An Effective Iterative Metamorphic Testing Algorithm Based on Program Path Analysis*

Guowei Dong, Changhai Nie, Baowen Xu, and Lulu Wang
School of Computer Science and Engineering, Southeast University, China
{dgw, changhainie, bwxu}@seu.edu.cn

Abstract

Metamorphic testing(MT) is very practical and effective for programs with oracle problems, and many researches have been done in this field. In this article, we present a new iterative MT algorithm, APCEMSI, to avoid the blindness of existing MT methods. The test suites generated by APCEMSI satisfy a criterion which is defined upon program path analysis technique. Experiment result shows that APCEMSI could find errors effectively with much fewer test cases.

1. Introduction

Software testing can be used for finding and correcting errors in software. But sometimes, there are *oracle problems* [1] in testing, that is, it is difficult to decide whether the result of the program under test agrees with expected result. To solve this problem, Chen presented *Metamorphic Testing* [2]. This method tests software by checking relations among execution results, and does not call for expected outputs.

Two traits of MT are: (1) To check execution results, *Metamorphic Relations* (MR)[2] should be constructed. (2) To attain original test cases [2], MT should be used with other test case generating methods.

Chen reported on the MT of programs for solving partial differential equations [3]; Gotlieb developed an automated framework to check against a restricted class of MRs [4]; Zhou introduced ways to design MRs for non-numerical problems [5]; Tse applied metamorphic approach to the unit testing[6] and integration testing[7] of context-sensitive middleware-based applications; MT has also been used in testing SOA software [8,9]; Chen introduced how to select useful MRs by a case study [10]; Chen investigated the integration of MT with global symbolic evaluation[11] and fault-based testing[12]; Wu gave the evaluation and comparison of special case testing, MT with special and random test cases [13], and put forward an iterative MT technique [14]. These researches validate that MT is good for solving oracle problem, but most of them only consider program's function, so blindness is hard to avoid.

In this article, we define a MT criterion, *APCEM* based on program path analysis technique, and present a new iterative MT algorithm, *APCEMSI* to generate *APCEM* satisfied test suites. The experimental results depending on mutation analysis technique [15] proves the efficiency of this new algorithm.

This paper is organized as follow: in section 2, some concepts are introduced; in section 3, a MT criterion and the effective iterative MT algorithm are presented; in section 4, the efficiency of *APCEMSI* is proved by experiments; in section 5, this paper is concluded.

2. Concepts

Definition 1 (Metamorphic Relation [2]**)** Let I_1, I_2, …, I_n (n>1) be n different inputs of program P, if their satisfaction of relation r implies that their corresponding outputs $P(I_1)$, $P(I_2)$, …, $P(I_n)$ satisfy relation r_f, that is:

$$r(I_1, I_2, \ldots, I_n) \Rightarrow r_f(P(I_1), P(I_2), \ldots, P(I_n)) \qquad (1)$$

then (r, r_f) is called a *Metamorphic Relation*(MR) [2].

Definition 2 (Metamorphic Domain) Let D be program P's input domain, if $D' \subseteq D$, and we could do MT on D', then D' is *Metamorphic Domain* of P, $D_{MT}(P)$ for short.

Definition 3 (Binary Metamorphic Relation) As to program P, if for $\forall I_1 \in \xi \subseteq D_{MT}(P)$, $\exists I_2 \in D_{MT}(P)$, and they make

$$r_b(I_1, I_2) \Rightarrow r_{bf}(P(I_1), P(I_2)) \qquad (2)$$

satisfied, then (r_b, r_{bf}) is a *Binary MR* of P, and its *Definition Domain* is ξ, recorded as $D_R((r_b, r_{bf}))$. I_1 is called *Original Input*(OI for short), and I_2 is the *Follow-up Input*(FUI for short) of I_1 based on (r_b, r_{bf}), signed as $I_2 = FU(I_1, (r_b, r_{bf}))$.

* This work was supported in part by the National Natural Science Foundation of China (60425206, 60633010, 60403016, 60503033), Natural Science Foundation of Jiangsu Province (BK2005060), High Technology Research Project of Jiangsu Province (BG2005032), Excellent Talent Foundation on Teaching and Research of Southeast University, and Open Foundation of State Key Laboratory of Software Engineering in Wuhan University.

For example, program $P(I)$ computes function $f(x)$, where $f(x)$ is:

$$f(x)=\begin{cases}0 & x\le 0\\ Sin(x) & x>0\end{cases}$$

According to Definition 2, $D_{MT}(P(x)) = (0, +\infty)$. We construct a binary MR mr_{Sin} for $P(I)$, where mr_{Sin} is:

mr_{Sin}: $(I_1+I_2=\pi/2,\ [P(I_1)]^2+[P(I_2)]^2=1)$

According to Definition 3, $D_R(mr_{Sin}) = (0, \pi/2)$. If we take $\pi/3$ as OI, then $FU(\pi/3, mr_{Sin})=\pi/6$.

Definition 4 (Applying Domain) Let mr be a binary MR of program P, if for $\forall I_1\in\delta\subseteq D_R(mr)$, no matter what mutation operators [16] are imported, I_1, $FU(I_1, MR)$ and their outputs will always satisfy mr, then δ is called the *Blind Domain* of mr, $D_{bl}(mr)$ for short, and $(D_R(mr)\text{-}D_{bl}(mr))$ is called the *Applying Domain* of mr, represented by $D_{app}(mr)$. Testing dose not make sense if OIs are selected from Blind Domain.

For example, program *Square*(*double, double*) is used for calculating rectangle square, where Square is:

```
double Square (double a, double b) {
        return a * b;
}
```

We construct a binary MR for this program:

$\boldsymbol{mr_{sq}}$: $((a', b') = (b, a),\ Square(a', b') = Square(a, b))$

According to Definition 4, $D_{bl}(mr_{sq}) = \{(a, b) \mid a = b\}$, and $D_{app}(mr_{sq}) = \{(a, b) \mid a \neq b\}$.

3. Effective Iterative MT Algorithm

Wu has proposed an n-iterative MT method [14] which uses MRs in a chain style. This method performs well in fault detection and test case generation, but it dose not assure that all codes are run, so some errors might not be found. The problem is solved in this paper.

3.1. Metamorphic Testing Criterion

In the rest of this article, mr_i $(i\in[1, m])$ denotes a binary MRs of program P; Each metamorphic test case, (mr_i, I_{i1}, I_{i2}) is a triple, where $I_{i2} = FU(I_{i1}, mr_i)$; TC denotes a metamorphic test suite for P; $Path_j$ $(j\in[1, n])$ represents an executable path of P, that is, $Path_j$ could be run when we execute P with some inputs; $DS(Path_j)$ is the set of inputs that execute $Path_j$, namely input sub-domain of $Path_j$; $ExcPath(I)$ indicates the path that is executed with input I.

APCEM(***A****ll-****P****ath* ***C****overage for* ***E****very* ***MR***): For $\forall\, mr_i$ and $\forall\, Path_j$, if $D_{app}(mr_i)\cap DS(Path_j)\neq\varnothing$, then $\exists\,(mr_i, I_{i1}, I_{i2})\in TC$, where $(ExcPath(I_{i1}) = Path_j)\vee(ExcPath(I_{i2}) = Path_j)$ is true. In this situation, the metamorphic test suite TC satisfies *APCEM*.

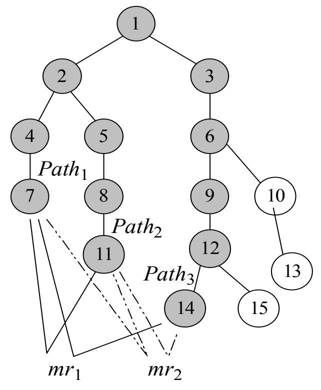


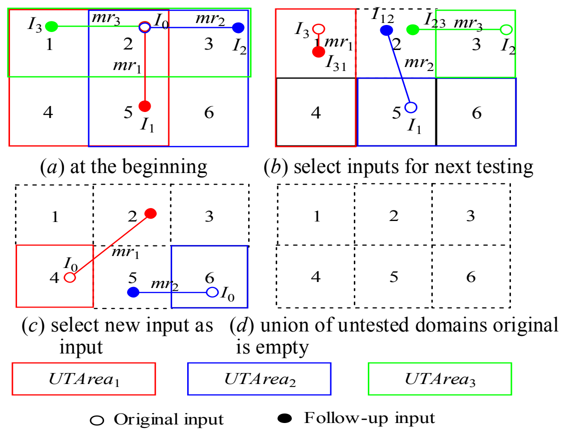
Fig.1. *APCEM*

For example, Fig.1 shows the Control Flow Graph of a program P, and it contains 5 executable paths, but only $Path_{1,2,3}$ are tested by MT method. We construct 2 binary MRs, $mr_{1,2}$ for P, and $D_R(mr_1) = D_R(mr_2) = D_{MT}(P) = \bigcup_{j=1}^{3} DS(Path_j)$. Four test cases, $T_1 = (mr_1, i_1, i_2)$, $T_2 = (mr_1, i_3, i_4)$, $T_3 = (mr_2, i_5, i_6)$ and $T_4 = (mr_2, i_7, i_8)$ are designed, and the execution paths of i_1-i_8 are $Path_1$, $Path_2$, $Path_3$, $Path_1$, $Path_2$, $Path_1$, $Path_2$, and $Path_3$ in turn. According to the definition of *APCEM*, the test suite $\{T_1, T_2, T_3, T_4\}$ meets this criterion.

3.2. Effective Iterative MT Algorithm

Based on *APCEM*, we present an algorithm named APCEMSI(*APCEM Satisfied Iterative Algorithm*). This algorithm tests program in an iterative style, that is, the FUIs are used as OIs for succeeding testing. But the FUIs that don't reach special conditions are discarded. This process doesn't stop until the test suite generated satisfies *APCEM*. Details are shown in Alg.1.

Because each executable path associates with an input sub-domain, the coverage of path could be reflected by that of corresponding domain. We instrument program and compute input sub-domains



(a) at the beginning
(b) select inputs for next testing
(c) select new input as original input
(d) union of untested domains is empty
UTArea₁
UTArea₂
UTArea₃
Original input
Follow-up input

Fig.2. Process of testing with *APCEMSI*

```
Initialize m domains $UTArea_i$ ($i \in [1, m]$),
which are used for storing untested domain of $mr_i$;
for $i \longleftarrow 1$ to $m$
    $UTArea_i = D_{app}(mr_i)$;
end for
Initialize queue $Q_{inp}$ to store inputs that could be used as
original inputs in next testing, initialize queue $Q_{temp}$ to
temporarily store some FUIs of an original inputs;
while ( $\bigcup_{i=1}^{m} UTArea_i \neq \varnothing$ )
    Randomly select $I \in \bigcup_{i=1}^{m} UTArea_i$ ;
    $Q_{inp}$.enqueue($I$);
    while ($Q_{imp}$ is not empty)
        $I = Q_{inp}$.dequeue();
        for $times \longleftarrow 1$ to $m$
            Randomly select a MR $mr_i$ ($i \in [1, m]$)
            that has not been chosen;
            if ($I \in UTArea_i$)
                $I' = FU(mr_i, I)$;
                if (($I'$ hasn't been put into $S_{inp}$ during
                        former testing) and ($ExcPath(I) \neq$
                        $ExcPath(I')$))
                    $Q_{temp}$.enqueue($I'$);
                end if
                Test program with ($mr_i$, $I$, $I'$);
                $UTArea_i = UTArea_i - DS(ExcPath(I))$
                        $- DS(ExcPath(I'))$;
            end if     /* end if ($I \in UTArea_i$)*/
        end for        /* end for $times \longleftarrow 1$ to $m$ */
        while ($Q_{temp}$ is not empty)
            $I' = Q_{temp}$.dequeue();
            if ($I' \in \bigcup_{i=1}^{m} UTArea_i$ )
                $Q_{inp}$.enqueue($I'$);
            end if
        end while      /* end while ($Q_{temp}$ is not empty)*/
    end while          /* end while ($Q_{imp}$ is not empty)*/
end while              /* end while ( $\bigcup_{i=1}^{m} UTArea_i \neq \varnothing$ )*/
```

Alg.1. *APCEMSI*

with the technique of symbolic execution [17].

Fig.2 indicates the process of testing with *APCEMSI*. $Path_{1\text{-}6}$ are 6 executable paths of program P. Six squares in Fig.2 denote $DS(Path_i)$ ($i \in [1, 6]$), and we use $DOM(x_1, x_2, \ldots, x_n)$ ($x_1, x_2, \ldots, x_n \in [1, 6]$) to represent $\bigcup_{j=x_1,x_2,\ldots,x_n} DS(Path_j)$. Three binary MRs, $mr_{1,2,3}$ are constructed for P. As (a) illustrates, $D_{MT}(P) = DOM(1\text{-}6)$, $D_{app}(mr_1) = DOM(1,2,4,5)$, $D_{app}(mr_2) = DOM(2,3,5,6)$, and $D_{app}(mr_3) = DOM(1,2,3)$; ($b$) shows that after P has been tested with OI I_0, untested domains of $mr_{1,2,3}$ are $DOM(1,4)$, $DOM(5,6)$, and $DOM(3)$, that is, the untested paths of these 3 relations are $Path_{1,4}$, $Path_{5,6}$, and $Path_3$ respectively, and $\bigcup_{i=1}^{3} UTArea_i = DOM(1,3,4, 5,6)$. Besides, I_0's FUIs, I_1, I_2, and I_3 are chosen for the following testing because they are all included in $\bigcup_{i=1}^{3} UTArea_i$, but their FUIs, I_{31}, I_{12}, and I_{23} are not; As we can see in (c), when no FUIs could be selected and $\bigcup_{i=1}^{3} UTArea_i$ is not empty, new OIs should be randomly selected from $\bigcup_{i=1}^{3} UTArea_i$; Finally, when $\bigcup_{i=1}^{3} UTArea_i$ is empty, testing is over, as (d) displays.

Complexity of *APCEMSI* is $O(m*n + n*p)$, where m is the number of MRs, n the number of executable paths which are tested by MT, and p the maximum node number of all paths(each executed sentence should be checked during symbolic execution). In most situations, FUIs are not used later, and each executable path is only executed symbolically once, so the complexity of this algorithm is quadratic. If the test suite produced by *APCEMSI* is TC, then $|TC| \leq m*n$.

```
double TriangleSquare (int a, int b, int c) {
1   int match = 0;
P1 if (a == b)
2       match = match + 1;
P2 if (a == c)
3       match = match + 2;
P3 if (b == c)
4       match = match + 3;
P4 if (match == 0)   /* if a, b and c are not equals to each other*/
P5      if (a + b <= c) {
5           System.out.println("Not a triangle");
6           return 0.0;
P6      } else if (b + c <= a) {
7           System.out.println ("Not a triangle");
8           return 0.0;
P7      } else if (a + c <= b) {
9           System.out.println ("Not a triangle");
10          return 0.0;
        } else {
11          double p = (a + b + c)/2.0;
12          System.out.println ("Scalene");
13          return sqrt (p*(p-a)*(p-b)*(p-c)); /* compute square */
        }
P8 else if (match == 1)                /* if (a = b ≠ c) */
P9      if (a + b <= c) {
14          System.out.println ("Not a triangle");
15          return 0.0;
        } else {
16      double h = sqrt (pow (a, 2) - pow(c/2.0, 2));
17      System.out.println ("Isosceles");
18      return (c*h)/2.0;              /* compute square */
    }
P10 else if (match == 2)               /* if (a = c ≠ b) */
P11     if (a + c <= b) {
19          System.out.println ("Not a triangle");
20          return 0.0;
        } else {
21          double h = sqrt (pow (a, 2) - pow (b/2.0, 2));
22          System.out.println ("Isosceles");
23          return (b*h)/2.0;          /* compute square */
        }
P12 else if (match == 3)               /* if (b = c ≠ a) */
P13     if (b + c <= a) {
24          System.out.println ("Not a triangle.");
25          return 0.0;
        } else {
26          double h = sqrt (pow (b, 2) - pow (a/2.0, 2));
27          System.out.println ("Isosceles");
28          return (a*h)/2.0;          /* compute square */
        }
    else {                             /* if (a = b= c) */
29      System.out.println ("Equilateral");
30      return (sqrt (3.0)*a*a)/4.0;   /* compute square */
    }
}
```

Fig.3. Code of *TriSquare*

Tab.1. Binary MRs for *TriSquare*

MR	r_b	r_{bf}	$D_{bl}(mr_i)$
mr_1	$(a', b', c') = (b, a, c)$	$TriSquare\ (a', b', c') = TriSquare\ (a, b, c)$	$\{(a, b, c) \mid a = b\ \&\&\ a + b > c\}$
mr_2	$(a', b', c') = (a, c, b)$	$TriSquare\ (a', b', c') = TriSquare\ (a, b, c)$	$\{(a, b, c) \mid b = c\ \&\&\ b + c > a\}$
mr_3	$(a', b', c') = (c, b, a)$	$TriSquare\ (a', b', c') = TriSquare\ (a, b, c)$	$\{(a, b, c) \mid a = c\ \&\&\ a + c > b\}$
mr_4	$(a', b', c') = (2*a, 2*b, 2*c)$	$TriSquare\ (a', b', c') = 4*TriSquare(a, b, c)$	$\varnothing$
mr_5	$(a', b', c') = (\sqrt{2b^2 + 2c^2 - a^2}, b, c)$	$TriSquare\ (a', b', c') = TriSquare\ (a, b, c)$	$\{(a, b, c) \mid a^2 = b^2 + c^2\}$
mr_6	$(a', b', c') = (a, \sqrt{2a^2 + 2c^2 - b^2}, c)$	$TriSquare\ (a', b', c') = TriSquare\ (a, b, c)$	$\{(a, b, c) \mid b^2 = a^2 + c^2\}$
mr_7	$(a', b', c') = (a, b, \sqrt{2a^2 + 2b^2 - c^2})$	$TriSquare\ (a', b', c') = TriSquare\ (a, b, c)$	$\{(a, b, c) \mid c^2 = a^2 + b^2\}$

* $D_R(mr_i) = D_{MT}(TriSquare) = \{(a, b, c) \mid a + b > c\ \&\&\ b + c > a\ \&\&\ a + c > b\}$, $i \in [1, 7]$. Let $DMT = D_{MT}(TriSquare)$.

3.3. Case Study

Fig.3 illustrates the code of program *TriSquare*. This program first decides whether 3 positive real numbers, *a*, *b* and *c*, could construct a triangle. If so, type and square of this triangle are calculated.

We construct 7 binary MRs for *TriSquare*, details are shown in Tab.1. Among them, mr_5 is constructed based on parallelogram's characteristics. In Fig.4, it is obviously that $OD = OB = b$, so the squares of triangle *ODC* and *OBC* are equivalent to each other, and, that is, $TriSquare(a, b, c) = TriSquare(k, b, c)$. Because $|BD|^2 + |AC|^2 = |AB|^2 + |BC|^2 + |CD|^2 + |DA|^2$, $k = \sqrt{2b^2 + 2c^2 - a^2}$. $mr_{6,7}$ are formed in the same way.

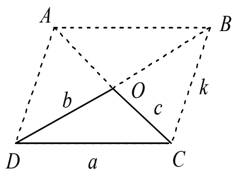


Fig.4. Principle of mr_5

Fig.5 illustrates the use of *APCEMSI* on *TriSquare*. Each input is a triple; Number on the right side of an input is the tag of path that it executes. Tab.2 gives the relevant relationships between tags and path[1]; Number on a line which connects OIs and FUIs is MR tag, for example, 1 indicates mr_1; Broken line denotes that the FUI above it executes the same path as its OI, so it should not be used later; Real line denotes that the FUI above it is not in $\bigcup_{i=1}^{7} UTArea_i$, so it isn't used later either; Black triangle denotes that OI isn't in the untested domain of associated MR, so FUI is not generated at all; Rectangle denotes that the input in it has been used as an OI before. Finally, 26 test cases are produced and the program runs 28 times.

Tab.2. Paths and their Tags of *TriSquare*

Tag	Executable Paths	Sub-Domain of Paths
1	1F-2F-3F-4T-5F-6F-7F	$\{(a, b, c) \mid a \neq b \neq c \neq a\} \cap DMT$
2	1T-2F-3F-4F-8T-9F	$\{(a, b, c) \mid a = b \neq c\} \cap DMT$
3	1F-2T-3F-4F-8F-10T-11F	$\{(a, b, c) \mid a = c \neq b\} \cap DMT$
4	1F-2T-3F-4F-8F-10F-12T-13F	$\{(a, b, c) \mid b = c \neq a\} \cap DMT$
5	1F-2T-3F-4F-8F-10F-12F	$\{(a, b, c) \mid a = b = c > 0\}$

[1] Number in the 2nd column denotes the path condition tag in Fig.3; "T" indicates that path condition is true, while "F" false. For example, "1F" represents that "P1"is false.

4. Experimental Results

Mutation analysis is a powerful technique to assess the quality of a test suite [13]. As MRs' performance is embodied by MT test suite, we use mutation analysis here to estimate test suites and MRs. Four mutants are imported into *TriSquare* based on two types of mutant operators: AOR(Arithmetic Operator Replacement) and DSA(Data Statement Alterations) [16]:

Mutant1: Exchange sentence 2 and 3;
Mutant2: Replace sentence 11 with "*p*= (*a*+*b*+*c*)*2";
Mutant3: Replace "/2" in 18, 23 and 28 with "*2";
Mutant4: Replace sentence 30 with "*return sqrt*(3)**a***a*/2".

4.1. Measurements for Test Suite and MR

Mutation Score(*MS*) Measure the quality of a test suite *TC*, which is the percentage of mutants detected:

$$MS(TC) = M_k \big/ (M_t - M_q) \tag{3}$$

where M_k is the number of mutants detected by *TC*, M_t the total number of mutants, and M_q the number of equivalent mutants that cannot be detected by any set of test data [13].

Faults on problem Path Detection ratio(*FPD*) Measure the quality of a test suite *TC* for program *mp* including a mutant, which is the percentage of test cases failed:

$$FPD(mp, TC) = N_f \big/ N_{fp} \tag{4}$$

where N_f is the number of test cases that could detect the mutant in *mp*, and N_{fp} the number of test cases that executes paths including mutants.

MR Detection Performance (*MDP*) Measure the mutation detection performance of MR *mr* in terms of test suite *TC*, which is the number of mutants that the test cases using *mr* could find:

$$MDP(mr, TC) = \sum\nolimits_{i=1}^{n} FindMut_i \tag{5}$$

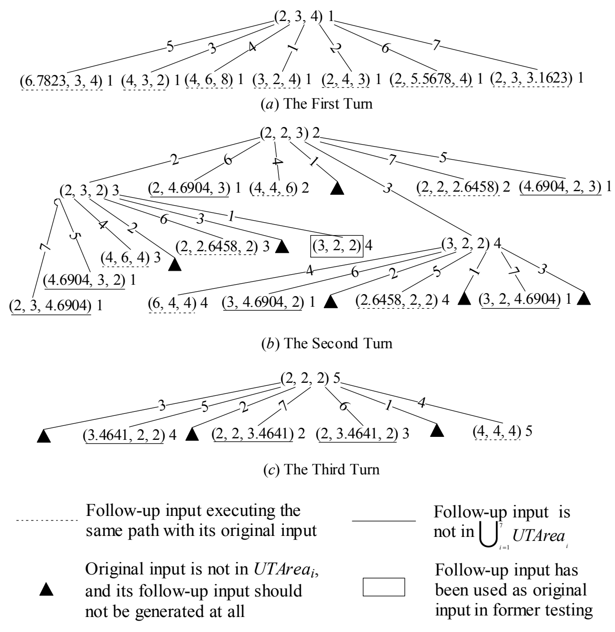

(a) The First Turn
(b) The Second Turn
(c) The Third Turn
Follow-up input executing the same path with its original input
Follow-up input is not in ⋃ UTArea_i
Original input is not in UTArea_i, and its follow-up input should not be generated at all
Follow-up input has been used as original input in former testing

Fig.5. Testing of *TriSquare* with *APCEMSI*

FindMut$_i$ is 1 if at least one test case that uses *mr* could detect the *i*th mutant, otherwise is 0, and *n* is the number of mutants.

MR Detection ratio for each Mutant (*MDM*) Measure the detection performance of MR *mr* for each mutant in terms of test suite *TC*, which is the percentage of failed test cases using *mr* :

$$MDM(mr, mp, TC) = Nmr_f \big/ Nmr_{mp} \qquad (6)$$

where *mp* is a mutant program, *Nmr$_f$* the number of test cases which use *mr* and could detect the mutant in *mp*, and *Nmr$_{mp}$* the number of test cases that use *mr* and executes paths including mutants.

4.2. Conclusion for Experiment

We test each mutant imported program 100 times (50 with *mr*$_{1\text{-}7}$, and 50 with *mr*$_{1\text{-}4}$). In this section, *TrSqi* represents the program into which Mutant*i* is imported ($i \in [1, 4]$). *FPDi* and *MDMi* denote *FPD* and *MDM* values for *TrSqi*.

Testing results are shown in Tab.3, and there are only 6 result patterns, which could be divided into 4 groups. The first 3 groups are based on *mr*$_{1\text{-}7}$, while the last on *mr*$_{1\text{-}4}$. Group1 contains *Pattern*1 only. If the first randomly selected OI executes *Path*$_1$, and the second *Path*$_{2,3,4}$, testing result will belong to *Pattern*1; Group2 includes *Pattern*2 and 3. If the first randomly chosen OI executes *Path*$_{2,3,4}$, *Pattern*2 or 3 will emerge; Group3 contains *Pattern*4 and 5. If the inputs executing *Path*$_{2,3,4}$ are all deduced from a randomly chosen OI which runs *Path*$_5$, the testing result will fall into *Pattern*4; Group4 only includes *Pattern*6. Base on the above data, following conclusions can be drawn:

1) Using *APCEMSI*, both the test cases generated and program executing times are much fewer than those of traditional MT. Tab.3 shows that the amount of test cases using *mr*$_{1\text{-}7}$ is 20~26, and the program executing times is 15~28, while those of the traditional method are 35 and 40 (5 paths and 7 MRs). In most situations (47 times in 50 here), all mutants could be detected.

2) MRs selection and OIs directly impact the testing results. In Tab.3, *MS* values of *Pattern*1-5 are no less than 75%, while that of *Pattern*6 is only 25%, so MR selection affects the testing result. The differences between *Pattern*1,2,3,4,5 indicate that random OI choice also makes a great effort on testing.

3) MR based on complex theory is more useful. We find that in Tab.4 the *MDP* values of *mr*$_{5\text{-}7}$ are larger than *mr*$_{1\text{-}4}$. This is mostly because the constructing theory of *mr*$_{5\text{-}7}$ is more complex than linear theory that *mr*$_{1\text{-}4}$ adopt.

4) MR with larger applying domain is more useful. Because *DS*(*Path*$_5$) is not in the applying domains of *mr*$_{1,2,3}$, so *MDM*4 for these 3 MRs are all 0.00%, and *Mutant*4 could not be detected by them.

5) When we test a program whose input is multi-tuple, MR making more corresponding tuples different is more useful. Tab.4 indicates that the *MDM* values of *mr*$_{6,7}$ on *TrSq*1 are only 33.33% in *Pattern*1-3. The input of *TriSquare* is a triple. Compare to OI, FUIs from *mr*$_{5,6,7}$ only make one tuple changed. When we test *TrSq*1 with *mr*$_6$ or *mr*$_7$, sometimes two executions of one test case both refer to unchanged tuples. In this situation, *Mutant*1 could not be detected.

5. Conclusion

In this article, we define a MT criterion, *APCEM* based on program path analysis technique, and present a new iterative MT algorithm, *APCEMSI* to generate *APCEM* satisfied test suites. The efficiency of this

Tab.3. Testing Results of *TriSquare* with *APCEMSI*

MR	Pattern	Amount of Test Cases	Executing Times	*FPD*1	*FPD*2	*FPD*3	*FPD*4	*MS*	Times
mr$_{1\text{-}7}$	*Pattern*1	26	28	46.15%	69.23%	50.00%	75.00%	**100%**	**39**
	*Pattern*2	24	22	46.15%	63.64%	50.00%	75.00%	**100%**	**5**
	*Pattern*3	23	21	46.15%	60.00%	50.00%	75.00%	**100%**	**3**
	*Pattern*4	23	22	0.00%	69.23%	60.00%	75.00%	**75%**	**2**
	*Pattern*5	20	15	0.00%	60.00%	60.00%	75.00%	**75%**	**1**
mr$_{1\text{-}4}$	*Pattern*6	11	13	40.00%	0.00%	0.00%	0.00%	**25%**	**50**

Tab.4. *MDP* and *MDM* of MRs when testing *TriSquare*

Pattern / Mutant / MRs	*Pattern*1-3					*Pattern*4-5				
	*MDM*1	*MDM*2	*MDM*3	*MDM*4	*MDP*	*MDM*1	*MDM*2	*MDM*3	*MDM*4	*MDP*
mr_1	100.00%	0.00%	0.00%	0.00%	1	0.00%	0.00%	0.00%	0.00%	0
mr_2	0.00%	0.00%	0.00%	0.00%	0	0.00%	0.00%	0.00%	0.00%	0
mr_3	100.00%	0.00%	0.00%	0.00%	1	0.00%	0.00%	0.00%	0.00%	0
mr_4	0.00%	0.00%	0.00%	0.00%	0	0.00%	0.00%	0.00%	0.00%	0
mr_5	100.00%	100.00%	75.00%	100.00%	4	0.00%	100.00%	100.00%	100.00%	3
mr_6	33.33%	100.00%	75.00%	100.00%	4	0.00%	100.00%	100.00%	100.00%	3
mr_7	33.33%	100.00%	75.00%	100.00%	4	0.00%	100.00%	100.00%	100.00%	3

algorithm is proved by experiment and some conclusions drawn from it are provided.

As future work, we would like to: (1) Improve *APCEMSI* to agree with large programs and software including complex loops and predications. (2) Consider MT for OO program. Some researchers produced equivalent pairs of method sequences [18] for class-level testing based on the idea of MT, but this technique is intricate. There are some similarities between program path and state-transition path, so one of our next goals is testing OO program with state-transition graph and MT.

6. References

[1] E.J. Weyuker. “On testing non-testable programs.” *The Computer Journal*, 1982. 25(4): p. 465~470.

[2] T.Y. Chen, S.C. Cheung, and S.M. Yiu. “Metamorphic testing: a new approach for generating next test cases.” in *Technical Report HKUST-CS98-01,* 1998.

[3] T.Y. Chen, J. Feng, and T.H. Tse. “Metamorphic testing of programs on partial differential equations: a case study.” in *Proceedings of the 26th Annual International Computer Software and Applications Conference*, 2002.

[4] A. Gotlieb, and B. Botella. “Automated metamorphic testing.” in *Proceedings of the 27th Annual International Computer Software and Applications Conference,* 2003.

[5] Z.Q. Zhou, D.H. Huang, T.H. Tse, Z.Y. Yang, H.T. Huang, and T.Y. Chen. “Metamorphic Testing and Its Applications.” in *Proceedings of the 8th International Symposium on Future Software Technology,* 2004.

[6] T.H. Tse, S.S. Yau, W.K. Chan, L. Heng, and T.Y. Chen. “Testing context-sensitive middleware-based software applications.” in *Proceedings of the 28th Annual International Computer Software and Applications Conference,* 2004.

[7] W.K. Chan, T.Y. Chen, L. Heng, T.H. Tse, and S.S. Yau. “A metamorphic approach to integration testing of context-sensitive middleware-based applications.” in *Proceedings of the 5th Annual International Conference on Quality Software,* 2005.

[8] W.K. Chan, S.C. Cheung, and K.P.H. Leung. “Towards a Metamorphic Testing Methodology for Service-Oriented Software Applications.” in *Proceedings of the 5th International Conference on Quality Software,* 2005.

[9] W.K. Chan, S.C. Cheung, and Karl R.P.H. Leung. “A metamorphic testing approach for online testing of service-oriented software applications.” in *a Special Issue on Services Engineering of International Journal of Web Services Research*, 2007(2): p. 60~80

[10] T.Y. Chen, D.H. Huang, T.H. Tse, and Z.Q. Zhou. “Case Studies on the Selection of Useful Relations in Metamorphic Testing.” in *Proceedings of the 4th Ibero-American Symposium on Software Engineering and Knowledge Engineering,* 2004.

[11] T.Y. Chen, T.H. Tse, and Z.Q. Zhou. “Semi-proving: an integrated method based on global symbolic evaluation and metamorphic testing.” *ACM SIGSOFT Software Engineering Notes,* 2002. 27(4): p. 191~195.

[12] T.Y. Chen, T.H. Tse, and Z.Q. Zhou. “Fault-based testing without the need of oracles.” *Information and Software Technology*, 2003. 45(1): p. 1~9.

[13] P. Wu, X.C. Shi, J.J. Tang, H.M. Lin, and T.Y. Chen. “Metamorphic Testing and Special Case Testing: A Case Study.” *Journal of Software,* 2005. 16(7): p. 1210~1220.

[14] P. Wu. “Iterative Metamorphic Testing.” in *Proceedings of the 29th Annual International Computer Software and Applications Conference,* 2005.

[15] T.A. Budd. “Mutation analysis: Ideas, examples, problems and prospects.” in *Proceedings of the Summer School on Computer Program Testing.*, 1981: p.129~148.

[16] A.J. Offutt, A. Lee, G. Rothermel, R.H. Untch, and C. Zapf. “An Experimental Determination of Sufficient Mutant Operators.” *ACM Transactions on Software Engineering and Methodology,* 1996. 5(2): p. 99~118.

[17] J.C. King, and J. Thomas. “Symbolic execution and program testing.” *Communications of the ACM,* 1976. 19(7): p. 385~394.

[18] H.Y. Chen, T.H. Tse, and T.Y. Chen. “TACCLE: a methodology for object-oriented software testing at the class and cluster levels.” *ACM Transactions on Software Engineering and Methodology,* 2001. 10(1): p. 56~109.