The Journal of Systems and Software 115 (2016) 18–30

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Mutation testing cost reduction by clustering overlapped mutants

Yu-Seung Ma[a,*], Sang-Woon Kim[b]

[a] Electronics and Telecommunications Research Institute, Daejeon, South Korea
[b] FormalWorks, Inc., Seoul, South Korea

ARTICLE INFO

Article history:
Received 20 October 2014
Revised 24 December 2015
Accepted 6 January 2016
Available online 13 January 2016

Keywords:
Software testing
Mutation testing

ABSTRACT

Mutation testing is a powerful but computationally expensive testing technique. Several approaches have been developed to reduce the cost of mutation testing by decreasing the number of mutants to be executed; however, most of these approaches are not as effective as mutation testing which uses a full set of mutants. This paper presents a new approach for executing fewer mutants while retaining nearly the same degree of effectiveness as is produced by mutation testing using a full set of mutants. Our approach dynamically clusters expression-level weakly killed mutants that are expected to produce the same result under a test case; only one mutant from each cluster is fully executed under the test case. We implemented this approach and demonstrated that our approach efficiently reduced the cost of mutation testing without loss of effectiveness.

1. Introduction

Mutation testing (DeMillo et al., 1978) is a testing technique that measures the quality of test cases and helps in designing new test cases. Mutation testing involves modifying a program by introducing simple syntactic changes and creating possible faulty versions, called *mutants*. Test cases are executed against both the original program and mutants. A mutant is *killed* by a test case that causes the mutant program to produce a different output from the original program's output. Test cases are considered to be effective if they kill most mutants.

Although mutation testing is powerful (Mathur and Wong, 1994; Frankl et al., 1997), its high execution cost has always been a problem to be solved. Several approaches have been proposed to reduce the cost of mutation testing. Cost reduction approaches are well reviewed in the survey papers (Jia and Harman, 2011; Usaola and Mateo, 2010). Among the approaches, reducing the number of mutants to be executed is the most obvious way to reduce costs. However, most approaches that run fewer mutants have the drawback that they are not as effective as mutation testing that uses a full set of mutants.

To address this issue, we propose a new approach that executes fewer mutants but is nearly as effective as approaches that use a full set of mutants. Our approach dynamically clusters mutants that are expected to produce the same results against a test case. For that, the execution of mutants whose mutated code is in a common position is halted immediately after executing the mutated code. The intermediate results are then compared, and mutants with identical intermediate results are clustered. A single mutant from each cluster is then fully executed using a strong mutation method against the test case. If the fully executed mutant is killed (or live), all remaining mutants in the cluster are considered to be killed (or live). The advantage of our approach is that it reduces the cost of mutation testing by restricting the number of mutants that are fully executed. In addition, it can lead to a further cost reduction by easily combining existing cost reduction approaches.

Our approach was implemented by extending a Java mutation system and experiments were conducted to determine the efficiency.

This paper is organized as follows. Sections 2 describes related work and Section 3 provides some background information in support of the new approach. Section 4, the main part of the paper, describes the cost reduction method for mutation testing. Section 5 presents experimental results and a cost comparison. Section 6 presents conclusions and discusses future work.

2. Related work

The number of mutants significantly affects the execution cost of mutation testing because each mutant is executed repeatedly against at least one (potentially many) test case. Many researches have attempted to reduce the number of mutants without significant loss of test effectiveness.

* Corresponding author. Tel.: +82 42 860 6551.
E-mail address: ysma@etri.re.kr (Y.-S. Ma).

http://dx.doi.org/10.1016/j.jss.2016.01.007
0164-1212/

A mutation survey paper (Jia and Harman, 2011) classified the approaches that use fewer mutants into four categories: (1) mutant sampling, (2) selective mutation, (3) higher order mutation, and (4) mutant clustering.

Mutant sampling (Acree, 1980; Budd, 1980) uses a small percentage, say *x%*, of randomly selected mutants and ignores the remaining mutants. An empirical study conducted by Wong (1993) showed that test sets adequate for 10% of randomly chosen mutants were only 16% less effective than mutation analysis that used a full set of mutants.

Selective mutation (Wong et al., 1994; Offutt et al., 1996) uses *selective mutation operators*, comprising portions of the entire mutation operators that are nearly as effective as non-selective mutation. Several studies (Wong et al., 1994; Offutt et al., 1996) have been conducted to identify efficient selective mutation operators. One of the widely used selective mutation operator set consists of five mutation operators (Offutt et al., 1996): ABS, UOI, LCR, AOR and ROR. Experimental trials showed that these five operators provide nearly the same coverage as non-selective mutation operators, with cost reduction of at least four times with small programs.

Higher order mutation (Jia and Harman, 2009; Polo et al., 2008; Mateo et al., 2013) uses higher-order mutants instead of first order mutants. The approach was originally proposed to identify higher-order mutants that denote subtle faults, and it also suggested subsuming higher order mutant which may be preferable to replace first order mutant.

Mutant clustering (Hussain, 2008; Ji et al., 2009) is an approach that selects a subset of mutants using a clustering algorithm. Each mutant in a cluster is likely to be killed by the same set of test cases; thus, one or a part of mutants from each cluster is used for mutation testing. The possibility of mutant clustering was first shown in the master's thesis of Hussain (2008), which clusters mutants by analyzing similarities among mutants. However, the method determines the similarity by executing all mutants repeatedly against all test cases. Ji et al. (2009) uses a domain analysis to determine similarity among mutants. The method uses symbolic execution to analyze the domain of variables; thus, its effectiveness is subordinate to the symbolic execution's ability.

This section introduces two additional categories for reducing the number of mutants: 'mutation subsumption' and 'dynamic mutant filtering' approaches.

Mutation subsumption (Kaminski et al., 2013; Just et al., 2012; Ammann et al., 2014; Kurtz et al., 2014) approach aims at not creating redundant mutants that are subsumed by other mutants. Kaminski et al. (2013) proposed that tests detecting three of the ROR (Relational Operator Replacement) mutants subsume (are guaranteed to detect) all seven ROR mutants; thus, the three ROR mutants were non-redundant. Just et al. (2012) investigated the subsumption relations among COR (Conditional Operator Replacement) mutants and suggested that only three COR mutants were non-redundant. Ammann et al. (2014) proposed a dynamic subsumption approach. They proposed a model for minimizing mutants with respect to a test set, thus, a minimal set of non-redundant mutants can be changed according to the used test set. Kurtz et al. (2014) define true subsumption, dynamic subsumption, and static subsumption to model the redundancy between mutants and develop a graph model to display the subsumption relationship.

Dynamic mutant filtering (Schuler and Zeller, 2009; Weiss and Fleyshgakker, 1993; Kim et al., 2013) is an approach which dynamically filters out mutants to be expected to be strongly live for each test case and improves the speed by excluding their execution. Schuler and Zeller (2009) executed only the reachable mutants with a code coverage analysis. Weiss and Fleyshgakker (1993) proposed an approach in which weak and strong mutations were combined for an interpretive mutation system. It filtered out weakly killed mutants. Kim et al.'s study (Kim et al., 2013) extended their approach to a non-interpretive mutation system. Experimental results showed that removing the execution of unreachable and weakly live mutants significantly reduced the cost of mutation testing.

The approach proposed in this paper takes advantage of both the dynamic mutant filtering approach and the mutant clustering approach. For each test case, weakly killed mutants are filtered, and weakly killed mutants with identical intermediate results are clustered. A single mutant from each cluster is then fully executed with the test case. The goal is to reduce the number of mutants that must be fully executed without reducing the test effectiveness.

3. Definition of conditionally overlapped mutants

We define the mutant M1 is *overlapped* to the mutant M2, and vice versa, if the mutants M1 and M2 are functionally identical. That is, the mutant M1 is an *overlapped mutant* of the mutant M2, and vice versa. Although the definition of an overlapped mutant is similar to that of an equivalent mutant, the concept is slightly different. An equivalent mutant (Yao et al., 2014) is a mutant that is functionally identical to the original program, thus it cannot be killed by any test case. On the other hand, an overlapped mutant is a mutant that is functionally identical to at least one other mutant and can be killed by some test cases if it is not an equivalent mutant. If a mutant is killed (or live), all of its overlapped mutants are killed (or live) in the same manner. Therefore, without executing all mutants, we can predict their results by running only one mutant from each set of overlapped mutants.

Consider the statement '$C = A + B$;' and its mutated versions of mutants $m1$ and $m2$: '$C = A - B$;' and '$C = A + (-B)$;'. In this example, the mutants $m1$ and $m2$ are overlapped. Executing both mutants would be a duplicated effort. The identification of overlapped mutants prior to execution would be beneficial; however, the complete detection of overlapped mutants is impossible because it is essentially the same problem as detecting equivalent mutants. Instead, we focus on a specific type of overlapped mutants, the *conditionally overlapped mutants*, described below.

Conditionally overlapped mutants are defined using a looser definition of the overlapped mutants. The mutant M1 is *conditionally overlapped (c-overlapped)* to the mutant M2 for a test case if the mutants M1 and M2 produce identical results for the test case. To avoid confusion, we will refer to overlapped mutants as *absolutely overlapped (a-overlapped) mutants*. A-overlapped mutants always produce identical results for any test cases; however, c-overlapped mutants are clustered depending on the test case, thus, different sets of c-overlapped mutants will be formed if a different test case is used.

Let us examine the program code in Fig. 1. Applying the ROR and AOR operators to the program yields 15 different mutated codes, as listed in Table 1. The ROR mutation operator is applied to the third line of code and produces seven mutants. The AOR mutation operator is applied to the fourth and sixth lines of code and produces eight mutants.

```
1  int myFunction(int A, int B) {
2     int C;
3     if ( A != B )
4        C = A + B;
5     else
6        C = A * B;
7     return C;
8  }
```

Fig. 1. An example code for conditionally overlapped mutants.

Table 1
Mutants and evaluated values for test cases $t1$ and $t2$.

Mutant	Code line	Mutated code	Test result		
			$t1$ (A = 4,B = 2)	$t2$ (A = 4,B = 4)	$t3$ (A = 5,B = 4)
ROR_1	3	A > B	6	16	9
ROR_2	3	A ≥ B	6	8	9
ROR_3	3	A < B	8	16	20
ROR_4	3	A ≤ B	8	8	20
ROR_5	3	A == B	8	8	20
ROR_6	3	True	6	8	9
ROR_7	3	False	8	16	20
AOR_1	4	A − B	2	16	1
AOR_2	4	A / B	2	16	1
AOR_3	4	A ∗ B	8	16	20
AOR_4	4	A % B	0	16	1
AOR_5	6	A + B	6	8	9
AOR_6	6	A − B	6	0	9
AOR_7	6	A / B	6	1	9
AOR_8	6	A % B	6	0	9

Table 1 also shows the results of the *'myFunction'* method under three test cases $t1$, $t2$, and $t3$. Under the $t1$ test case, four different results (0, 2, 6, and 8) are produced. Result 0 is produced by only one mutant, AOR_4. However, result 2 is produced by two mutants, AOR_1 and AOR_2, which indicates that the AOR_1 and AOR_2 mutants are c-overlapped. For result 6, as many as seven mutants (i.e., ROR_1, ROR_2, ROR_6, AOR_5, AOR_6, AOR_7, and AOR_8) are c-overlapped. For result 8, five mutants, ROR_3, ROR_4, ROR_5, ROR_7, AOR_3, are c-overlapped.

Please note that the ROR_1 and ROR_2 mutants produces the same result under the $t1$ test case but different result under the $t2$ test case. That is, the ROR_1 and ROR_2 mutants are c-overlapped for the $t1$ test case but not for the $t2$ test case.

Let $CoM(t)$ be sets of c-overlapped mutants for the test case t. Then, for the test cases $t1$, $t2$, and $t3$ in Table 1, c-overlapped mutant sets are generated as follows:

$$CoM(t1) = \{\{ROR_1, ROR_2, ROR_6, AOR_5, AOR_6, AOR_7, AOR_8\}, \{ROR_3, ROR_4, ROR_5, ROR_7, AOR_3\}, \{AOR_1, AOR_2\}, \{AOR_4\}\}$$

$$CoM(t2) = \{\{ROR_1, ROR_3, ROR_7, AOR_1, AOR_2, AOR_3, AOR_4\}, \{ROR_2, ROR_4, ROR_5, ROR_6, AOR_5\}, \{AOR_6, AOR_8\}, \{AOR_7\}\}$$

$$CoM(t3) = \{\{ROR_1, ROR_2, ROR_6, AOR_5, AOR_6, AOR_7, AOR_8\}, \{ROR_3, ROR_4, ROR_5, ROR_7, AOR_3\}, \{AOR_1, AOR_2, AOR_4\}\}$$

If we know c-overlapped mutants for a test case prior to their execution, applying only a mutant from a set of c-overlapped mutants is sufficient to determine whether any of them are killed. The cost of mutation testing is then reduced by executing only one mutant from each c-overlapped mutant set. With the results in Table 1, for example, four sets of c-overlapped mutants exist for the $t1$ test case, indicating that executing only four mutants instead of 15 against this test case is sufficient to determine the results of 15 mutants. Likewise, executing four mutants against the $t2$ test case and three mutants against the $t3$ test case is also sufficient.

4. Expression-level clustering C-overlapped mutual mutants

In our previous study (Kim et al., 2013), we reduced the mutation cost for Java programs by avoiding the execution of unreachable and expression-level weakly live mutants. Simple experimental result showed that excluding unreachable mutants reduced the number of mutant executions to fewer than 50% of the total and excluding weakly live mutants reduced the number of mutant executions by about 70%. This paper presents a technique that further reduces the cost of mutation testing by executing only one mutant from each c-overlapped mutant set, clustered based on the expression-level weakly killed mutants.

4.1. Description of our approach

The basic process of mutation testing involves generating and running mutants. Our approach introduces an additional filtering phase between generating and running mutants to reduce the number of mutants executed.

Fig. 2 shows an overview of our approach, which includes three phases: (1) mutant generation, (2) mutant filtering, and (3) strong mutation. Below, we describe how the three phases operate together.

4.1.1. Mutant generation phase

In our approach, mutants are not generated as individual programs; instead, they were encoded as two different programs: a *serialmutant* and a *metamutant*. The 'mutant generation' phase creates these two programs from the original program. The *serialmutant* has been discussed extensively in the paper of Kim et al. (2013). The serialmutant is a specialized program to conduct weak mutation against all mutants in only one execution. It executes original expressions and every mutated expression on the execution path of a test case. It was originally defined to conduct expression-level weak mutation and to produce a list of weakly live mutants. This paper extends the serialmutant to extract the information required to identify expression-level c-overlapped mutant sets among weakly killed mutants. This paper uses the term 'expression-level' because determining c-overlapped mutants is conducted right after each execution of mutated expressions.

The serialmutant is best explained using a simple example. Consider the *arithmetic operator replacement (AOR)* operator. The AOR operator replaces each occurrence of an arithmetic operator with each of the other possible arithmetic operators. Applying this rule to the assignment statement '`C = A − B;`' yields the following four mutated statements:

```
C = A + B;    // mutant 1
C = A * B;    // mutant 2
C = A / B;    // mutant 3
C = A % B;    // mutant 4
```

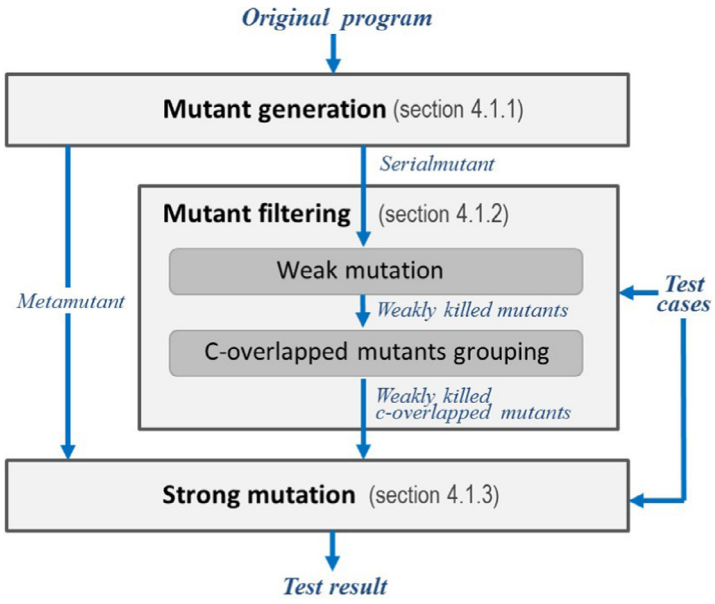

Fig. 2. Overview of our approach.

Let's define that a *change point* is a location in the program at which a mutation can be applied. In the above example, the location of the '-' operator is a change point. If we consider the `cpIndex` variable as the index of the change point, the original statement is changed below in the serialmutant code.

//changed statement in a *serialmutant*

```
C = SerialAOR(A, B, MINUS, cpIndex);
```

where *SerialAOR* is a *serialprocedure* of the AOR mutation operator. The left operand, right operand and arithmetic operator of the original expression are rewritten to parameters of the function call to the serialprocedure.

Fig. 3 shows the `SerialAOR` function, one of the serialprocedures for the AOR mutation operator. This function has four parameters: a left operand, a right operand, the type of the original arithmetic operator, and a change point index. In a serialprocedure, the original arithmetic expression is executed first (line 14), then its mutated versions are executed in sequence (lines 15–22). As in our previous approach (Kim et al., 2013), the identification (ID) number of a mutant whose intermediate result differs from that of the original expression is reported as an expression-level weakly killed mutant (line 20). In the example code, the ID number of a mutant is generated using the `getMutantID` function. In our extended approach, two values, the index of the change point and the result of the mutated expression, are

```
1   int arithmeticOP[] = {PLUS,MINUS,MULTIPLY,DIVIDE,MOD};
2
3   float AOR(float left_op, float right_op, int operator_type)  {
4     switch(operator_type){
5       case PLUS    : return left_op + right_op;
6       case MINUS   : return left_op - right_op;
7       case MULTIPLY: return left_op * right_op;
8       case DIVIDE  : return left_op / right_op;
9       case MOD     : return left_op % right_op;
10    }
11  }
12
13  float SerialAOR(float left, float right, int original_op, int cp_index) {
14    float original_result = AOR(left, right, original_op);
15     for(int i=0;i<arithmeticOP.length;i++){
16       if(original_op != arithmeticOP[i]) {
17         float mutant_result = AOR(left, right,  arithmeticOP[i]);
18         if( original_result != mutant_result)
19           String mutant_ID = getMutantID(cp_index, arithmeticOP[i] );
20           report(cp_index, mutant_ID, mutant_result));
21         }
22       }
23     return original;
24   }
```

Fig. 3. The `SerialAOR` serialprocedure for the `float` type.

[Original Code]

```
public class Triangle {
  public static TriangleType classify(int a, int b, int c) {
    int trian;
    if (a <= 0 || b <= 0 || c <= 0) return INVALID;
    trian = 0;
    if (a == b) trian = trian + 1;
    if (a == c) trian = trian + 2;
    if (b == c) trian = trian + 3;
    if (trian == 0) {
       if (a + b < c || a + c < b || b + c < a)
            return INVALID;
       else return SCALENE;
    }
    if (trian > 3) return EQUILATERAL;
    if (trian == 1 && a + b > c) return ISOSCELES;
    else if (trian == 2 && a + c > b) return ISOSCELES;
    else if (trian == 3 && b + c > a) return ISOSCELES;
    return INVALID;
    }
}
```

[Serialmutant Pseudocode]

```
public class Triangle {
  public static TriangleType classify(int a, int b, int c) {
    int trian;
    if( SerialCOR(SerialCOR(SerialROR(a,0,<=,1)
         ,SerialROR(b,0,<=,2),||,3),SerialROR(c,0,<=,4),||,5))
       return INVALID;
    trian = 0;
    if(SerialROR(a,b,==,6)) trian = SerialAOR(trian,1,+,7);
    if(SerialROR(a,c,==,8)) trian = SerialAOR(trian,2,+,9);
    if(SerialROR(b,c,==,10)) trian = SerialAOR(trian,3,+,11);
    if(SerialROR(trian,0,==,12)) {
       if ( SerialCOR(SerialCOR(SerialROR(SerialAOR(a,b,+,13),c,<,14)
               ,SerialROR(SerialAOR(a,c,+,15),b,<,16),||,17)
               ,SerialROR(SerialAOR(b,c,+,18),a,<,19),||,20) )
            return INVALID;
       else return SCALENE;
    }
    if(SerialROR(trian,3,>,21)) return EQUILATERAL;
    if(SerialCOR(SerialROR(trian,1,==,22)
           ,SerialROR(SerialAOR(a,b,+,23),c,>,24), &&,25))
         return ISOSCELES;
    else if(SerialCOR(SerialROR(trian,2,==,26)
           ,SerialROR(SerialAOR(a,c,+,27),b,>,27), &&,28))
         return ISOSCELES;
    else if(SerialCOR(SerialROR(trian,3,==,29)
         ,SerialROR(SerialAOR(b,c,+,30),a,>,31),&&,32))
         return ISOSCELES;
    return INVALID;
    }
}
```

Fig. 4. The original code and the serialmutant pseudocode for the `Triangle` program.

additionally reported to cluster weakly killed c-overlapped mutants later. In the code, these values are represented as the `cp_index` variable and the `mutant_result` variable. After executing all mutants associated with a given change point, the original result is returned as a result of the `serialAOR` serialprocedure (line 23).

Fig. 4 shows the serialmutant pseudocode generated from the well-known `Triangle` program, which classifies triangles based on lengths of the sides. The figure also shows the original code[1] of the `Triangle` program. We consider this simple program is good enough to explain how the serialmutant is created. Every target expression to be mutated of the original code is converted into a serialprocedure call in the serialmutant code. The first and the second parameters represent the left operand and the right operand

[1] https://github.com/david-schuler/javalanche/wiki/Example-Triangle.

Table 2
Mutants and evaluated values for the test case $t1$ ($A = 4$, $B = 2$).

Expression ID (Line number)	Mutant ID	Mutated code	Intermediate result	Weakly killed or live
1 (3)	ROR_1	`A > B`	True	Live
1 (3)	ROR_2	`A ≥ B`	True	Live
1 (3)	ROR_3	`A < B`	False	Killed
1 (3)	ROR_4	`A ≤ B`	False	Killed
1 (3)	ROR_5	`A == B`	False	Killed
1 (3)	ROR_6	`True`	True	Live
1 (3)	ROR_7	`False`	False	Killed
2 (4)	AOR_1	`A - B`	2	Killed
2 (4)	AOR_2	`A / B`	2	Killed
2 (4)	AOR_3	`A * B`	8	Killed
2 (4)	AOR_4	`A % B`	0	Killed
3 (6)	AOR_5	`A + B`	*Unreachable code*	Live
3 (6)	AOR_6	`A - B`	*Unreachable code*	Live
3 (6)	AOR_7	`A / B`	*Unreachable code*	Live
3 (6)	AOR_8	`A % B`	*Unreachable code*	Live

$WK - CoM_E(t1) = \{\{ROR_3, ROR_4, ROR_5ROR_7\}, \{AOR_1, AOR_2\}, \{AOR_3\}, \{AOR_4\}\}$.

of a binary expression. For the third parameter, we directly write the operator of a binary expression to increase the understandability, but its form should be changed to be compilable. The last parameter of the serialmutant call means a change point index of an expression. Compound expressions are converted to nested serial-procedure calls. This example is actually used as our experimental program, which will be shown in Section 5.

4.1.2. Mutant filtering phase

The 'mutant filtering' phase first conducts expression-level weak mutation using the extended serialmutant, then clusters c-overlapped mutants among the weakly killed mutants.

In our approach, c-overlapped mutants are identified by comparing mutated values of an expression among the weakly killed mutants. For this purpose, the serialmutant is extended to report weakly-killed mutants along with the value of their mutated expression. In the code example shown in Fig. 3, the `'report(...)'` method call of the 20th line performs this role. The first parameter of the function indicates the identification (ID) value of the change point, the second parameter indicates the ID value of the mutant, and the third parameter indicates the value of the mutated expression. These three values are saved as a 3-tuple. Identifying c-overlapped mutants is straightforward using the set. Pairs of a mutant ID and a mutated value are extracted for a given change point. The mutated values of pairs are then compared. If the mutated values are identical, the mutants given by the mutant IDs are c-overlapped. Because our approach clusters mutants generated at the same change point, every mutant belongs to only a cluster.

Table 2 presents the information obtained by the serialmutant using the $t1$ test case of the example code shown in Fig. 1. Four mutants (i.e., AOR_5, AOR_6, AOR_7, and AOR_8) have an unreachable mutated code. Three mutants (i.e., ROR_1, ROR_2, and ROR_6) have a reachable but weakly live mutated code. Executing these seven mutants would be meaningless and they are thus excluded from the determination of c-overlapped mutants.

Consider $WK\text{-}CoM_E(t)$ to be a set of weakly killed c-overlapped mutants for an expression under the test case t. For the $t1$ test case, the $WK\text{-}CoM_E$ set can then be calculated as the bottom of Table 2. Execute only one mutant from the c-overlapped cluster is sufficient because all the mutants in the cluster are expected to produce the same output. That is, inducing strong mutation with only four mutants (e.g., ROR_3, AOR_1, AOR_3, and AOR_4) is sufficient to derive the results in Table 2. To sum up, the number of mutants that need to be executed for strong mutation is reduced from 15 to 11 by reachability analysis, reduced further from 11 to 8 by weak mutation, and reduced yet again from 8 to 4 by c-overlapped mutant clustering.

```
MetaAOR(left, right, int original_op) {
  switch(mutant_id){
  // where mutant_id is a global variable
    case 1: result=AOR(left,right,PLUS);
    case 2: result=AOR(left,right,MULTIPLY);
    case 3: result=AOR(left,right,DIVIDE);
    case 4: result=AOR(left,right,MOD);
    default:result=AOR(left,right,original_op);
  }
  return result;
}
```

Fig. 5. The pseudocode of a `MetaAOR` procedure.

Some expressions may be executed more than once during a test case run. Expressions inside a loop are one such example. Accurate c-overlapped sets can only be extracted by considering values for each execution of the expressions. With this reason, the proposed approach considers mutants to be c-overlapped if values for every execution of the expression inside a loop are same.

After identifying c-overlapped mutant sets among the weakly-killed mutants, we selected a representative mutant from each c-overlapped mutant set. Strong mutation is conducted with the representatives against a given test. If the representative mutant turns out to be strongly-killed, each mutant in the c-overlapped mutant set, including the executed representative, is reported as a strongly-killed mutant.

4.1.3. Strong mutation phase

The metamutant, which we adapt from the MSG technique (Kim et al., 2013), was used to conduct strong mutation. As a serialmutant, the metamutant encodes every mutant into a program but it simulated only one mutant at an execution. For the `'C = A + B;'` statement used in Section 4.1.1, it can be changed to `'C = MetaAOR(A,B,PLUS);'` in the metamutant code. Fig. 5 shows the `MetaAOR` function.

The 'strong mutation' phase of our approach executes the metamutant against the weakly killed c-overlapped sets produced by the 'mutant filtering' phase. At this phase, the metamutant should be executed as the number of c-overlapped sets by changing the identification number of mutants. This is a feature that is different from the serialmutant, where only one execution is required for simulating whole mutants during weak mutation. It is because a metamutant simulates only one mutant according to the given identification number of a mutant, as we can see from the code of Fig. 5, Therefore, the serialmutant and the metamutant should share a mutant ID scheme to execute the same mutant. Please note that the MSG method (Kim et al., 2013) using a metamutant is not the only solution for our 'strong mutation' phase. Other strong mutation methods such as the method compiling each mutant into a separate file can be used instead of the MSG method if they are possible to share the same mutant ID scheme with a serialmutant.

Although our strong mutation process executes not whole mutants but representative mutants, the calculation of mutation score consider whole mutants. For unexecuted mutants, whether they are killed or not is determined as the result of the representative mutant of the same cluster.

4.2. Limitations

The key to the proposed approach is to reduce the number of mutants that must be executed by clustering mutants. There can be various methods to cluster mutants. Our approach uses an

expression-level clustering method because of the performance. As a result, only one mutation operator is related for each cluster. This fact can be figured out with the result Table 2. If we widen the clustering level (e.g. statement-level or method-level), several mutation operators can be considered together during clustering and then more mutants will be grouped as a cluster. However, it can increase the clustering time unwantedly.

More than two mutants should be generated for the expression at a given change point to obtain the desired effects of clustering. However, some mutation operators generate only one mutant at a given change point. The COR mutation operator, which replace conditional operators, '&&' and '||', is one such example. Consider the expression 'A&&B'. For this case, only one mutated expression, 'A||B', is possible using the COR mutation operator; thus, it is meaningless to cluster the resulting mutants. Applying our approach to mutation operators that produce only one mutant at a given change point is not practical.

Our approach was designed for first-order mutants, which involve only a mutated expression for a mutant. Higher-order mutants contain more than one mutated expression. To handle higher-order mutants, an intermediate program state changed by a mutated expression executed first should be able to propagate to the next mutated expressions. However, a serialmutant devised in our approach does not change or propagate an intermediate program state by a mutated expression.

5. Experimentation

We conducted experiments to determine whether our approach efficiently reduced the cost of mutation testing. The experiments evaluated the approach using six Java programs.

5.1. Experimental setup

This section describes the mutation tools, target programs, mutation operators, and analysis method used in the experiment.

5.1.1. Tools

The purpose of the experiment was to compare the cost of mutation testing with two existing dynamic mutant filtering approaches: the approach filtering unreached mutants (Schuler and Zeller, 2009) and the approach filtering weakly live mutants (Kim et al., 2013). For this purpose, we implemented three versions of MuJava: MuJava$_{RE}$, MuJava$_{WS}$, and MuJava$_{CO}$. These represent mutation systems that use reachability analysis (Schuler and Zeller, 2009), the weak and strong approaches (Kim et al., 2013), and our approach, respectively. The tools were implemented based on Ma et al. (2006). However, the previous MuJava tool contained some errors in generating AOR and LOR mutants, and we fixed the errors for the experiment.

To achieve an unbiased comparison, the same implementation was used for common behavior such as conducting strong mutation. Each of the three tools consisted of a mutant generator and a mutant executor. The mutant executor further consisted of a mutant filter and a strong mutation executor. For MuJava$_{RE}$, the mutant generator generated only a metamutant, and the mutant filter discerned which mutants were reached by each test case. For MuJava$_{WS}$, the mutant generator generated a metamutant and a serialmutant, and the mutant filter discerned the weakly killed mutants. For MuJava$_{CO}$, the mutant generator generated a metamutant and an extended serialmutant, and the mutant filter discerned weakly killed c-overlapped mutants.

5.1.2. Target programs

Six Java programs are used in this experiment. First, two Java programs from the paper of Kim et al. (2013) were used: the `Triangle` program, a very simple and widely used program, and the `gnu.regexp` package. Test cases of the paper were also used to conduct mutation testing with these two programs. Second, we used four open-source Java applications which provide JUnit test cases. Nowadays, some Java applications provide their test cases of JUnit format. To obtain practical data, we searched open-source applications whose test cases were provided and used them in this experiment.

Table 3
Mutation operators for Java.

Operator	Description
AOD	Arithmetic Operator Deletion
AOI	Arithmetic Operator Insertion
AOR	Arithmetic Operator Replacement
ASR	Assignment Short-cut operator Replacement
BOD	Bitwise Operator Deletion
BOI	Bitwise Operator Insertion
BOR	Bitwise Operator Replacement
COD	Conditional Operator Deletion
COI	Conditional Operator Insertion
COR	Conditional Operator Replacement
ROR	Relational Operator Replacement
SOR	Shift Operator Replacement

5.1.3. Applied mutation operators

Mutation operators are usually developed considering the syntax of a programming language. Therefore, different programming languages have a different set of mutation operators. This paper used Java mutation operators from our previous paper (Kim et al., 2013), which were designed by adapting the selective mutation (Wong et al., 1994; Offutt et al., 1996) and the mutation subsumption (Kaminski et al., 2013; Just et al., 2012) approaches. However, this paper renamed the LOD, LOI, and LOR operators, which were used in the previous paper, to the BOD, BOI, and BOR mutation operators because logical operators are renamed as bitwise operators according to current Java tutorials. Table 3 shows the list of Java mutation operators considered in this paper.

As described in Section 4.2, our approach targets mutation operators that produce more than two mutants at a given change point. Among the mutation operators in Table 3, five mutation operators listed below satisfy the condition. For the ROR mutation operator, we adapted the mutation subsumption approach (Kaminski et al., 2013; Just et al., 2012) to remove redundancy between mutants.

- *AOR* : Arithmetic Operator Replacement
  Replace a binary arithmetic operator to other binary arithmetic operators. Java supports five arithmetic operators for floating-point and integer numbers; (1) +, (2) −, (3) ∗, (4) /, and (5) %.
- *ASR* : Assignment Short-Cut Operator Replacement
  Replace a short-cut assignment operator to other short-cut assignment operators of the same kind. For assignment operators, 11 short-cut operators; (1) +=, (2) −=, (3) ∗=, (4) /=, (5) %=, (6) &=, (7) | =, (8) ∧=, (9) <<=, (10) >>=, and (11) >>>=, are defined.
- *BOR* : Bitwise Operator Replacement
  Replace a binary bitwise operator to other binary bitwise operators. Java provides three binary bitwise operators; (1) &, (2) |, (3) ∧.
- *ROR* : Relational Operator Replacement
  Replace a relational operator to other relational operators or a relational expression to `true` or `false`. Java provides six relational operators; (1) >, (2) >=, (3) <, (4) <=, (5) ==, and (6) ! =. Instead of replacing all these relational operators, the mutation subsume study by Kaminski et al. (2013) suggested that

below three replacements specific to each relational operator are sufficient.

$>$	$\rightarrow$	$>=, !=,$ `false`	$>=$	$\rightarrow$	$>,$ `==, true`
$<$	$\rightarrow$	$<=, !=,$ `false`	$<=$	$\rightarrow$	$<,$ `==, true`
$!=$	$\rightarrow$	$<, >,$ `true`	`==`	$\rightarrow$	$<=, >=,$ `true`

- *SOR* : Shift Operator Replacement
  Replace a bit shift operator to other bit shift operators. Java provides three binary bit shift operators; (1) $<<$, (2) $>>$, (3) $>>>$.

This paper conducts experiments with these five mutation operators.

5.1.4. Analysis method

The experiment conducted three kinds of analysis. First, we examined the total number of mutants executed during strong mutation. The number of mutants executed was directly related to the cost of mutation testing. The smaller the number of mutant executions, the lower the cost of mutation testing. This experiment does not count the number of weak mutation executions because only an execution of the serialmutant is required for each test case; that is, the number of weak mutation executions is the same as the number of test cases. This number is much smaller than the number of strong mutation executions so it is negligible (Kim et al., 2013).

Next, to see how much cost reduction could be obtained with our approach, we analyzed the time required to execute mutants. The time is calculated as the sum of the time for filtering mutants and the time for running the filtered mutants with strong mutation. The time is measured under a computer of Intel Core i7-2600K 3.4 GHz with 8GB memory.

Finally, we compared a mutation score obtained by executing whole mutants without clustering and an estimated mutation score obtained by executing representative mutants from each cluster. Our approach uses the term, 'estimated mutation score', because our approach considers whole mutants in calculating a mutation score though some mutants are not actually run. For unexecuted mutants, whether they are killed or not is determined as the result of the representative mutant of the same cluster.

Let's assume that there is a set of 100 non-equivalent first-order mutants, and then, our approach produces 40 mutant clusters. If 30 representative mutants were killed among the 40 clusters, our approach *does not* calculate a mutation score as 75% (30/40). Instead, it *does* calculate a mutation score as the number mutants included in the 30 clusters divided by 100, the total number of mutants. If 80 mutants are included in the 30 clusters, the estimated mutation score will be 80% (80/100). If the estimated mutation score will be the same with the real mutation score, it means that our approach efficiently reduces the cost of mutation testing without decreasing the effectiveness of mutation testing using the whole set of mutants.

5.2. Deep analysis using small Java programs

The `Triangle` and the `gnu.regexp` programs were used to analyze our method. The `Triangle` program is shown at Fig. 4 in Section 4.1. It is implemented as a simple class, where the number of code lines was 20. The `gnu.regexp` package, a pure Java implementation of a traditional (non-POSIX) NFA regular expression engine, consists of 28 classes and one interface and comprises 1975 lines of code. Because the size of both programs is not big, we can discriminate equivalent mutants by hands.

Although we used the same Java applications with the paper of Kim et al. (2013) to generate mutants, this paper would show a little different number of mutants from the paper. It is because we fixed some errors of the previous MuJava tool, as we mentioned in Section 5.1.1.

Table 4
The number of non-equivalent mutants of the `Triangle` program.

Mutation operator	AOR	ROR	Total
Number of mutants	32	41	73

5.2.1. Analysis using the Triangle program

Our mutation systems created 87 mutants (36 AOR mutants and 51 ROR mutants) for the `Triangle` program, but fourteen of these mutants were turned out to be equivalent. A total of 73 non-equivalent mutants were shown in Table 4, but no mutant was generated by the ASR, BOR and SOR mutation operators.

This experiment used 15 test cases whose mutation score was 100%. Every test case was effective; that is, it kills at least one mutant. Table 5 shows how many mutants was killed by each test case. It also shows the number of mutants that were reached, weakly killed, weakly killed c-overlapped, and strongly killed, for each test case. In this analysis, each test case was executed against all 73 non-equivalent mutants respectively. Many of the reached mutants turned out to be weakly live; thus, filtering out weakly live mutants proved to be efficient. In addition, the weakly killed mutants were clustered into c-overlapped mutant sets to a satisfactory degree. Sometimes, the number of c-overlapped mutant sets was smaller than the number of strongly killed mutants, demonstrating that our approach effectively reduced the number of mutants that needed to be executed.

The last column of Table 5 shows the real mutation score and the estimated mutation score. The real mutation score means a mutation score obtained by executing whole mutants without applying our clustering approach. For all cases, the real mutation scores and the estimated mutation scores were the same. It means that that our approach produces the same mutation score with that of mutation testing using a full set of mutants. This result shows that our approach retains the effectiveness of mutation testing using a full set of mutants.

Table 6 lists the numbers of mutant executions with strong mutation against each mutation operator, for each mutation tool. In contrast to the result of Table 5, where each test case was executed respectively against entire mutants, the result of Table 6 was obtained by executing the t1–t15 test cases all together. Each test case was executed in sequence against only live mutants; that is, strongly killed mutants from the previous test cases were not executed with the later test cases. The 'Number of Mutant Executions' column represents the total number of mutant executions allowing duplication. For example, if a mutant is executed by both test cases 't1' and 't2', the number of mutant executions is two. Because a mutant is executed repeatedly until it is killed, it is no wonder that the number of mutants executions exceeds the number of generated mutants. For MuJava$_{CO}$, the number of mutant executions represents the number of weakly killed c-overlapped sets because it executes only one mutant from each set.

From the last row in Table 6, MuJava$_{WS}$ reduced the number of executed mutants to about 60% of the number executed for MuJava$_{RE}$ in total. MuJava$_{CO}$ showed the best performance among the three tools used, requiring 73 executions. It reduced the number of executed mutants to about 50% of MuJava$_{RE}$ executions and about 20% more reduction of MuJava$_{WS}$ executions. This simple study showed that our approach well reduced the number of mutants that needed to be executed.

Table 7 lists the mutant execution times for each MuJava system. The mutant execution time is the sum of a time for filtering

Table 5
The number of mutant executions for each test case of the `Triangle` program.

Test case	Reached mutants	Weakly killed mutants	Weakly killed c-overlapped set	Strongly killed mutants	Mutant score (%) Real	Estimated
t1	42	24	21	21	28.8	28.8
t2	42	24	19	5	6.8	6.8
t3	9	3	3	1	1.4	1.4
t4	36	18	14	11	15.1	15.1
t5	38	18	12	16	21.9	21.9
t6	44	23	14	10	13.7	13.7
t7	41	20	13	16	21.9	21.9
t8	44	22	14	18	24.7	24.7
t9	3	1	1	1	1.4	1.4
t10	6	2	2	1	1.4	1.4
t11	28	14	11	5	6.8	6.8
t12	35	19	16	5	6.8	6.8
t13	42	24	18	17	23.3	23.3
t14	44	22	14	15	20.5	20.5
t15	44	22	14	15	20.5	20.5

Table 6
The number of mutant executions with the *t1–t15* test cases.

Mutation operator	The number of generated mutants	Number of mutant executions		
		MuJava$_{RE}$	MuJava$_{WS}$	MuJava$_{CO}$ **(Our approach)**
AOR	32	49	47	37
ROR	41	96	43	36
Total	73	145	90	73

Table 7
Time required for executing mutants of the `Triangle` program (unit: second).

Approach	Mutant execution time		
	Mutant filtering	Strong mutation	Total
MuJava$_{RE}$	1.62	24.55	26.17
MuJava$_{WS}$	3.78	13.21	16.99
MuJava$_{CO}$	3.79	10.97	14.76

mutant to be fully executed and a time for conducting strong mutation with the filtered mutants. The MuJava$_{RE}$ showed the fastest performance in mutant filtering; however, it is time for strong mutation was the slowest of the three systems. The time required for strong mutation holds significant weight in calculating the total mutant execution time. MuJava$_{CO}$ showed the best performance. MuJava$_{CO}$ showed mutant filtering times similar to those MuJava$_{WS}$ because our approach slightly extended the serialmutant functionality of the MuJava$_{WS}$. By introducing such a simple small extension of the mutant filtering process, our approach reduced the time required for strong mutation. Although our approach displayed the slowest preprocessing time for weak mutation, the result showed that reducing the number of mutants that need to be executed with strong mutation proved an efficient way to reduce the mutation cost.

5.2.2. Analysis with the `gnu.regexp` package program

The 62 test cases from our previous study (Kim et al., 2013) were used in this experiment. Table 8 shows the number of mutants that were reached, weakly killed, weakly killed c-overlapped and strongly killed for each test case. We can see that every test case was designed to kill at least one mutant.

We investigated real mutation scores and estimated mutation scores for each test case of the `gnu.regexp` package. The last column of Table 8 shows the result. For all cases, the real mutation scores and the estimated mutation scores were the same. This result demonstrates that our approach retains the effectiveness of mutation testing using a full set of mutants in this experiment.

Table 9 shows a mutation score and the number of mutant executions with strong mutation testing when the 62 test cases were executed all together. A total of 3,594 mutant executions occurred using the MuJava$_{RE}$ system. MuJava$_{WS}$ reduced the number significantly, yielding 1496 mutant executions. This result shows that discerning weakly live mutants in advance and avoiding their execution with strong mutation was effective in reducing the number of mutant executions. Our approach further reduced the number and required only 1185 executions. Generally, the number of mutant executions was much larger than the number of mutants generated because each mutant was executed repeatedly until it was killed by a test case. However, the number of mutant executions required by our approach was not significantly more than the total number of generated mutants, 1016. Considering that, the result demonstrates that clustering weakly killed mutants was effective in reducing the number of mutants to be executed.

We investigated the final results among mutants in the same weakly killed c-overlapped set. Every mutants of the same set produced the same results with strong mutation, which demonstrates that our approach did not alter the mutation score of the `gnu.regexp` package. Actually, the estimated mutation scores of the `gnu.regexp` package obtained by our approach were the same of the real mutation scores of Table 9.

Table 10 lists the mutant execution times for each mutation system. It is no wonder that more time was required by the mutant filtering steps needed to extract more mutants for execution. As a result, the MuJava$_{CO}$ system required the longest preprocessing time among the three systems; however, the time required for strong mutation was smallest; as a consequence, the overall mutant execution time was the smallest in this case. The mutant execution cost of our approach was about half the cost of the MuJava$_{RE}$. The time required for strong mutation was proportional to the number of mutants executed, as shown in Table 9. These results show that reducing the number of mutants executed with strong mutation provides an efficient way to reduce the overall mutation cost, even though additional time is required to identify the mutants that should be executed.

Actually, the location where intermediate results of mutants are compared affects the number of mutants to be clustered. This papers adapt expression-level weak mutation (Offutt and Lee, 1994)

Table 8
The number of executed mutants for each test case of the `regexp` program.

Test case	Reached mutants	Weakly killed mutants	Weakly killed c-overlapped set	Strongly killed mutants	Mutant score (%) Real	Estimated
t1	182	123	92	93	9.7	9.7
t2	182	122	91	80	8.3	8.3
t3	106	61	45	27	2.8	2.8
t4	240	158	120	125	13.0	13.0
t5	184	140	109	114	11.9	11.9
t6	164	122	92	98	10.2	10.2
t7	129	96	70	80	8.3	8.3
t8	178	127	91	90	9.4	9.4
t9	197	142	102	110	11.4	11.4
t10	246	173	128	131	13.6	13.6
t11	233	164	123	127	13.2	13.2
t12	233	162	121	131	13.6	13.6
t13	214	158	124	107	11.1	11.1
t14	206	158	128	120	12.5	12.5
t15	148	104	71	76	7.9	7.9
t16	154	108	74	66	6.9	6.9
t17	158	114	85	61	6.3	6.3
t18	141	108	83	84	8.7	8.7
t19	131	105	86	60	6.2	6.2
t20	44	28	19	18	1.9	1.9
t21	171	117	86	86	8.9	8.9
t22	258	108	86	67	7.0	7.0
t23	522	381	318	276	28.7	28.7
t24	101	74	53	40	4.2	4.2
t25	148	109	83	82	8.5	8.5
t26	148	112	86	88	9.2	9.2
t27	101	74	53	40	4.2	4.2
t28	148	109	83	82	8.5	8.5
t29	148	112	86	88	9.2	9.2
t30	140	103	73	77	8.0	8.0
t31	179	125	95	81	8.4	8.4
t32	179	125	95	81	8.4	8.4
t33	95	63	53	21	2.2	2.2
t34	111	65	43	30	3.1	3.1
t35	105	76	52	22	2.3	2.3
t36	60	28	19	18	1.9	1.9
t37	110	58	39	29	3.0	3.0
t38	83	46	46	42	4.4	4.4
t39	96	51	33	36	3.7	3.7
t40	75	52	40	20	2.1	2.1
t41	32	14	8	9	0.9	0.9
t42	78	45	32	15	1.6	1.6
t43	87	50	37	15	1.6	1.6
t44	176	123	87	89	9.3	9.3
t45	176	123	87	89	9.3	9.3
t46	295	191	140	109	11.3	11.3
t47	306	204	154	120	12.5	12.5
t48	306	204	154	120	12.5	12.5
t49	197	137	105	84	8.7	8.7
t50	212	116	94	88	9.2	9.2
t51	171	112	80	74	7.7	7.7
t52	219	145	101	72	7.5	7.5
t53	135	105	79	64	6.7	6.7
t54	12	6	4	2	0.2	0.2
t55	4	3	3	3	0.3	0.3
t56	34	20	14	11	1.1	1.1
t57	41	29	22	16	1.7	1.7
t58	44	32	26	22	2.3	2.3
t59	44	30	25	17	1.8	1.8
t60	41	25	20	14	1.5	1.5
t61	41	30	26	19	2.0	2.0
t62	45	35	31	21	2.2	2.2

and uses mutation operators that replace basic operators such as arithmetic operators and relational operators. However, those mutation operators produced small number of candidate mutants subject to our clustering method; sometimes, one or two mutants were weakly killed for an expression. If we use other methods which increase the number of candidate mutants to be compared, the number of fully executed mutants could be reduced more.

5.3. Experiments using open-source applications providing JUnit test cases

To provide objective and practical data, we conducted an additional comparison work using real-world Java applications which provide JUnit test cases. For that, we searched open source programs providing JUnit test cases and adapted our tool to understand JUnit test cases. We chose four open-source applications shown in Table 11 and conducted mutation testing using the test cases included in the applications.

The sizes of target applications were various from about 3000 lines to about 20,000 lines. Thousands of mutants were generated for each application. Some may wonder why the joda-time applications produced mutants less than expected. The application uses primitive operators such as arithmetic and relational operators much less than other applications.

In this experiment, equivalent mutants were not discriminated because they were filtered out as weakly live mutants, thus would not affect the efficiency of our approach, represented as MuJava$_{CO}$. This experiment focuses on the cost reduction obtained by clustering weakly killed mutants. As we know, equivalent mutants cannot not be included in a set of weakly killed mutants. Please note that equivalent mutants will affect the efficiency of the weak and strong mutation approach, represented as MuJava$_{WS}$. However, investigating the efficiency of MuJava$_{WS}$ is out of the scope of this paper.

Table 12 shows a mutation score and the number of executed mutants against those applications with the three mutation systems: MuJava$_{RE}$, MuJava$_{WS}$, and MuJava$_{CO}$.[2]

The three MuJava systems produced the same mutation score, but the number of mutant executions was different for each system. By filtering out weakly live mutants, many of the mutants were excluded from strong mutation testing with MuJava$_{WS}$. However, the `MuJava`$_{WS}$ system was not as useful for reducing the number of AOR mutant executions because most of the AOR mutants filtered by `MuJava`$_{RE}$ turned out to be weakly killed. On the other hand `MuJava`$_{CO}$ system decreased the number of AOR mutant executions by clustering c-overlapped mutants. The `MuJava`$_{CO}$ system reduced the number of mutant executions for other mutants as well as AOR mutants. This result showed that our approach effectively reduced the cost of mutation testing. Actually, the reduction of mutant executions was similar to the result of previous experiments. Our approach, `MuJava`$_{CO}$, reduced the total number of mutant executions by about 10% more than `MuJava`$_{WS}$.

In this experiment, we did not measure the time required to generate and execute mutants. Instead, we only examine the number of executed mutants because the mutation cost is strongly affected by the number of mutants to be executed. We consider previous two experiments would be enough to explain mutation cost depend on the number of executed mutants. Actually, it took several hours (more than one day for the joda-time application) to conduct strong mutation testing with our approach for each target application. This reflects that to shorten the time required to execute mutation testing is still an important issue.

We also calculated the real mutation scores obtained by executing whole mutants and compared the real mutation scores with the mutation scores estimated by our approach. All the estimated mutation scores were the same with the real mutation scores, which demonstrated that our approach retained effectiveness in this experiment.

The mutation scores in the experiment were not high as we expected. If we eliminated equivalent mutants, the actual mutation

[2] JavaNCSS homepage=http://www.kclee.de/clemens/java/javancss/tool.

Table 9
The number of mutants executed for the `gnu.regexp` package.

Mutation operator	Mutation score ($\frac{\#\ of\ killed\ mutants}{\#\ of\ generated\ mutants}$)	Total number of mutant executions		
		MuJava$_{RE}$	MuJava$_{WS}$	MuJava$_{CO}$ (**Our approach**)
AOR	81.1 ($\frac{133}{164}$)	364	353	288
ASR	77.8 ($\frac{42}{54}$)	60	54	48
BOR	52.0 ($\frac{26}{50}$)	109	103	68
ROR	73.4 ($\frac{509}{693}$)	3061	986	781
Total	73.9 ($\frac{710}{961}$)	3594	1496	1185

Table 10
Time required to execute the mutants of the `gnu.regexp` package (unit: second).

Approach	Mutant execution time		
	Mutant filtering	Strong mutation	Total
MuJava$_{RE}$	49.01	826.75	875.76
MuJava$_{WS}$	94.51	424.08	518.59
MuJava$_{CO}$	102.10	341.14	443.24

Table 11
Target applications providing JUnit test case.

Java rogram	Number of classes	SLOC[a]	JUnit test case	Number of mutants (AOR, ROR, BOR, ASR)
num4j	86	2725	194	3356
joda-time	241	20,119	3,858	1697
jaxen	213	8428	524	4517
jdom	71	6435	187	4927

[a] SLOC (Source Line of Code) was calculated using the JavaNCSS.

scores would be a little higher. Actually, the JUnit test cases were not intended for a mutation testing and a high mutation score is not a necessary condition to evaluate our approach. To evaluate our approach properly, we excluded the execution of ineffective test cases, test cases that does not kill any mutant. However, please note that the number of test cases of the joda-time exceeds the number of mutants from Table 12, but the mutation score was not close to 100%. Actually, the test cases of the joda-time application kill at least one mutant when it runs against mutants individually. However, if every test case was executed all together and strongly killed mutants from the previous test cases were excluded from execution with the later test cases. In this case, some test cases will lose their effectiveness if mutants that will be killed by them were already killed by previous test cases. Because our experiment does not count unreached mutants at all, many of those ineffective mutants be filtered thus experimental results will not be affected seriously.

Some readers might expect more cost reduction for ROR mutants. Because we adapt the ROR mutation operator considering the mutant subsumption, the cost reduction was not big as shown in the example of Table 1 in Section 3. If we did not consider the mutant subsumption, more cost reduction would be achieved with our approach. Table 13 shows the number of ROR mutants generated under both the condition which considers the mutant subsumption and the condition which does not consider the mutant subsumption.

Table 14 shows the cost reduction of ROR mutants of Table 13. As shown in the table, without considering the mutant subsumption, the number of generated ROR mutants increased about twice. As a result, the number of mutant executions was increased also for each MuJava system. Especially, from the result of the MuJava$_{WS}$ system, we could easily see that the number of weakly killed mutants was increased a lot. Actually, the value of ROR

Table 13
Experiment without a mutation subsumption of the ROR mutation operator.

Type	The number of ROR mutants			
	num4j	joda-time	jaxen	jdom
Subsumption	900	765	2469	2589
No subsumption	2100	1785	5761	6041

Table 12
The number of mutants executed for real-world programs.

Java program	Mutation operator	Mutation score ($\frac{\#\ of\ killed\ mutants}{\#\ of\ generated\ mutants}$)	Total number of mutant executions		
			MuJava$_{RE}$	MuJava$_{WS}$	MuJava$_{CO}$ (**Our approach**)
num4j	AOR	71.6 ($\frac{1723}{2408}$)	2431	2388	2271
	ASR	70.8 ($\frac{34}{48}$)	80	39	37
	ROR	63.2 ($\frac{569}{900}$)	1308	704	659
	Total	**69.3 ($\frac{2326}{3356}$)**	**3819**	**3131**	**2967**
joda-time	AOR	80.7 ($\frac{665}{824}$)	54,508	50,447	43,799
	ASR	81.8 ($\frac{54}{66}$)	22,076	22,050	22,047
	BOR	66.7 ($\frac{28}{42}$)	14,743	6031	6028
	ROR	62.2 ($\frac{475}{765}$)	37,531	5328	4766
	Total	**72.1 ($\frac{1223}{1697}$)**	**128,858**	**83,856**	**76,640**
jaxen	AOR	55.2 ($\frac{117}{212}$)	2453	2400	1835
	ASR	0.0 ($\frac{0}{46}$)	0	0	0
	BOR	0.0 ($\frac{0}{46}$)	586	294	294
	ROR	33.3 ($\frac{821}{2469}$)	271,154	47,066	40,801
	Total	**34.9 ($\frac{2118}{4517}$)**	**274,193**	**49,760**	**42,930**
jdom	AOR	20.3 ($\frac{48}{236}$)	973	960	742
	ASR	8.3 ($\frac{4}{48}$)	8	8	6
	BOR	0.0 ($\frac{0}{46}$)	0	0	0
	ROR	32.8 ($\frac{849}{2589}$)	5348	1841	1676
	Total	**30.9 ($\frac{901}{2919}$)**	**6329**	**2809**	**2424**

Table 14
Experiment without a mutation subsumption of the ROR mutation operator.

Java program	Mutation score ($\frac{\#\ of\ killed\ mutants}{\#\ of\ generated\ mutants}$)		Total number of mutant executions		
			$MuJava_{\_RE}$	$MuJava_{\_WS}$	$MuJava_{\_CO}$ (**Our approach**)
num4j	Subsumption:	63.2 ($\frac{569}{900}$)	1308	704	659
	No subsumption:	70.3 ($\frac{1476}{2100}$)	2074	1806	1021
joda-time	Subsumption:	62.1 ($\frac{475}{765}$)	37,531	5328	4766
	No subsumption:	71.7 ($\frac{1280}{1785}$)	62,249	13,108	5032
jaxen	Subsumption:	33.3 ($\frac{821}{2469}$)	271,154	47,066	40,801
	No subsumption:	44.8 ($\frac{2580}{5761}$)	402,783	140,914	45,797
jdom	Subsumption:	32.8 ($\frac{849}{2589}$)	5348	1841	1676
	No subsumption:	45.0 ($\frac{2720}{6041}$)	9432	6305	2151

mutated expression is only two kinds: `true` and `false`, and the ROR mutation operators generates five mutated versions for an expression when it does not consider the mutant subsumption. Therefore, it is obvious that our approach is very effective to reduce ROR mutant executions without the mutant subsumption. In the case of the jaxen application, the effect of our approach was clear.

The result of Table 14 shows that our approach is still effective even if it is combined with other cost reduction approach. Actually, the goals of our approach and the mutant subsumption were very similar; that is, avoiding the execution of overlapped mutants. Therefore, the mutant subsumption approach excludes overlapped mutants in advance, which our approach might also discriminate. However, our approach can exclude more mutants by considering intermediate program states by each test case.

6. Conclusion

This paper presents an approach to reduce the cost of mutation testing by reducing the number of mutants that need to be fully executed with strong mutation. The idea underlying our approach is that a mutant can be equivalent to (an) other mutants, and clustering the mutants can reduce the number of executed mutants. The feasibility of implementing this approach was examined by introducing the concept of c-overlapped mutants, mutants that are expected to produce identical result under a test case. Our approach dynamically identified weakly killed c-overlapped mutants under a test case, and only one mutant from each c-overlapped cluster was executed with strong mutation. If the fully executed mutant was killed (or live), all remaining mutants of the same cluster were considered to be killed (or live).

Our approach clusters mutants only when their results were considered to be the same. Therefore, one benefit of our approach is that it is guaranteed to be nearly as effective as strong mutation testing, but with a lower execution cost. Our approach clusters mutants by comparing values of an innermost expression. In our experiment, every mutants in a cluster produced the same final result when they conduct strong mutation testing, which implies that our approach produces the same mutation score with strong mutation testing. We expect that mutants in the same cluster almost never produce different final results.

Experimental results showed that our method successfully reduced the number of executed mutants. Actually, the number of mutant executions was reduced well by avoiding the execution of weakly live mutants. Our approach further reduced the number of mutant executions by clustering weakly killed mutants: about 10% more reduction was achieved for the applications used in the experiments. Comparing the approach which executes only reached mutants, our approach reduced the number of mutants executions quite a lot; for some applications, our approach reduce the number of mutant executions more than a half. This result showed that clustering weakly killed mutants was effective in reducing the number of mutants to be executed.

One limitation of our approach was that it clustered mutants for an expression and thus it was applicable only to mutation operators generating more than two mutants. If we widen the comparison scope for the clustering, such as clustering mutants for a statement, more cost reduction can be achieved by clustering more mutants.

Acknowledgment

This work was supported by Institute for Information & communications Technology Promotion (IITP) Grant funded by the Korea government (MSIP) (10047067, Development of SMP RTOS technology for high performance and realtime multicore embedded systems).

References

Acree, A.T., 1980. On Mutation. Georgia Institute of Technology, Atlanta, GA (Ph.D. thesis).
Ammann, P., Delamaro, M.E., Offutt, J., 2014. Establishing theoretical minimal sets of mutants. In: Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation, pp. 21–30.
Budd, T.A., 1980. Mutation Analysis of Program Test Data. Yale University, New Haven, CT (Ph.D. thesis).
DeMillo, R.A., Lipton, R.J., Sayward, F.G., 1978. Hints on test data selection: help for the practicing programmer. IEEE Comput. 11 (4), 34–41.
Frankl, P.G., Weiss, S.N., Hu, C., 1997. All-uses vs mutation testing: an experimental comparison of effectiveness. J. Syst. Softw. 38 (3), 235–253.
Hussain, S., 2008. Mutation Clustering. King's College London, UK (Master's thesis).
Ji, C., Chen, Z., Xu, B., Zhao, Z., 2009. A novel method of mutation clustering based on domain analysis. In: Proceedings of the 21st International Conference on Software Engineering and Knowledge Engineering (SEKE'09). Boston, Massachusetts.
Jia, Y., Harman, M., 2009. Higher order mutation testing. Inf. Softw. Technol. 51 (10), 1379–1393.
Jia, Y., Harman, M., 2011. An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. 37 (5), 649–678.
Just, R., Kapfhammer, G., Schweiggert, F., 2012. Do redundant mutants affect the effectiveness and efficiency of mutation analysis? In: Proceedings of the 2012 IEEE 5th International Conference on Software Testing, Verification and Validation (ICST), pp. 720–725.
Kaminski, G., Ammann, P., Offutt, J., 2013. Improving logic-based testing. J. Syst. Softw. 86 (8), 2002–2012.
Kim, S.W., Ma, Y.S., Kwon, Y.R., 2013. Combining weak and strong mutation for a noninterpretive Java mutation system. J. Softw. Test. Verif. Reliab. 23 (8), 647–668.
Kurtz, B., Ammann, P., Delamaro, M.E., Offutt, J., Deng, L., 2014. Mutant subsumption graphs. In: Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation Workshops, ICSTW '14. IEEE Computer Society, Washington, DC, USA, pp. 176–185.
Ma, Y.S., Offutt, J., Kwon, Y.R., 2006. MuJava: a mutation system for Java. In: Proceedings of International Conference on Software Engineering, pp. 827–830.
Mateo, P.R., Usaola, M.P., Alemán, J.L.F., 2013. Validating second-order mutation at system level. IEEE Trans. Softw. Eng. 39 (4), 570–587.
Mathur, A.P., Wong, W.E., 1994. An empirical comparison of data flow and mutation-based test adequacy criteria. Softw. Test. Verif. Reliab. 4 (1), 9–31.
Offutt, A.J., Lee, A., Rothermel, G., Untch, R., Zapf, C., 1996. An experimental determination of sufficient mutation operators. ACM Trans. Softw. Eng. Methodol. 5 (2), 99–118.

Offutt, A.J., Lee, S.D., 1994. An empirical evaluation of weak mutation. IEEE Trans. Softw. Eng. 20 (5), 337–344.
Polo, M., Piattini, M., Garcia-Rodriguez, I., 2008. Decreasing the cost of mutation testing with second-order mutants. Softw. Test. Verif. Reliab. 19 (2), 111–131.
Schuler, D., Zeller, A., 2009. Javalanche: efficient mutation testing for Java. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the International Symposium on Foundations of Software Engineering, Amsterdam, The Netherlands, pp. 297–298.
Usaola, M.P., Mateo, P.R., 2010. Mutation testing cost reduction techniques: a survey. IEEE Softw. 27 (3), 80–86.
Weiss, S.N., Fleyshgakker, V.N., 1993. Improved serial algorithms for mutation analysis. In: Proceedings of the 1993 ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA '93:, pp. 149–153.
Wong, W.E., 1993. On Mutation and Data Flow. Purdue University, West Lafayette, IN (Ph.D. thesis).
Wong, W.E., Delamaro, M.E., Maldonado, J.C., Mathur, A.P., 1994. Constrained mutation in C programs. Proceedings of the 8th Brazilian Symposium on Software Engineering. Brazilian Computer Society, pp. 439–452.
Yao, X., Harman, M., Jia, Y., 2014. A study of equivalent and stubborn mutation operators using human analysis of equivalence. Proceedings of the 36th International Conference on Software Engineering, ICSE 2014. ACM, pp. 919–930.

Yu-Seung Ma received the B.S., M.S., and Ph.D. degrees in computer science from Korea Advanced Institute of Science and Technology (KAIST), Korea, in 1998, 2000 and 2005, respectively. In February 2005, she joined in the Real-time SW Research Team at Embedded Software Research Division of the Electronics and Telecommunications Research Institute (ETRI), Korea, where she is currently a senior researcher. Her research interests include program testing, mutation testing, and embedded software engineering.

Sang-Woon Kim received the B.S. degree in 2000, M.S. degree in 2002, and Ph.D. degrees in 2011, all in computer science at the Korea Advanced Institute of Science and Technology (KAIST), Korea. In April 2008, he joined FormalWorks, Inc. and worked on developing several verification and validation systems for automotive embedded software. His research interests are in the area of mutation testing, testing automation, and program analysis.