Hierarchical Higher Order Mutant Classification

CS580 Term Paper
Alex Fout, Colorado State University Department of Computer Science
05/03/2016

Abstract

Mutation testing involves automatically generating altered versions (mutants) of a program and improving the test suite to distinguish the output of the mutants and the original program (i.e. kill the mutants). Unfortunately, some mutants are functionally equivalent to the original program and therefore can never be distinguished. Manual inspection of mutants to determine if they are equivalent is time consuming, so automated mutant classification techniques have been developed. One existing method, Higher Order Mutant (HOM) classification, involves combining unkilled (i.e. unclassified) mutants with mutants that have already been killed and observing if the test suite can distinguish the output of the combined mutant from the killed one alone. This method is effective but entails a higher number of tests in order to classify mutants. This project builds upon the concept of HOM classification by constructing a mutation tree, where mutants are combined in pairs successively until a single mutant at the root of the tree contains all mutations. The mutants in the tree are executed beginning at the leaves (which are unclassified mutants), and classification rules similar to the HOM classifier are applied until all unclassified mutants are either possibly killable or possibly equivalent. Three experiments were conducted on a simple eight class Java program and the results of HOM classification and the new approach (HHOM classification) were compared. The HHOM classifier uses significantly fewer tests than the HOM classifier, but tends to over classify mutants as possibly killable, particularly for the larger class tested. A number of extensions are proposed that could improve HHOM classifier performance while still limiting the number of tests.

Keywords: mutation testing, equivalent mutant detection, mutant classification, higher order mutation

1. Introduction

The ability to detect faults in a program is a reflection of the effectiveness and quality of a test suite. Testers need ways to determine how to improve the quality of a test suite by removing, refining, and adding tests to the test suite. This is commonly performed by applying the tests to programs with known faults and refining the tests to detect these faults. These faults can be generated manually, though that is time consuming and therefore not scalable. Alternately, faults could be injected into a program automatically. One such process is mutation.

Mutation testing involves generating *mutants* from an original, correct version of a program. Faults are injected using *mutation operators* which are rules for chaning the original program. Mutation operators are intended to “represent the mistakes that programmers often make” [2]. Intuitively, a higher quality test suite would be able to distinguish between the original program and the mutant, whereas a lower quality test suite would not be able to make the same distinction. Formally, a mutant is *killed* (strongly) by a test if the output of that test differs between the original program and the mutant [1, p178]. Given a set of mutants, a test suite can be given a mutation score, which is the fraction of mutants killed by any test in the test suite [1,

p181]. A test suite can be improved by making it kill more mutants and therefore giving it a higher mutation score [1, p181].

In principle, mutants may contain any number of mutations, but in practice mutants for test suite refinement are typically limited to a single mutation (known as *simple mutants, first order mutants,* or *FOMs*). This approach is motivated by two assumptions: the Competent Programmer Hypothesis (CPH) and the Coupling Effect (CE). CPH states that faulty programs created by programmers are usually "close to the correct version" [2]. CE states that "a test data set that detects all simple mutants in a program will also detect a large percentage of the complex mutants" [7, quoted in 2]. Under these assumptions, modifying a test suite to kill FOMs reflects the idea that faults are usually small, and that tests which detect smaller faults will also detect larger ones.

Since mutant creation is an automatic process, some mutants might be functionally equivalent to the original version of the program. Such mutants are by definition not killable and denoted *equivalent*. Equivalent FOMs can usually be identified upon visual inspection by a programmer, but this does not scale well. The general problem of detecting equivalent mutants algorithmically is undecidable [2], but several heuristics have been developed to approximate detection [3][8][4]. Multiple heuristics attempt to classify mutants as either possibly killable or possibly equivalent. Allowing testers to focus on refining the test suite to kill those mutants in the possibly killable class, disregarding the mutants which are possibly equivalent.

One mutant classification method called Higher Order Mutation (*HOM*) classification uses second order mutants (or *SOMs*) to detect mutant equivalence as follows[4]: Unkilled mutants are considered unclassified mutants (*UMUT*s) that may be killable or may be equivalent. A set of classification mutants (*CM*s) is created for each unclassified mutant, and a classification mutant is combined with the unclassified mutant to create a SOM. Each SOM is tested and the output is compared against the output of the classification mutant. If the outputs differ, then the unclassified mutant is considered possibly killable, otherwise it is considered possibly equivalent. Classification mutants may be created using all FOMs, only FOMS which have been killed, or only FOMs for which the mutation is in the same method as the UMUT.

Given multiple CMs, an UMUT is classified as possibly killable if *any* of the CMs yields a different output from the associated SOM. Thus, a larger CM set increases the likelihood of classifying a mutant as possibly killable. Depending on the number of classification mutants used, this approach may not scale well to real world situations. If there are n UMUTs, m CMs, and k tests in the test suite, then there are $O(mnk)$ tests in the worst case. If $m = O(n)$, this becomes $O(n^2k)$.

The purpose of this project is to develop and evaluate a new mutant classification method which effectively detects equivalent mutants and scales more effectively than HOM classification. The goal is a new method which performs as well as existing methods while employing fewer overall tests. An approach with fewer overall tests is more feasible in a real world situation where programs and test suites are comparatively large.

2. Approach

Hierarchical Higher Order Mutant Classification

CS580 Term Paper

Alex Fout, Colorado State University Department of Computer Science

05/03/2016

This project proposes a hierarchical method for mutant classification called Hierarchical Higher Order Mutation classification (or HHOM classification). This approach is similar to the HOM classifier in that it uses higher order mutants to identify equivalent FOMs. Instead of designating a separate set of CMs, UMUTs are used to classify each other. Alternately, the UMUTs themselves could be viewed as the CM set. More specifically, a mutation tree is created by successive pairwise combinations of mutants. The test suite is executed against each mutant in the mutation tree, and the output of each compared in order to classify the UMUTs.

The mutation tree is constructed as follows. The first level of the tree consists of all unclassified FOMs. The mutants at this level are (randomly) paired and combined to create SOMs which constitute the second level of the tree (each set of “parent” FOMs produces a single “child” SOM). Analogously, these SOMs are (randomly) paired and combined to create fourth order mutants at the third level of the tree, etc. until there is a single mutant in the last level of the tree. Two parents are required to create each child in the next level, so if a level has an odd number of mutants, then one of the mutants is duplicated in order to create an even number of parents. The duplicate mutant is not paired with itself when creating children. Each level of the tree contains roughly half the number of mutants as the previous level, hence the total number of mutants in the tree is $O(n)$, and the total number of tests run is $O(nk)$.

When UMUTs are combined to create a higher order mutant, each UMUT serves as a classification mutant for the other UMUT. Hence, the output of the child mutant can be compared to the output of each UMUT in order to determine if the *other* UMUT is killable. Concretely, consider two UMUTs, A and B, which are combined to create a child mutant AB. If the output of AB is different than the output of A, then the addition of B’s mutation changed the output of A, hence B can be classified as possibly killable. The reasoning for this is that if B’s mutation changes the output of A, then AB is not equivalent to A. Since A is very similar to the original program, then the same mutation applied to the original program may also produce a mutant (B) which is not equivalent to the original program. If the output of both parents is different than the output of the child, then both parents can be classified as killable.

If the output of both parents is the same as the output of the child, then one approach would be to classify each mutant as possibly equivalent. However, recall that for HOM classification several CMs are used to create higher order mutants with an UMUT. Even if several CM’s yield SOMs with identical output to the CM, the UMUT may still be classified as possibly killable if another SOM has output different from its CM. In keeping with this idea, HHOM classification does not immediately classify two parent mutants as possibly equivalent if the output of both is the same as the child. Instead, classification is dependent on the results at higher levels in the tree. Whenever a mutant is classified as possibly killable, then all unclassified ancestors of that mutant can also be classified as killable.

This is formalized in the following algorithm for HHOM classification:

1. Run the test suite against the original program to create a set of unkilled (unclassified) mutants. Create the mutation tree. If any layer (other than the last layer) has an odd number of mutants, duplicate a mutant at random to have a complete pairing (do not pair the duplicate with itself).

2. For each mutant "family" of two parents and one child:
   If both parents are already classified, do nothing.
   Else, apply the test suite to the parents and children as needed.
   a. If the output of both parents is different from the child, then both parents are possibly killable. All unclassified ancestors of the parents are also possibly killable.
   b. If the output of both parents is the same as their child, then do not classify the parents yet, unless the child is the last node, in which case the parents are possibly equivalent. In such case, all unclassified ancestors of the parents are also possibly equivalent.
   c. If the output of one parent is the same as its child, but the output of the other parent is different from its child, then the former parent (and its ancestors) is possibly equivalent and the latter parent (and its ancestors) is possibly killable. *Note: this will not occur for the first level of the tree since the output of all unclassified mutants is the same as the original program*, *hence the output of all umuts is the same.*

   Step 2 is complete when all FOMs are classified, which may not require testing all mutants in the tree.

3. Evaluation

Experiment

The experiment conducted for this project involved generating first order mutants for a test program given a manually created test suite. The unkilled mutants were classified using both HOM classification and HHOM classification, and each classification technique was evaluated by examining the confusion matrix, precision, recall, accuracy, and F-score. The experiment was repeated with three different sets of test mutants. The first two classes (Bishop and Rook) contain similar code, and produce a similar number of mutants. The third class (Pawn) contains more code and produces more than twice the number of mutants as the first two classes.

The program under test is based on a chess themed programming assignment from a freshman level Java/OOP course. There are eight Java classes representing different types of chess pieces and a chess board (see figure 1 for class hierarchy). The assignment was graded using an automated grading system which compares student's programs to a "master" version written by a TA using tests also written by the TA. For this project, the program under test was derived from the master program and the test suite was derived from the set of tests used for grading. The program under test passed all test cases. The original master program and grading tests were written by Hajar Homayouni, Colorado State University. No student code was used for this project. The test suite consists of 51 Junit tests which test that chess pieces are correctly placed on the board, print correctly, and return the correct legal moves for various board configurations.

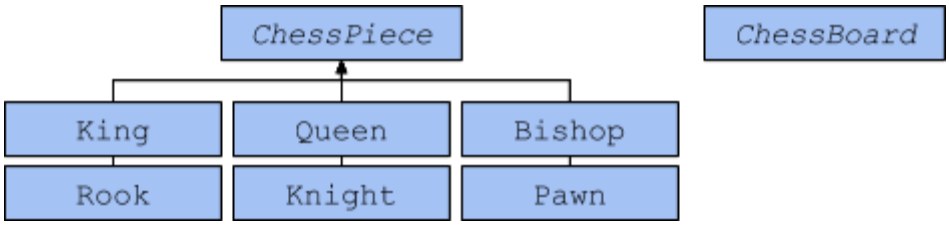

Figure 1, Class diagram for program under test

FOMs were created with muJava[6] using all available traditional mutation operators. Although FOMs were created for each class, each experiment was run using mutants from only a single class at a time. This allowed multiple experiments to be run using the same program under test and test suite. This is also consistent with the restriction applied in [4] that CM's be drawn from the same class as the mutant being classified. Each set of FOMs was run using the test suite and marked as killed or unkilled. All unkilled mutants were manually inspected and classified as equivalent or killable to establish "ground truth". These manual classifications were used only to evaluate the performance of the classification methods. Both classifiers potentially use the output of a mutant test multiple times in the classification process. Therefore for both classifiers, test output was saved and stored, so that no mutant was executed more than once. Output from one classifier was not used in the other classifier.

Both HOM and HHOM classifiers utilize higher order mutants, but there is currently no freely available software which performs higher order mutation automatically. Therefore a custom mutation engine was developed which combines any two mutants together (of any order) to create a higher order mutant. Due to time limitations, this mutation engine is rudimentary and has two major limitations. First, it is unable to compare mutants which do not share the same number of lines of code, therefore mutations which add or remove lines to the original program were not examined. Second, it cannot combine two different mutations that are on the same line. In such case, the mutation from the first mutant (according to some ordering) is kept and the other discarded. For example, if mutant A has mutations on lines {3, 6} and mutant B has mutations on lines {6, 8}, then the combined mutant AB will have mutations on lines {3, 6, 8}, where the mutation on line 6 will match that in mutant A, since it appears first in the mutant name AB. Conversely, line 6 of mutant BA will match that in mutant B. As a result of this limitation, Classification mutants which contain mutations on the same line as the UMUT being classified were not used for classifying.

Operator	Description	Operator	Description
AOR	Arithmetic Operator Replacement	LOR	Logical Operator Replacement
AOI	Arithmetic Operator Insertion	LOI	Logical Operator Insertion
AOD	Arithmetic Operator Deletion	LOD	Logical Operator Deletion
COR	Conditional Operator Replacement	SOR	Shift Operator Replacement
COI	Conditional Operator Insertion	ASR	Assignment Operator Replacement

COD	Conditional Operator Deletion	ROR	Relational Operator Replacement

Table 1, Method-level mutation operators used, from [5]

Mutated Class (Experiment)	Total FOMs	Killed FOMs	Unkilled FOMs	
			Killable	Equivalent
Bishop	130	110	7	13
Rook	137	117	7	13
Pawn	414	209	185	20

Table 2, FOMs for each mutate class

To compare both mutant classification methods we introduce the following definitions from [4]:

1. *True Killable*: number of killable mutants that are correctly classified as possibly killable
2. *False Killable*: number of equivalent mutants that are incorrectly classified as possibly killable
3. *True Equivalent*: number of equivalent mutants that are correctly classified as possibly equivalent
4. *False Equivalent*: number of killable mutants that are incorrectly classified as possibly equivalent

Classification can be evaluated using the following metrics, where β is a weight reflecting the relative importance of recall over precision[4]:

1. $Precision = \frac{True\ killable}{True\ killable + False\ killable}$
2. $Recall = \frac{True\ killable}{True\ killable + False\ equivalent}$
3. $Accuracy = \frac{True\ killable + True\ equivalent}{True\ killable + False\ killable + True\ equivalent + False\ equivalent}$
4. $F_\beta = (1 + \beta^2) \frac{Precision * Recall}{\beta^2 * Precision + Recall}$

Precision measures the fraction of the mutants classified as possibly killable which are actually killable. A higher value gives test developers more confidence that mutants thus classified can actually be killed with new test cases. Recall measures the fraction of killable mutants that are actually classified as possibly killable. A higher value gives test developers more confidence that killable mutants are not being ignored because they are thought to be equivalent. Accuracy is an overall measure of classifier performance. A higher value gives test developers more confidence that the classifier is generating correct classifications in general.

F is a combined measure of classifier performance that incorporates precision and recall using the parameter β, which controls the relative importance of precision with respect to recall. A value of $\beta = 1$ indicates equal weighting for each, a value of $\beta > 1$ weights precision more important than recall, and a value of $\beta < 1$ weights precision less important than recall. Values of 1, 2, and 0.5 were used.

Results

Table 3 shows confusion matrices for each experiment and Table 4 shows the associated precision, recall, accuracy, and F measure.

HOM classifier Bishop	Equivalent	Killable
Possibly Equivalent	13	4
Possibly Killable	0	3

HHOM classifier Bishop	Equivalent	Killable
Possibly Equivalent	8	4
Possibly Killable	5	3

HOM classifier Rook	Equivalent	Killable
Possibly Equivalent	12	4
Possibly Killable	1	3

HHOM classifier Rook	Equivalent	Killable
Possibly Equivalent	2	2
Possibly Killable	11	5

HOM classifier Pawn	Equivalent	Killable
Possibly Equivalent	19	42
Possibly Killable	0	144

HHOM classifier Pawn	Equivalent	Killable
Possibly Equivalent	0	30
Possibly Killable	19	157

Table 3, Confusion matrices for classification experiments

		Average Tests*/UMUT	Precision	Recall	Accuracy	F(β=1)	F(β=2)	F(β=0.5)
Bishop	HOM	95.4	1.00	0.43	0.80	2.00	1.61	2.63
	HHOM	3.1	0.38	0.43	0.55	2.00	2.08	1.92
Rook	HOM	107.3	0.75	0.43	0.75	2.00	1.72	2.39
	HHOM	3.1	0.31	0.71	0.35	2.00	2.61	1.62
Pawn	HOM	73.4	1.00	0.77	0.80	2.00	1.86	2.17
	HHOM	2.5	0.89	0.84	0.76	2.00	1.96	2.03

Table 4, Metrics for each classifier in each experiment.
*A test refers to running entire test suite on a mutant

Discussion

Across all three experiments, the HOM classifier performs consistently. The confusion matrices for Bishop and Rook are nearly identical. Most mutants are either true killable or true equivalent, which is reflected in the high accuracy value. The raw number of false killable mutants is low for all three experiments, which could save a programmer time trying to write test cases to kill equivalent mutants. This is particularly noteworthy for the Pawn class, which has

perfect precision, even with significantly more mutants being classified. The recall for the HOM classifier is markedly lower than precision since a significant portion of killable mutants are classified as possibly equivalent. The high precision and lower recall could be regarded as "conservative" in the sense that there is a high likelihood that possibly killable mutants are actually killable, but there is a somewhat low likelihood that all killable mutants have been identified.

The HHOM classifier appears to trade off precision for recall compared to the HOM classifier. In all three experiments, it has a lower precision and higher recall. More killable mutants are being classified as killable, which affords the programmer more opportunities to improve the test suite. The drawback is that there are more equivalent mutants mixed in with the truly killable ones, which require time to recognize and disregard. This classifier can be viewed as a more "aggressive" classifier, since more mutants are labeled possibly killable, but there are more incorrect classifications. This can easily be seen in the significantly lower accuracy scores. Notably, none of the equivalent mutants are being correctly classified for the Pawn class. From a programmer's perspective, this provides little utility in effectively reducing the number of mutants, since the only mutants discarded as equivalent are actually killable, and none of the equivalent mutants have been discarded.

Nonetheless, the precision, recall, and accuracy all increase for the Pawn class, and it is possible that this trend would continue for larger classes. The current HHOM classification rules allow that any descendent of an UMUT could classify that mutant as possibly killable, so one would expect the likelihood of classifying a UMUT as possibly killable to increase with the size of the tree. This is in turn affected by the number of UMUTs. It may be possible that the increased accuracy of the HHOM classifier for the Pawn class is due to its aggressive classification as possibly killable, coupled with the increased proportion of UMUTs which are killable.

The greatest difference between the two classifiers is the number of program/mutant executions required to classify a mutant. In the worst case, the HOM classifier creates, compiles, and executes a SOM for each pairing of CM mutant and UMUT. In particular, the worst case occurs every time a CM is classified as possibly equivalent, which is a large proportion of classifications. In contrast, the HHOM mutation tree contains only roughly $O(n)$ mutants, where n is the number of UMUTs. Therefore, whereas the HOM classifier performs roughly 70-100 tests per UMUT on average, the HHOM classifier requires only about 3 tests per UMUT. This is a very significant improvement in testing time. Interestingly, even with significantly fewer tests, the HHOM classifier still classifies more mutants as killable.

Threats To External Validity:

The program under test for this project is likely not representative of industry Java applications. There are only a few classes and the inheritance hierarchy among them is simple. However, both the HOM and HHOM classifier are local in nature, since all mutants are restricted to a single class at a time. Therefore such differences may not threaten external validity. The percentage of equivalent mutants in the Pawn class is consistent with [2], which gives a range

of 10-40%. The Bishop and Rook classes exhibit a higher percentage than expected, but this percentage is derived from only 20 UMUTs.

The class sizes may not be representative of classes used in industry. As discussed in the results section, the number of mutants in the tree may affect the behavior of the HHOM classifier, so it is difficult to generalize the results of these experiments to a class of any size.

The performance of both the HOM and HHOM classifiers depend on the test suite. Not only does the test suit determine which mutants are killed and unkilled, it also is the primary tool used to classify UMUTs. A weaker test suite may leave more mutants unkilled, and would perhaps be less prone to classify UMUTs as possibly killable. This project is limited to only a single test suite applied to a single program under test, so it is difficult to predict how these classifiers would perform in different conditions from the results of this project alone.

Threats To Internal Validity

The change in behavior of the HHOM classifier observed for the Pawn class could be influenced by multiple factors. Not only is the UMUT set much larger for the Pawn class, but the fraction of those UMUTs which are killable is significantly larger as well. The increased number of UMUTs creates a larger mutation tree, which could impact the behavior of the classifier, and the make up of the UMUT set also may be more amenable with the HHOM's tendency to more aggressively classify mutants as possibly killable.

Threats To Construct Validity

Each metric is a direct measure of classifier performance with the given test suite for the program under test. There are no existential threats to construct validity.

4. Related Work

This project compares the novel approach with the Higher Order Mutant classifier introduced by Kintis et al in [4], but there exist other mutant classification schemes as well. In the same paper, Kintis et al propose three options for the CM set, namely using all FOMs, only killed FOMs, and all FOMs in the same method as the UMUT's mutation. The latter option shows inferior performance in their experiments, and the former two show comparable performance, while the killed FOM option contains fewer mutants and therefore entails fewer tests. This project compares only against the killed FOM option, since it can be viewed as the most effective choice which uses the fewest number of mutants.

Papadakis et al propose mutant based classification using the concept of coverage impact[8]. Coverage impact involves running a set of test cases against an original and mutant program and recording the trace for each execution. Any line of code for which the number of executions in the original program differs from the number of executions in the mutant has been impacted by the mutation. The number of such lines that have been impacted indicates the degree of impact. Impact can either be measured using the entire program (whole impact), or by looking only at non-local (different method from the mutation) portions of the program (non-local

impact). They report higher precision for non-local-impact classification but higher recall for whole impact classification. Both approaches perform better than random classification.

Kintis et al also evaluate a two step classifier which first uses a coverage impact classifier, then feeds all possibly equivalent mutants into the HOM classifier which further classifies some as possibly killable[4]. They saw a statistically significant increase in recall when using the two step classifier compared to using either individual classifier alone. This project did not compare results with the two step classifier, instead it is proposed that an analogous classifier could be constructed using the coverage impact classifier paired with the HHOM classifier.

Another classification method relies on “state infection conditions” which determine whether a mutant is killable[3]. Constraints are generated which capture the necessary conditions for a mutant to reach an infected state (i.e. different from the original program). A constraint solver is then used to determine if the constraints can simultaneously be satisfied. If the answer is yes, then the mutant is classified as possibly killable. If the answer is no, then the mutant is known to be equivalent, since state infection is necessary for a mutant to be equivalent. If the constraint solver cannot produce an answer or the infection conditions cannot be determined, then the mutant is considered possibly killable. Unlike other classification techniques, this approach does not require executing the mutants before they can be classified. Effectiveness is measured not using precision/recall, but instead by observing the decrease in the number of mutants that need to be executed, since equivalent mutants can be removed prior to execution.

5. Conclusions

Mutant classification has seen some success in removing equivalent mutants from a test suite. However, success in a real world context requires classifier performance and scalability. The HOM classifier exhibits higher overall performance than the HHOM classifier, but the sheer number of required tests becomes prohibitive at scale. Contrapositively, the HHOM classifier is much more scalable because of the low number of required tests, but did not perform as well overall as the HOM classifier. However, since the HHOM has shown some promise with classification, there may exist modifications which can improve its performance while keeping the overall number of tests low. Some possibilities include incorporating some killed mutants into the mutation tree, altering the classification rules, or “chopping off” the top of the tree to further reduce the number of tests and to reduce the likelihood of classifying mutants as possibly killable.

The HHOM classifier also shows promise when the number of killed mutants is small. In this situation, the CM set for the HOM classifier would be small if only killed mutants are used (justified in the related work section), but the HHOM could still function as designed.

References

[1] Ammann, P., & Offutt, J. (2008). *Introduction to software testing*. New York: Cambridge University Press.

[2] Jia, Y., & Harman, M. (2011). An Analysis and Survey of the Development of Mutation Testing.*IEEE Transactions on Software Engineering IIEEE Trans. Software Eng., 37*(5), 649-678.

[3] Just, R., Ernst, M. D., & Fraser, G. (2013). Using state infection conditions to detect equivalent mutants and speed up mutation analysis. *Proceedings of the Dagstuhl Seminar 13021: Symbolic Methods in Testing*. arXiv:1303.2784, preprint.

[4] Kintis, M., Papadakis, M., & Malevris, N. (2014). Employing second-order mutation for isolating first-order equivalent mutants. *Software Testing, Verification and Reliability Softw. Test. Verif. Reliab., 25*(5-7), 508-535.

[5] Ma, Y.,, & Offut, J. (2011, December). Description of Method-level Mutation Operators for Java. Retrieved May 3, 2016, from https://cs.gmu.edu/~offutt/mujava/mutopsMethod.pdf

[6] Ma, Y., Offutt, J., & Kwon, Y. R. (2005). MuJava: An automated class mutation system. *Software Testing, Verification and Reliability Softw. Test. Verif. Reliab., 15*(2), 97-133. doi:10.1002/stvr.308

[7] Offutt, A. J. (1992). Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology TOSEM ACM Trans. Softw. Eng. Methodol., 1*(1), 5-20.

[8] Papadakis, M., Delamaro, M., & Traon, Y. L. (2014). Mitigating the effects of equivalent mutants with mutant classification strategies. *Science of Computer Programming, 95*, 298-319.7