logical Negation on Argument	0.131566	30
OAAN	Arithmetic Operator Mutation	0.041376	71
OABN	Arithmetic by Bitwise Operator	-0.02075	27
OAEA	Arithmetic Assignment by Plain Assignment	-0.194022	2
OALN	Arithmetic Operator by Logical Operator	0.022149	40
OBBN	Bitwise Operator Mutation	-0.023452	3
OBNG	Bitwise Negation	-0.00275	6
OBSN	Bitwise Operator by Shift Operator	-0.035337	3
OCNG	Logical Context Negation	0.097971	57
OCOR	Cast Operator by Cast Operator	0.024727	9
Oido	Increment/Decrement Mutation	0.069952	14
OLAN	Logical Operator by Arithmetic Operator	0.027124	119
OLBN	Logical Operator by Bitwise Operator	-0.00362	67
OLLN	Logical Operator Mutation	0.041048	25
OLNG	Logical Negation	0.06966	78
OLSN	Logical Operator by Shift Operator	-0.003438	45
ORSN	Relational Operator by Shift Operator	0.160376	91
SGLR	goto Label Replacement	0.07605	1
SMTC	n-trip continue	0.031519	9
SMVB	Move Brace Up and Down	-0.062404	2
SSWM	switch Statement Mutation	-0.020412	15
STRI	Trap on if Condition	0.094324	85
SWDD	while Replacement by do-while	0.032363	1
VGPR	Mutate Global Pointer References	-0.091281	13
(Intercept)		-0.036398	

Higher Order Mutation

aims at finding the rare mutants that denote subtle faults. Research shows that by using second order mutants it reduced the test effort with 50%. The effectiveness of the test case was not very affected. The authors of the method proposed certain ways of combining first order mutants to generate the second order ones. This way they have found high order mutants harder to kill than any of the first order ones.

Cost Reduction Techniques

Strong Mutation assumes that a mutant is killed only if it gives a different output than the original program.

Weak Mutation checks mutants after the execution of the mutated component. This way resources are spared by not executing everything. However they can be less effective as the strong mutation approach.

Firm Mutation is based on the compare state and represent a hybrid version. It takes place between after the execution of the mutation and the final output.

Runtime optimization infers the result of a mutant form the source code directly.

Mutation testing framework

Developing tools for mutation testing is the only thing that can bridge the lab version of mutation testing with the real industry. The paper "Mutation analysis using Mutant Schemata" was chosen as it provides a different view and method to test code [Untch et al.(1993)Untch, Offutt, and Harrold]. It searches for an alternative method to improve performance which is important for software testing. The new method for mutation analysis includes usage of the so called "program schemata" to encode all the mutants for a program and to incorporate them into one metaprogram which is compiled and ran at a higher speed.

The paper proves that the program schemata gives results of over 300% better performance, it is easier to implement than interpretive systems. It is also easier to move across as it uses the same compiler and runtime support, used during its development and/or deployment. Mutant Schema Generation (MSG) systems produce a compilable program in the same language as the program that is being tested, the test takes place using the same compiler, used for the tested program.

Another advantages of the MSG systems allows mutants to be executed at compiled speeds without recompilation or storage need for every mutant. These systems are cheaper to build. The authors manually generated a meta mutant for the Newton procedure in the figure below. They also manually generated a list of mutant descriptors. Then they produced 385 mutants.

Additionally, a elementary library of meta procedures and a driver were developed and implemented to make a working MSG mutation analysis system. They then compared the speed of their newton meta mutant with the speed of testing the newton in a Methra Software Testing Environment. Their comparison revealed that mutation analysis performed by direct execution of the meta mutant was 4.1 times faster than interpretive execution.

```
 1   PROCEDURE Newton(Number:REAL; VAR Sqrt:REAL);
 2   (* Find square root using Newton's method. *)
 3   VAR
 4       NewGuess, Delta, Epsilon : REAL;
 5   BEGIN
 6       Epsilon :=  0.001;
 7       NewGuess :=  (Number / 2.0) + 1.0;
 8       Sqrt :=  0.0;
 9       Delta :=  NewGuess - Sqrt;
10       WHILE Delta > Epsilon DO
11           Sqrt :=  NewGuess;
12           NewGuess := (Sqrt+(Number/Sqrt))/2.0;
13           Delta :=  NewGuess - Sqrt;
14           IF Delta < 0.0 THEN
15               Delta :=  -Delta;
16           END;
17       END; (* END WHILE *)
18   END Newton;
```

Conclusion

Even though Mutation testing has its drawback, "we should be careful not to throw the baby out with the bath water" [Jia and Harman(2011)]. Mutation testing is promising and progress is expected to be made in the generation of test cases that kill survivor mutants.

References

[Jia and Harman(2011)] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on*, 37(5):649–678, 2011.

[Just et al.(2014)Just, Jalali, Inozemtseva, Ernst, Holmes, and Fraser] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 654–665. ACM, 2014.

[Andrews et al.(2006)Andrews, Briand, Labiche, and Namin] James H Andrews, Lionel C Briand, Yvan Labiche, and Akbar Siami Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *Software Engineering, IEEE Transactions on*, 32(8):608–624, 2006.

[Frankl et al.(1997)Frankl, Weiss, and Hu] Phyllis G Frankl, Stewart N Weiss, and Cang Hu. All-uses vs mutation testing: an experimental comparison of effectiveness. *Journal of Systems and Software*, 38(3): 235–253, 1997.

[Siami Namin et al.(2008)Siami Namin, Andrews, and Murdoch] Akbar Siami Namin, James H Andrews, and Duncan J Murdoch. Sufficient mutation operators for measuring test effectiveness. In *Proceedings of the 30th international conference on Software engineering*, pages 351–360. ACM, 2008.

[Untch et al.(1993)Untch, Offutt, and Harrold] Roland H Untch, A Jefferson Offutt, and Mary Jean Harrold. Mutation analysis using mutant schemata. In *ACM SIGSOFT Software Engineering Notes*, volume 18, pages 139–148. ACM, 1993.