TCE+: An Extension of the TCE Method for Detecting Equivalent Mutants in Java Programs

Mahdi Houshmand and Samad Paydar(✉)

Dependable Distributed Embedded Systems (DDEmS) Laboratory, Computer Engineering Department, Ferdowsi University of Mashhad, Mashhad, Iran
mahdi.houshmand@mail.um.ac.ir, s-paydar@um.ac.ir

Abstract. While mutation testing is considered to be an effective technique in software testing, there are some impediments to its widespread use in industrial projects. One of these challenges is the equivalent mutant problem, and a line of research is dedicated to proposing new methods for addressing this problem. Trivial Compiler Equivalence (TCE) method is recently introduced as a simple technique that actually relies only on the optimizations made by the compiler. It is shown by empirical studies that employing TCE with the *gcc* compiler results in a fast and effective technique for detecting equivalent mutants in *C* programs. However, considering the fact that the Java compilers generally do not perform noticeable optimizations, the question is how effectively does TCE perform on Java programs? In this paper, experimental evaluations are discussed which demonstrate that using TCE technique with *javac* compiler results in very poor performance. As a result, this paper proposes to use the Java obfuscators as the complementary component, because of the optimizations they make. The experimental evaluations confirm that using TCE with the *ProGuard* obfuscation tool provides an effective and efficient method for detecting equivalent mutants in Java programs.

Keywords: Mutation testing · Equivalent mutant · Trivial compiler equivalence · Java

1 Introduction

Mutation testing is considered to be an effective approach to evaluate and also to improve an existing test set [1]. It works based on the notion of mutants, where each mutant is created by making a simple modification on the program under test. The set of possible modifications are defined by the mutation operators that are defined for the programming language of the target program. If there is a test set that the program has successfully executed on, then mutation testing can be applied to provide a measure of the quality of that test set. This is performed by running each mutant M on the test cases to investigate whether the test cases are powerful enough to detect the injected fault, i.e. the mutation. If the result of running the mutant on a test cases is different from the result of running the original program on that test case, then the test case has been able to distinguish, or kill, that mutant. The greater ratio of the mutants of the

M. Dastani and M. Sirjani (Eds.): FSEN 2017, LNCS 10522, pp. 164–179, 2017.
DOI: 10.1007/978-3-319-68972-2_11

program are killed by the test set, the higher is the score of that test set. Finally, if there remains any live mutant, i.e. mutants that are not killed by any test case, then there are two possible cases for each live mutant: (1) whether this is a sign of the weakness of the test set, or (2) the mutant is an equivalent mutant, i.e. the corresponding mutation has made a syntax change without changing the semantic, and hence, the mutant cannot be killed by any test case.

When applying mutation testing, a method is necessary to distinguish which of the above cases holds for a live mutant. Without differentiating these two cases, it is possible that the test case designer wastes his time and effort in trying to find a test case for killing an equivalent mutant, which is actually not killable. Further, an equivalent mutant may cause the quality of the test set to be underestimated.

While mutation testing has been empirically proven to be able to simulate real-world programming errors [24], and hence to be an effective method for evaluating and improving test sets, there some non-negligible impediments towards its application in industrial software. The first problem is that mutation testing is a costly method, since the number of possible mutants, even for a relatively small program is usually high. Creating the mutants, compiling and executing them over the test cases and comparing the execution result usually requires noticeable time and computation resources.

Another problem is the equivalent mutants introduced before. Consequently, different approaches have been introduced during the last two decades for addressing this problem by employing different techniques like machine learning [14], logical constraint solving [15], data flow pattern analysis [8], gamification [17], program slicing [10] and code similarity measures [13]. One of the approached introduced recently, is the Trivial Compiler Equivalence (TCE) approach [12] which is a simple, fast and effective technique for detecting equivalent mutants.

The TCE technique actually relies on the optimizations performed by the compiler, and it tries to determine equivalence of a mutant by comparing it with the original program, in their binary, i.e. compiled, format. TCE has been evaluated in [12] on C programs using the *gcc* compiler that is capable of performing different levels of optimizations when compiling the program. The evaluations have shown that TCE is an effective method for equivalent mutant detection in C programs. Considering Java programs, however, TCE is not expected to perform noticeably, since the Java compiler performs almost no specific optimization, and it leaves the optimizations to be performed by Java Virtual Machine at runtime (JVM) [26]. We believe there is room for evaluating the TCE technique on Java programs. Hence, in this paper, we experimentally evaluate performance of TCE on Java programs, and further, we introduce TCE+ as an extension of TCE which utilizes the ProGuard[1] Java obfuscator in addition to the compiler to address the lack of compiler optimizations.

The rest of the paper is organized as follows. Section 2 briefly reviews the related works. In Sect. 3, the experimental evaluation of the TCE and TCE+ techniques on Java programs is discussed. Finally, Sect. 4 concludes the paper.

[1] http://proguard.sourceforge.net/.

2 Related Work

In order to address the equivalent mutant problem in the mutation testing domain, different approaches have been proposed during the last two decades. This problem, in its general form is an undecidable problem [2, 3] and therefore it is not expected to be able to find an automated method that can solve every instance of this problem correctly and completely. As a result, some of the proposed approaches employ heuristics or limit the characteristics of the program under study, for instance restricting the number of iterations of the loops [25]. A literature review on the approaches for tackling with the equivalent mutant problem is provided in [4], where it is concluded that the equivalent mutant detection techniques are still *"far from perfect"*.

Some works attempt to deterministically determine whether a specific mutant is equivalent or not. For instance, in [8, 18] a set of 9 data flow patterns is introduced that result in equivalent mutants. In addition, a framework is proposed which uses static analysis of data flow to check each mutant of a program against these patterns. If a mutant follows one of the predefined patterns, then it is equivalent, otherwise it is considered to be non-equivalent. As another example, [15] introduces a technique that extracts a set of logical constraints from a mutant such that solving those constraints proves that the mutant is equivalent to the original program. Then, the constraints are given to a constraint solver tool for the purpose of detecting equivalent mutants. The method assumes certain characteristics on the mutants which limits applicability of the method (e.g. recursive functions are not supported). A similar approach based on constraint solving techniques is also introduced in [16].

Some works implicitly use the idea that for an undecidable problem, it is not possible to provide a complete automated solution and hence human intervention is unavoidable. Therefore, they try to help the human experts in analyzing the mutants and in making decision about their equivalence. This help can be provided in form of identifying the mutants that are more likely to be equivalent. Therefore, these methods follow a inexact approach and generate a recommended list of mutants, ordered by their equivalence probability, that need to be manually analyzed by the human expert to make the final decision. For instance, in [11], the idea is that the probability that a mutant is not equivalent is related to how its coverage on a specific test set differs from the coverage of the original program. In other words, the greater the coverage is affected, the lower is the probability of the mutant being equivalent. A similar approach for determining equivalent mutants based on the coverage impact is also proposed in [5, 6]. Machine learning techniques are also used in some works like [14] to provide a probabilistic approach to detection of equivalent mutants.

Another example of the works that count on human involvement for detection of equivalent mutants is [17] that uses gamification technique. It introduces a two-player game in which one player tries to create mutants that are hard to kill, and the other one tries to introduce test cases that kill the mutants. The game indirectly can contribute to detecting mutants that are more likely to be equivalent.

Another group of works try to avoid creation of equivalent mutants by more advanced mutation generation techniques. For instance, [19] proposes to consider the fact that different mutation operators perform differently from the point of view of the

difficulty of killing their resulting mutants. This can be employed to selectively use mutation operators that less frequently create equivalent mutants. Another group of works have shown that using higher order mutants instead of first-order mutants can reduce the number of equivalent mutants generated for a program [9, 20–22].

Other techniques that have been used for exact equivalent mutant detection include code similarity measures and clone detection techniques [13], program slicing techniques [10], co-evolution algorithms [7].

An interesting approach that is recently proposed for detection of the equivalent mutants is the TCE approach [12], which uses a very simple and straightforward technique. TCE works based on the idea that the advanced optimizations performed by a compiler can remove some type of the mutations that have not affected the semantic of the program, and hence if the equivalent mutant is compiled, the result of compiling can be the same as the result of compiling the original program. It is demonstrated through experimental evaluations that the TCE technique is successful in effectively detecting equivalent mutants of a C program using the *gcc* compiler optimizations. However, since the Java compilers generally do not perform noticeable optimizations, the performance of TCE on Java programs needs to be investigated. As a result, current paper proposes TCE+ technique as an extension of TCE that utilizes ProGuard for the purpose of optimizing Java code. In addition to performing different optimizations, e.g. dead code removal, unused variable removal and peephole optimizations, ProGuard is also able to obfuscate, shrink and pre-verify Java byte codes. However, TCE+ uses ProGuard only for the purpose of optimizations and it does not use obfuscation or shrinking capabilities of ProGuard. It is beyond the scope of this paper to describe the optimization techniques employed by ProGuard or *gcc*, however, Table 1 briefly mentions some of the main optimizations performed by each of these tools.

In [12], TCE has been shown to be able to find, in addition to equivalent mutants, the duplicated mutants, i.e. mutants that are equivalent to each other, but not necessarily equivalent to the original program. Since there is no advantage in using two duplicated mutants, it is interesting to be able to detect duplicated mutants. In this paper, we evaluate the TCE and TCE+ methods for the purpose of detecting equivalent and duplicated mutants of Java programs.

Table 1. Some of the optmization techniques employed by the subject tools

Tool	Optimization techniques
gcc Compiler	Dead Code Elimination, Transforming Conditional Jumps, Constant Folding, De-Virtualization, Function Inlining, Predictive Commoning, Elimination of Useless Null Pointer Checks, Peephole Optimization, Global Common Subexpression Elimination
ProGuard	Dead Code Elimination, Peephole Optimization, Marking Classes as Final, Variable Allocation Optimization, Method Inlining, Return Value Propagation, Removing Write-only Fields

3 Experimental Study

In this section, the experimental evaluation of the TCE and TCE+ approaches over Java programs is discussed. First, the research questions are introduced and then, different elements of the experiments are described. Finally, the results of the experiments are discussed.

3.1 Research Questions

Since the TCE approach has been shown to be both effective and efficient in detecting equivalent and duplicated mutants in C programs, the main research question this paper seeks to answer is:

RQ. How do the TCE and TCE+ approaches perform on Java programs?
To answer this question, two more specific research questions are introduced.

RQ1. How effective are the TCE and TCE+ approaches at detecting equivalent and duplicated mutants in Java programs?
To answer this question, the number of equivalent and duplicated mutants detected by the TCE and TCE+ techniques, and also the ratio of the detected equivalent mutants to the existing equivalent mutants is reported.

RQ2. How efficient is TCE+ for the purpose of equivalent mutant detection?
This question is answered by computing the execution time of the TCE+ approach to see if it is efficient enough to be used in practice. While we have not evaluated TCE+ on large programs, we believe that the efficiency of the technique for the large programs can be estimated based on the results obtained for the small programs.

3.2 Dataset and Golden Standard

For the purpose of the experimental evaluations, first, a dataset is prepared including 5 java programs, and then, for each program, its mutants are created by the MuJava mutation testing tool [23]. Table 2 shows the name of each program, its size in terms of physical Source Line of Code (SLOC) and the number of its mutants. The mutation operators that MuJava has applied on the subject programs are mentioned in Table 3.

In addition, a golden standard is created by manually checking each mutant of the subject programs to determine whether it is equivalent to the original program. This manual analysis is performed separately by three experts who have had more than 10 years of experience in object oriented programming in Java. After each expert has

Table 2. Dataset used in the experiments

Program	Subject program	Physical SLOC	Number of mutants
P1	BubbleSort	15	111
P2	Bisect	25	189
P3	Triangle	46	456
P4	QuickSort	50	341
P5	java.util.StringTokenizer	174	772

Table 3. Mutation operators applied by MuJava on the subject programs

Operator	Operator definition
AODS: Short-cut Arithmetic Operator Deletion	{(x,remove(x)) | x ∈ {++, −−}}
AODU: Unary Arithmetic Operator Deletion	{(−v, v)}
AOIS: Short-cut Arithmetic Operator Insertion	{(v, −−v), (v, v–), (v, ++v), (v, v++)}
AOIU: Unary Arithmetic Operator Insertion	{(v, −v)}
AORB: Binary Arithmetic Operator Replacement	{(x,y) | x,y ∈ {+, −, *, /, %} ∧ x ≠ y}
AORS: Shortcut Arithmetic Operator Replacement	{(x,y) | x,y ∈ {++, −−} ∧ x ≠ y}
ASRS: Shortcut Assignment Operator Replacement	{(x,y) | x,y ∈ {+=, −=, *=, /=, %=} ∧ x ≠ y}
CDL: Constant DeLetion	{(op c, remove(op c)) | op ∈ {+, −, *, /, %, >, >=, <, <=}}
COD: Conditional Operator Deletion	{(!(e), e) | e ∈ {if(e), while(e), for(s; e; s)}}
COI: Conditional Operator Insertion	{(e, !(e)) | e ∈ {if(e), while(e), for(s; e; s)}}
COR: Conditional Operator Replacement	{(x,y) | x,y ∈ {&&, ‖, ^} ∧ x ≠ y}
LOI: Logical Operator Insertion	{(v, ∼ v)}
ODL: Operator DeLetion	{(v op, remove(v op)), (op v, remove(op v)) | op ∈ {+, −, *, /, %, <, <=, >, >=}}, {(v++, v), (v−−, v), (−−v, v), (++v, v) | op ∈ {++, −}}
ROR: Relational Operator Replacement	{(x,y) | x,y ∈ {>, >=, <, <=, ==, !=} ∧ x ≠ y}
SDL: Statement DeLetion	{(s, remove(s))}
VDL: Variable DeLetion	{(v [op], remove(v [op])) | op ∈ {+, −−, *, /, %, ++, –, <, <=, >, >=}

finished his job, the results have been compared so that any possible conflict is resolved. Actually, there were 7 such cases that needed the experts to discuss with each other to agree on the result.

3.3 Experimental Environment

All the experiments are performed on a PC with Microsoft Windows 7 operating system, Intel Core i5-4400 processor and 8 GB RAM. Further, we have used the Oracle's Java compiler javac version 1.8.0_60 to compile the programs and the mutants, and also ProGuard 5.3 to optimize the compilation results. Finally, for the purpose of comparing the binary files, the Windows utility program FC is used with the parameters /B and /LB1.

3.4 Experiments

To answer the research questions, four experiments are designed. The first two experiments evaluate the TCE and TCE+ techniques for the purpose of equivalent mutant

detection and the second two experiments evaluate them for detecting duplicated mutants. The processes used in these experiments are shown in Figs. 1, 2, 3 and 4.

```
Input: P (original program)
Output: EM (list of the equivalent mutants of P)

//compile step
compile P to P_class
for each mutant M of P
  compile M to M_class
//comparison step
for each mutant M of P
  result = compare M_class to P_class
  if (result == 'no difference')
    add M to EM
return EM
```

Fig. 1. Process of experiment 1: TCE for equivalent mutant detection

```
Input: P (original program)
Output: EM (list of the equivalent mutants of P)

//compile step
compile P to P_class
for each mutant M of P
  compile M to M_class
//optimization step
convert P_class to P_jar
optimize P_jar to P_jar,op
extract P_class,op from P_jar,op
P_class = P_class,op
for each mutant M of P
  convert M_class to M_jar
  optimize M_jar to M_jar,op
  extract M_class,op from M_jar,op
  M_class = M_class,op
//comparison step
for each mutant M of P
  result = compare M_class to P_class
  if (result == 'no difference')
    add M to EM
return EM
```

Fig. 2. Process of experiment 2: TCE+ for equivalent mutant detection

```
Input: P (original program)
Output: DM (list of the removable duplicated mutants of
P)

//compile step
for each mutant M of P
  compile M to M_class
//comparison step
Pairs: empty list
for each mutant M1 of P
  for each mutant M2 of P
    if (M1 != M2 and filesize(M1_class)==filesize(M2_class))
        result = compare M1_class to M2_class
        if (result == 'no difference')
          add pair(M1, M2) to Pairs
//removal step
sort Pairs based on the first element of the pairs
for each Pair in Pairs
    M1 = first element of Pair
  M2 = second element of Pair
  if not (DM contains M1)
      add M2 to DM
return DM
```

Fig. 3. Process of experiment 3: TCE for duplicated mutant detection

In the first experiment, for each subject program P, P is compiled to P_{class} and each mutant M of P is compiled to M_{class}. Then each compiled mutant M_{class} is compared to the P_{class}. If no difference is identified in this comparison, it is considered that TCE has determined the corresponding mutant as an equivalent mutant.

The second experiment evaluates the TCE+ approach by including an optimization phase before the comparison step. In order to perform the optimization, first a jar file is created from the compiled file, i.e. P_{class} or M_{class}. The jar file is then given to ProGuard to do the optimizations. The resulting jar file is then decompressed to extract the optimized compiled file which then goes through the binary comparison.

In the third experiment, each compiled mutant of the program is compared to all other compiled mutants of that program that have the same file size. If there is no difference between the corresponding binary files, those two mutants are added as a pair to the list of duplicated mutants. After processing all the mutants, a simple algorithm shown in Fig. 3 is used to determine the list of mutants that can be removed.

The fourth experiment is very similar to the third experiment and the only difference is that it compares the optimized version of the compiled mutants which are created by the process described for the second experiment.

```
Input: P (original program)
Output: DM (list of the removable duplicated mutants of
P)

//compile step
for each mutant M of P
  compile M to M_class
//optimization step
for each mutant M of P
  convert M_class to M_jar
  optimize M_jar to M_jar,op
  extract M_class,op from M_jar,op
  M_class = M_class,op
//comparison step
Sort mutations based on their file size
for each mutant M1 of P
  if (M1 in DM)
    continue;
  for each mutant M2 of P
    if (M2 in DM)
      continue;
    if (M1 != M2
      and filesize(M1_class) == filesize(M2_class))
       result = compare M1_class to M2_class
       if (result == 'no difference')
         add M2 to DM
    else
        break;
return DM
```

Fig. 4. Process of experiment 4: TCE+ for duplicated mutant detection

3.5 Result Analysis

The results of the first two experiments are shown in Table 4. As it is shown in this table, TCE approach has not detected any equivalent mutant in the subject programs. Therefore, it can be concluded that since the Java compiler does not perform noticeable optimizations [26], applying TCE on Java programs is not effective for detecting equivalent mutants. However, the TCE+ technique, which compensates the limitation of the Java compiler by utilizing ProGuard's optimizations, has identified some equivalent mutants for each of the subject programs. Therefore, TCE+ has been able to address the shortcomings of the TCE method. However, the number of detected equivalent mutants is small and at the best case, i.e. the Bisect program, it accounts for

only 7% of all the mutants. The worst case is also the BubbleSort program that the detected equivalent mutants are only 2% of all the mutants.

In order to judge the effectiveness of the TCE+ approach, it is required to know the ratio of the detected equivalent mutants to all the existing equivalent mutants. Therefore, the results of the first two experiments have been compared with the golden standard. As shown in the last column of Table 4, TCE+ has been able to detect from 18% to 100% of all the existing equivalent mutants. It has missed 9, 2 and 7 equivalent mutants respectively for the BubbleSort, QuickSort and StringTokenizer programs. For the other two programs, i.e. Bisect and Triangle, all the existing equivalent mutants have been found by TCE+ .

Based on these results, we conclude that TCE+ is generally effective and it is successful in detecting a good ratio of the existing equivalent mutants. However, it is interesting to analyze the detected and undetected equivalent mutants based on their mutation operators.

The distribution of the mutation operators over all the generated mutants is shown in Table 5. The top-3 mutation operators that have created the greatest proportion of the mutants are AOIS, ROR and SDL, which have created respectively 33%, 20% and 10% of all the mutants. There are some operators like AOSE and AODU that have negligible contribution to the number of mutants created.

Table 4. Results of experiments 1 and 2: Detecting equivalent mutants

Program	Number of detected equivalent mutants		Percentage of detected equivalent mutants to all mutants		Percentage of detected equivalent mutants to all existing equivalent mutants	
	TCE	TCE+	TCE	TCE+	TCE	TCE+
P1	0	2	0	2	0	18
P2	0	14	0	7	0	100
P3	0	23	0	5	0	100
P4	0	10	0	3	0	83
P5	0	34	0	4	0	83

In Table 6, the distribution of the mutation operators over all the existing equivalent mutants is shown. An interesting point is that the AOIS operator which has created about 33% of all the mutants is also responsible for creating about 77% of all the equivalent mutants in the golden standard. Further, the ROR operator has created about 14% of all the equivalent mutants. From another point of view, about 13% of the mutants created by the AOIS operator have been equivalent. This value for the ROR operator has been about 4%. This means that the performance of the TCE+ technique over these two mutation operators is of greater importance, compared to other mutation operators.

Table 5. Distribution of the mutation operators over all the mutants

Program	Mutation operator															
	AODS	AODU	AOIS	AOIU	AORB	AORS	ASRS	CDL	COD	COI	COR	LOI	ODL	ROR	SDL	VDL
P1			30	3	16	2		4		3		11	8	19	10	5
P2			80	13	32			2		3			16	19	14	10
P3			128	11	36			3		24	14	43	32	119	31	15
P4	2		108	18	36	6		8		9		40	20	55	28	11
P5		2	262	33		7	20		6	39	20	80	33	163	100	7
Total	2	2	608	78	120	15	20	17	6	78	34	174	109	375	183	48
Ratio (%)[a]	<1	<1	33	4	6	1	1	1	<1	4	2	9	6	20	10	3

[a] Percentage to all the mutants

The distribution of the mutation operators over all the equivalent mutants that are found by TCE+ is shown in Table 7. Comparing this table with Table 6 shows that TCE+ has successfully detected all the equivalent mutants created by the AOIS operator, which account for about 77% of all the equivalent mutants. Hence, considering the ratio of AOIS-generated equivalent mutants, it can be concluded that the TCE + approach is an effective method for detection of equivalent mutants in Java programs. However, it is also important to note that TCE+ has not detected any of the 14 equivalent mutants created by the ROR operator (5 for BubbleSort, 2 for QuickSort and 7 for StringTokenizer). It also has missed 4 other equivalent mutants of BubbleSort, 2 created by the AORB operator, 1 by ODL and 1 by the CDL operator.

Regarding detection of the duplicated mutants, the results of the third and the fourth experiments are presented in Table 8. This table shows that TCE and TCE+ have identified respectively from 8% to 14% and from 13% to 23% of the mutants of the subject programs as being duplicated. Since the duplicated mutants do not contribute to the mutation testing results, they can be removed from the mutants. Considering all the five subject programs, TCE and TCE+ have identified respectively 9% and 16% of all the mutants as being duplicated. As a result, we conclude that while TCE+ noticeably outperforms TCE, both approaches are effective in detecting duplicated mutants.

An interesting point is that while TCE has not detected any equivalent mutant, but it has detected non-negligible number of duplicated mutants. Further analysis of the

Table 6. Distribution of the mutation operators over the existing equivalent mutants

Program	Mutation operator															
	AODS	AODU	AOIS	AOIU	AORB	AORS	ASRS	CDL	COD	COI	COR	LOI	ODL	ROR	SDL	VDL
P1			2		2			1					1	5		
P2			12	2												
P3			20	1									1			1
P4			10											2		
P5			34											7		
Total	0	0	78	3	2	0	0	1	0	0	0	0	2	14	0	1
Ratio (%)[a]	0	0	77	3	2	0	0	1	0	0	0	0	2	14	0	1

[a] Percentage to Existing Equivalent Mutants

Table 7. Distribution of the operators over the equivalent mutants detected by TCE+

Program	Mutation operator															
	AODS	AODU	AOIS	AOIU	AORB	AORS	ASRS	CDL	COD	COI	COR	LOI	ODL	ROR	SDL	VDL
P1			2													
P2			12	2												
P3			20	1									1			1
P4			10													
P5			34													
Total	0	0	78	3	0	0	0	0	0	0	0	0	1	0	0	1
Ratio (%)[a]	0	0	94	4	0	0	0	0	0	0	0	0	1	0	0	1

[a] Percentage to all equivalent mutants detected by TCE+

Table 8. Results of experiments 3 and 4: Detecting duplicated mutants

Program	Number of detected duplicated mutants		Percentage of detected duplicated mutants to all mutants	
	TCE	TCE+	TCE	TCE+
P1	15	25	14	23
P2	16	31	8	16
P3	52	89	11	20
P4	34	59	10	17
P5	60	99	8	13

results reveals that the detected duplicated mutants are not a result of the optimizations made by TCE, but they are resulted from the fact that applying some MuJava mutation operators on some program statements may create exactly the same syntactic changes. In other words, for each pair of duplicated mutants detected by TCE, both mutants are syntactically-equal. An example pair is shown in Table 9. While TCE+ has detected all the duplicated mutants found by TCE, it has also detected other results which are syntactically different but semantically duplicated. An example is shown in Table 10.

Another interesting point is that, as shown in Table 11, 44% of all the duplicated mutants detected by TCE are created by the ROR operator. The other 23% are associated with the VDL operator. Only about 1% of the detected duplicated mutants are results of the AOIS operator. The results for the TCE+ technique are also presented in Table 12. This table shows that, compared to TCE, the TCE+ technique is able to detect the duplicated mutants that are created by a wider set of mutation operators. Actually, TCE+ has detected duplicated mutants of type AOI, AORB, CDL and LOI operators, of which none is detected by the TCE method.

Finally, to answer RQ1, we conclude that TCE is not effective for detecting equivalent mutants of Java programs, but it can effectively detect the duplicated mutants. Further, TCE+ is effective for detecting both equivalent and duplicated mutants.

Table 9. An example duplicated mutant detected by TCE

Original statement	Mutant by ODL operator	Mutant by CDL operator
x = (M + x)/2;	x = M + x;	x = M + x;

Table 10. An example duplicated mutant detected by TCE+ but missed by TCE

Original Statement	Mutant by AOIS Operator	Mutant by AOIS Operator
public void setEpsilon (double epsilon) {this. mEpsilon = epsilon;}	public void setEpsilon (double epsilon) {this. mEpsilon = epsilon−−;}	public void setEpsilon (double epsilon) {this. mEpsilon = epsilon++;}

Table 11. Distribution of the operators over the duplicated mutants detected by TCE

Program	Mutation Operator															
	AODS	AODU	AOIS	AOIU	AORB	AORS	ASRS	CDL	COD	COI	COR	LOI	ODL	ROR	SDL	VDL
P1													4	3	3	5
P2													4		2	10
P3													3	27	7	15
P4			2										10	9	5	8
P5													8	39	11	2
Total	0	0	2	0	0	0	0	0	0	0	0	0	29	78	28	40
Ratio (%)[a]	0	0	1	0	0	0	0	0	0	0	0	0	16	44	16	23

[a] Percentage to all duplicated mutants detected by TCE

Table 12. Distribution of the operators over the duplicated mutants detected by TCE+

Program	Mutation Operator															
	AODS	AODU	AOIS	AOIU	AORB	AORS	ASRS	CDL	COD	COI	COR	LOI	ODL	ROR	SDL	VDL
P1			1		4			4					4	4	3	5
P2			13	2									4		2	10
P3			19	1	3			1				1	4	38	7	15
P4			14		6			6					10	10	5	8
P5			34										8	43	12	2
Total	0	0	81	3	13	0	0	11	0	0	0	1	30	95	29	40
Ratio (%)[a]	0	0	27	1	4	0	0	4	0	0	0	0	10	31	10	13

[a] Percentage to all duplicated mutants detected by TCE+

In order to evaluate efficiency of TCE+ for detecting equivalent mutants, its execution time for different steps, i.e. (1) compiling the mutants, (2) optimization of the compiled mutants, and (3) comparison of the optimization results, is separately measured for each subject program. The process of detecting duplicated mutants also includes the first two steps, but in the third step, it compares the optimization results differently. Therefore, the execution time of this step is also measured to evaluate efficiency of TCE+ for detecting duplicated mutants. The results are presented in Table 13.

Table 13. Execution time of TCE+ for detecting equivalent and duplicated mutants

Program	Execution time (s)					
	Compile	Optimization	Comparison for detecting equivalent mutants	Comparison for detecting duplicated mutants	Total for detecting equivalent mutants	Total for detecting duplicated mutants
P1	36	68	1	1	105	105
P2	57	124	3	1	184	182
P3	137	289	6	3	432	429
P4	101	188	5	2	294	291
P5	235	617	12	5	864	857

As shown in Table 13, the execution times of detecting equivalent mutants and duplicated mutants do not differ noticeably, and they are about 1 s per mutant. Therefore, to answer RQ2, we conclude that TCE+ can be considered as an efficient method. Further, the comparison times, both for equivalent and duplicated mutants, are negligible. However, the optimization time is about 2–3 times the compile time. It is worth noting that the compile time is an inherent overhead of mutation testing, since in mutation testing, each mutant should be compiled and executed against the test cases. Therefore, the overhead imposed by TCE+ is the optimization time. Considering the fact that TCE+ can effectively detect equivalent and duplicate mutants, and these mutants do not need to be executed over the test cases, it means that TCE+ reduces the cost of mutation testing by reducing the number of mutants that need to be run and specially by removing the mutants that due to their equivalence, can waste the time of the test case designers. Hence, we believe the overhead of optimization time which involves CPU cycles can be considered as acceptable by the reduction it provides in required human effort. Consequently, we conclude that TCE+ is cost effective.

4 Conclusion

In this paper, the performance of TCE technique for detecting equivalent mutants in Java programs is evaluated. As the experimental evaluations have demonstrated, TCE has not detected any equivalent mutant in the subject programs and hence it cannot be considered to effective. To address this problem, current paper has proposed the TCE+ technique which extends TCE by utilizing an obfuscator like ProGuard, capable of performing some optimizations on Java programs.

The experimental evaluations show that while there are mutation operators like ROR for which TCE+ performance is weak, there are also operators like AOIS that TCE+ is able to find all of its equivalent mutants. Considering the contribution of each operator to the number of equivalent mutants of a typical program, TCE+ can be considered to be an effective and efficient method for detecting both equivalent and duplicated mutants for Java programs.

Current paper has investigated performance of TCE+ on small programs. Hence, it is required to perform similar experiments on larger Java programs to see how the performance of TCE+ changes as the program size increases. A challenge in this regard

is preparation of the golden standard, since for large programs, the number of mutants is noticeable and it needs considerable effort to build a reliable golden standard. This is a main direction of our future work. Further, more precise analysis of the behavior of TCE+ on different mutation operators is an important job that we have scheduled for our future works. The results of such analysis will provide insights on possible improvements on ProGuard from the specific point of view of equivalent mutant detection.

References

1. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. **37**(5), 649–678 (2011)
2. Budd, T.A., Angluin, D.: Two notions of correctness and their relation to testing. Acta Informatica **18**(1), 31–45 (1982)
3. Offutt, A.J., Pan, J.: Automatically detecting equivalent mutants and infeasible paths. Softw. Testing Verification Reliab. **7**(3), 165–192 (1997)
4. Madeyski, L., et al.: Overcoming the equivalent mutant problem: a systematic literature review and a comparative experiment of second order mutation. IEEE Trans. Softw. Eng. **40** (1), 23–42 (2014)
5. Schuler, D., Zeller, A.: Covering and uncovering equivalent mutants. Softw. Testing Verification Reliab. **23**(5), 353–374 (2013)
6. Papadakis, M., Le Traon, Y.: Mutation testing strategies using mutant classification. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing. ACM (2013)
7. Adamopoulos, K., Harman, M., Hierons, R.M.: How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. In: Deb, K. (ed.) GECCO 2004. LNCS, vol. 3103, pp. 1338–1349. Springer, Heidelberg (2004). doi:10.1007/978-3-540-24855-2_155
8. Kintis, M., Malevris, N.: Using data flow patterns for equivalent mutant detection. In: 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops (ICSTW). IEEE (2014)
9. Jia, Y., Harman, M.: Higher order mutation testing. Inf. Softw. Technol. **51**(10), 1379–1393 (2009)
10. Hierons, R., Harman, M., Danicic, S.: Using program slicing to assist in the detection of equivalent mutants. Softw. Testing Verification Reliab. **9**(4), 233–262 (1999)
11. Schuler, D., Andreas Z.: (Un-) Covering equivalent mutants. In: 2010 Third International Conference on Software Testing, Verification and Validation. IEEE (2010)
12. Papadakis, M., et al.: Trivial compiler equivalence: a large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 1. IEEE (2015)
13. Kintis, M., Malevris, N.: Identifying more equivalent mutants via code similarity. In: 2013 20th Asia-Pacific Software Engineering Conference, vol. 1. IEEE (2013)
14. Vincenzi, A.M.R., et al.: Bayesian-learning based guidelines to determine equivalent mutants. Int. J. Softw. Eng. Knowl. Eng. **12**(06), 675–689 (2002)
15. Nica, S., Wotawa, F.: Using constraints for equivalent mutant detection. arXiv preprint arXiv:1207.2234 (2012)
16. Just, R., Ernst, M.D., Fraser, G.: Using state infection conditions to detect equivalent mutants and speed up mutation analysis. arXiv preprint arXiv:1303.2784 (2013)

17. Rojas, J.M., Fraser, G.: Code defenders: a mutation testing game. In: The 11th International Workshop on Mutation Analysis. IEEE (2015)
18. Kintis, M., Malevris, N.: MEDIC: A static analysis framework for equivalent mutant identification. Inf. Softw. Technol. **68**, 1–17 (2015)
19. Yao, X., Harman, M., Jia, Y.: A study of equivalent and stubborn mutation operators using human analysis of equivalence. In: Proceedings of the 36th International Conference on Software Engineering. ACM (2014)
20. Harman, M., Jia, Y., Langdon, W.B.: A manifesto for higher order mutation testing. In: 2010 Third International Conference on Software Testing, Verification, and Validation Workshops (ICSTW). IEEE (2010)
21. Nguyen, Q.V., Madeyski, L.: Searching for strongly subsuming higher order mutants by applying multi-objective optimization algorithm. In: Le Thi, H.A., Nguyen, N.T., Do, T.V. (eds.) Advanced Computational Methods for Knowledge Engineering. AISC, vol. 358, pp. 391–402. Springer, Cham (2015). doi:10.1007/978-3-319-17996-4_35
22. Omar, E., Ghosh, S., Whitley, D.: Constructing subtle higher order mutants for Java and AspectJ programs. In: 2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE). IEEE (2013)
23. Ma, Y.-S., Offutt, J., Kwon, Y.-R.: MuJava: a mutation system for Java. In: Proceedings of the 28th International Conference on Software Engineering. ACM (2006)
24. Just, R., et al.: 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. ACM (2014)
25. Weitao, W., Hirohide, H.: Improvement of equivalent mutant detection using loop count restriction. In: The International Conference on Software Engineering, Mobile Computing and Media Informatics (SEMCMI 2015) (2015)
26. Diehl, S.: A formal introduction to the compilation of Java. Softw.-Pract. Experience **28**(3), 297–327 (1998)