A Method of Metamorphic Relations Constructing for Object-oriented Software Testing

首先根据面向对象的软件程序构造蜕变关系标准

然后构造蜕变关系

最后根据蜕变关系标准提高蜕变关系

Xinglong Zhang, Lei Yu, Xuemei Hou

State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450001, China

paper92@163.com

***Abstract*—To solve the Oracle problem of methods sequence in object-oriented software testing, a method of metamorphic relations constructing for object-oriented software testing based on algebraic specification was proposed. Firstly, metamorphic relations constructing criteria for object-oriented testing was defined based on the characteristics of object-oriented software program. Then metamorphic relations were constructed based on GFT algorithm (Generating a Finite Number of Test Case). Finally the metamorphic relations were improved according to these criteria. The improved method was verified through constructing IntStrack class and SavAcc class metamorphic relations. The experiment results show that the metamorphic relations redundancy is decreased significantly. So the new method has a low metamorphic relations redundancy and improves the efficiency of software testing.**

Keywords—Object oriented; metamorphic relation; algebraic specification

冗余

I. INTRODUCTION

Software testing is an important part in the process of software development, and the traditional testing technology is restricted by the Oracle problem[1], that in some situations, it is impossible or practically too difficult to decide whether the program outputs on the test cases are correct. So it is difficult to determine whether the program execution results are identical to the expected results. A metamorphic testing methods has been proposed by Chen et al[2]. It is an automated approach to alleviating the oracle program which can check correctness of the program by compare multiple executions.

缓解

It is obvious that metamorphic relations are the core part of the metamorphic testing, as they are not only used in test case generation, but also provide a mechanism for test result verification, and it is related to the efficiency of the test. There will be some problems, one of them is lacking necessary metamorphic relations constructing criteria[3-4]. Its negative influences lies in two aspects: 1) generate a large number of test cases that have similar function. 2) testing is inadequate because of low functional coverage.

Some construction methods of metamorphic relation have been presented. Upulee Kanewala[5] proposed a metamorphic relation constructing method using machine leaning techniques. Huai Liu[6] proposed composite metamorphic relation constructing method based on the already identified metamorphic relations. MURRHY[7] proposed seven general metamorphic relation constructing criteria, but it's only suitable for numerical procedure. Zhan-we Hui[8] construct a decomposition model for metamorphic relation with the formal model and provide three sub-relations for decrease complexity of metamorphic relation. Although Wang Rong[9] summed up a series of the basic guidelines for metamorphic relation on the basis of previous work, he did not give a specific construction method. When we test object-oriented programs all these methods mentioned above are not possible. So it is necessary to propose a object-oriented metamorphic relation construction method.

In object-oriented testing, when testing method sequence of the class, test case consists of a sequence method and the corresponding parameter values. Jinan University Professor Chen Huo-yen[10-12] proposed GFT algorithm to generate a limited number of base pairs as a test case based on algebraic specifications. In a given algebraic specification, replace class variables with normal form that its lengths less than positive integer k to generate a finite equivalence base pairs as the test case.

GTF算法的缺点

By examining the GFT algorithm, the following shortcomings are found: (1) the value K is difficult to determine. although the K value is determined by white-box information, metamorphic relation also contains a lot of redundancy. (2) Replace variables in axiom with normal form may generate wrong metamorphic relation because of not considering semantic errors, such as cross-border problems. (3) GFT only relies on axiom. Metamorphic relation constructed based on GFT is inadequate, because it's not

语义的

公理

978-1-5090-2239-7/16/$31.00 copyright 2016 IEEE
SNPD 2016, May 30-June 1, 2016, Shanghai, China

verified the object state after method calling.

As for the drawback of the GFT, we makes the improvements for GFT. First, using white-box information to select normal form can reduce the redundancy of metamorphic relations; Secondly, proposed a method of metamorphic relations constructing for object-oriented software. This method ensure that the object state of all methods can be tested, and increase the adequacy of metamorphic relations. At last, when conducting a test, the metamorphic relation will be generated into a test case, thus the returned value of the object and the object's current state can be employed to judge the equivalence property of execution of results, which to a certain extent simplifies the Oracle problem of object-oriented class-level testing.

II. CONCEPTS

Definition 1(metamorphic relation, *MR*)[13] Suppose that program *P* computes function *f*, which is also referred to as the specification that *P* must accord with. Let $x_1, x_2 \ldots x_n$ $(n > 1)$ be n different inputs of function *f*, if their satisfaction of relation *r* implies that their corresponding outputs $f(x_1)$, $f(x_2), \cdots, f(x_n)$ satisfy relation r_f that is

$$r(x_1, x_2, \cdots, x_n) \Rightarrow r_f(f(x_1), f(x_2), \cdots, f(x_n))$$

Then (r, r_f) is a *MR* of *f*. Because *P* is an implementation of *f*, if *f* is correct, it should also satisfy this relation, that is

$$r(I_1, I_2, \cdots, I_n) \Rightarrow r_f(P(I_1), P(I_2), \cdots, P(I_n))$$

Where $I_1, I_2, \cdots, I_n$ are inputs of P that associate with x_1 $x_1, x_2, \cdots, x_n$, and $P(I_1)$, $P(I_2), \cdots$, $P(I_n)$ are their outputs. So (r, r_f) is also an *MR* of *P*. When we test *P* with (r, r_f), the

推导

test cases originally given are original test cases, others which are deduced from relation *r* are follow-up test cases.

Definition 2 (algebraic specification)[10] The algebraic specification of class *C* can be denoted by $C = (S, F, V)$. *S* is a subset of class *C*. *F* is a collection of method of class *C*. Each method *f* can be expressed as $\_.f() : S \times S \times \ldots \times S \rightarrow S$, the left-hand side is input parameters, the right-hand side is output parameters. The algebraic specification consists of equational axioms that describe the behavioral properties of the operations expressed by *V*. The following example shows an algebraic specification of the class IntStack of integer stacks:

module *INTSTACK* is
classes *Int Bool IntStack*
inheriting *INT*
operations
new: -->IntStack
_.empty: IntStack -->Bool
.push(): IntStack Int -->IntStack
_.pop: IntStack -->IntStack
_.top: IntStack-->Int U{NIL}
variables
S: IntStack
N: Int
axioms
a_1: *new.empty = true*
a_2: *S.push(N).empty = false*
a_3: *new.pop = new*
a_4: *S.push(N).pop = S*
a_5: *S.top = NIL if S.empty*
a_6: *S.push(N).top = N*

In a given class *C*, operations or methods generating new objects of *C* are called creators. Operations or methods changing the values of attributes of object in *C* are called constructors or transformers. Operations or methods that only output the values of attributes of objects in *C* are called observers.

Definition 3 (term) A syntactically valid sequence of operations in a given algebraic specification is called a term. For example, *new.push.pop* is a term, but *new.empty.pop* is not in the class of integer stacks above.

Definition 4 (normal form) In algebraic specification, a term is in mormal form if and only if it cannot be further transformed by any axiom in the specification. For example, *new.push*(1).*push*(2) is in normal form, but *new.push*(1). *push*(2).*pop* is not.

Definition 5 (method sequence metamorphic relation) In testing of object-oriented software, let the equational sequence, $\_.p_i, \_\,q_s \in F, (i = 1, 2 \ldots m, s = 1, 2 \ldots n)$, $p_1().p_2()$ $\ldots \_\,p_m() = \_.q_1().q_2() \cdots q_n()$ be method sequence of *MR*.

Definition 6 (single-value metamorphic relation) In testing of object-oriented software, let $\_.p_1().p_2() \ldots \_.p_m()$ $= A$ $\_.p_i \in F$ $(i = 1, 2, \cdots, m)$, be single-value *MR*, if A is a value rather than a sequence of operations.

III. CRITERION ANALYSIS

A. criterions of constructing metamorphic relations

Based on the characteristics of the object-oriented algebra specification, a large number of experiments and results analysis, criterions of constructing *MRs* were proposed as follows:

Criterion 1 Given a algebraic specification $C = (S,F,V)$, the following formula can be deduced: $\forall \_.f() \in F$, $\exists\{\_.p_1().p_2()\cdots\_.p_m().f()=\_.q_1().q_2()\cdots q_n()\}\vee\{\_.p_1().p_2()\cdots\_.p_m().f()=A\}$. The constructed *MRs* should make all the methods contained in this class appear once in the last position of the method sequence on the either side of the *MR* to the greatest extent. This criterion is to make sure that the returned value of every called method and the current object state can be tested. Please pay attention that if the both sides of the *MR* are method sequences, then try not to end the method sequences on the two sides using the same method, for which will make it hard to expose relevant mistakes of the method.

Criterion 2 If a method $\_.f() \in F$ is not a observer and only appear at the end of the *MR*, construct new *MR* in order to ensure that the current states after all methods except observers called can be tested. For method sequence *MR*, this method can appear in everywhere.

Criterion 3 Construct *MR* should combine white-box, for instance boundary conditions (the length of array). The more white-box informations we consider, the more fault detection MR we have.

Criterion 4 The *MR* set which the length of *MRs* are more longer and the number of them are shorter is more effective. Construct compositional *MRs* is a effective way to reducing the number of *MRs*. The fault detection of Composite *MR* is better than its corresponding component *MRs*.

Criterion 5 The more difference between two sides of *MR*, the more likely their outputs are not equal. For the object-oriented *MR*, the concept of "difference between two sides of *MR*" involve difference in method sequence length and number of method type.

B. Construction procedures of metamorphic relation

Suppose that in algebraic specification of class *C*, each axiom a_i performs the following steps to construct *MR* sets.

1) a_i is a *MR*, if variables of class *C* is not included in axiom a_i.

2) If axiom a_i include variables of class *C*, according to guideline 3, take $RP=\{r_j, j=1,2,\cdots l\}$ as replacement sequences, l is the number of replacement sequences.

3) Construct new equation a_{ij} by replacing variables of class *C* in axiom a_i with r_j. a_{ij} is a *MR* deducted from axiom a_i.

4) Do semantic checking of *M*R constructed by previous step. Revise *MR* that doesn't conform to semantic.

5) Construct composite *MR*. Construct $\_.f_1().\cdots.p_1().p_2().\cdots.p_n().\cdots.f_k() = \_.r_1().r_2().\cdots$ where $\exists\{\_.p_1().p_2().\cdots.p_m() = \_.q_1().q2()\cdots q_n()\}\wedge\{\_.f_1().\cdots q_1()\cdots q_n()\cdots.f_k() = \_.r_1().r_2()\cdots r_l()\}$.

6) According to criterion 1, traverse all the transformation relations, find the method that does not appear at the end of any method sequence, construct a new *MR*.

7) According to the criterion 2, new *MR* is constructed with the methods that only appear at the end of the sequence.

Construct *MR* of IntStack according to the steps below:

1) Replace variable *S* of class *C*. we generally use constructor to replace. According to white-box information, the length of array in stack is 100, so we can choose method sequence of one hundred push as the replacement. The $RP = \{r_1, r_2\}$ is as follows:

r_1: *new*

r_2: $new.push(N_1).push(N_2)\cdots push(N_{100})$

Replace object variable in axiom with r_i, the constructed *MRs* are as follows.

a_1: $new.empty=true$

a_{21}: $new.push(N).empty=false$

a_{22}: $new.push(N_1).push(N_2)...push(N_{100}).push(N).empty=false$

a_3: $new.pop=new$,

a_{41}: $new.push(N).pop=new$,

a_{42}: $new.push(N_1).push(N_2)\cdots push(N_{100}).push(N).pop= new.push(N_1).\ push(N_{100})$

a_{51}: $new.top = NIL$

a_{61}: $new.push(N).top=N$

a_{62}: $new.push(N_1)\cdots push(N_{100})push(N).top=N$

2) Carry out a semantic check on the above *MRs* and correct the one at variance with semanteme. For example, in terms of the stack operation, the maximum array length is 100, only 100 integers can be pushed on to the stacked. When push

the 101st integer, that integer fails to be pushed. What is on the stack are still the pushed 100 integers. Hence, a_{42} should be modified as $new.push(N_1).push(N_2)\cdots push(N_{100}).push(N).pop= new.push(N_1). push(N_{99})$.

3) According to criterion 5, a_{22}, a_{62} are more difference than a_{21}, a_{61}. Choose the first two *MRs*. Based on criterion 4, compose three MRs a_3、a_{41}、a_{42} into $a_{3\text{-}41\text{-}42}$: $new.pop.push(N) pop.push(N_1). push(N_2) \cdots push(N_{100}).push(N).pop = new.push(N_1) \cdots push(N_{99})$ and $a_{3\text{-}41\text{-}42}{}'$: $new.pop.push(N).pop.push(N_1). push(N_2) \cdots push(N_{100}).push(N).pop = new.pop.push(N).pop. push(N_1)\cdots push(N_{99})$. According to criterion 5, $a_{3\text{-}41\text{-}42}$ is more difference than $a_{3\text{-}41\text{-}42}{}'$, put $a_{3\text{-}41\text{-}42}$ to *MR* set. *MR* set is as follows:

a_1: $new.empty = true$

a_{22} : $new.push(N_1).push(N_2)\cdots push(N_{100}).empty = false$

$a_{3\text{-}41\text{-}42}$: $new.pop.push(N).pop.push(N_1).push(N_2) \cdots push (N_{100}) .push(N).pop= new.push(N_1)\cdots push(N_{99})$

a_{51}: $new.top = NIL$

a_{62}: $new.push(N_1).push(N_2)\cdots push(N_{100}).push(N).top = N$

4) Based on criterion1, the method new is not appear at the end of sequence method in any *MR*. Construct a *MR* that one side of it ends with new. Add a_{41}: $new.push(N).pop=new$ in *MR* set.

The finally *MR* set is as follows:

a_1: $new.empty = true$

a_{22} : $new.push(N_1).push(N_2)\cdots push(N_{100}).empty = false$

a_{41}: $new.push(N).pop=new$

$a_{3\text{-}41\text{-}42}$: $new.pop.push(N).pop.push(N_1).push(N_2).push(N_3) \cdots push(N_{100}) .push(N).pop= new.push(N_1)\cdots push(N_{99})$

a_{51}: $new.top = NIL$

a_{62}: $new.push(N_1).push(N_2)\cdots push(N_{100}).push(N).top = N$

A. Analysis on the constructed metamorphic relation set is as follows:

1) a_1 can verify the object state after new calling and the returned value after empty calling, in addition, it can also verify the mistakes of stack length correction in the last call of empty method, and stack array bound mistakes caused by the correction on stack length mistakes in the new method.

2) a_{22} can verify the object state after new and push and the returned value after empty calling, in addition, it can also verify the mistakes of stack length correction in the last call of empty method, and stack array bound mistakes caused by the correction on stack length mistakes in the new and push methods.

3) a_{41} can verify the object state after new, push, pop calling. It can verify the returned value and object state caused by the mistakes of stack length correction.

4) $a_{3\text{-}41\text{-}42}$ can verify object state after new, pop and push calling. It can verify stack length of the two sides differ from each other, and stack array bound mistakes that caused by mistakes of stack length modification.

5) a_{51} can verify the object state after new, and the returned value after top calling. It can verify stack array bound mistakes that caused by mistakes of stack length modification. In addition, it can also verify the mistakes that returned value of top is not NIL.

6) a_{62} can verify the object state after new, push calling, and the returned value after top calling. It can verify stack array bound mistakes caused by the modification on stack length mistakes in the new, push and top methods. It can also verify the mistakes that returned value of top is not *N*.

In summary, the *MRs* constructed in this method can verify the consistency of executing results on both sides in method sequence; if both sides of the method sequence can't implement a certain defect point, the defect point can't be identified. Then construct a new *MR*. Generally speaking, as the variety for defects is in a wide range, so we usually set up a threshold value, if the failure-detecting rate reached this threshold, the test can be stopped.

IV. EXPRIMENT

B. Mutantion

In order to verify the effectiveness and efficiency of the *MRs*, we use mutation testing to analyze. Mutation testing (or Mutation analysis or Program mutation) is used to design new software tests and evaluate the quality of existing software tests. Mutation testing involves programs modified in small ways, each mutated version is called a mutant. Traditional mutation testing execute test cases in origin program and mutations respectively and the results are compared. If the results of mutations differ from origin program, it means the mutations are killed. It is usually appear three results when using mutation testing to verify the mutation detection capability of *MRs*[15]. (1) The test results do not satisfy *MR*. It shows that the *MR* killed mutations, namely the *MR* can detect these mutations.(2) The test results satisfy *MR*. It shows that the *MR* did not kill mutants, namely the *MR* cannot detect such mutations. (3) When a mutation terminates due to the program execution fails, that means the *MR* killed mutation.

We use mujava to generate 122 mutations as shown in Table 1.

C. measurement criterion

The evaluation of *MRs* and test cases should be measured through multi-dimensional indicators. So the following three measurement criterions are proposed.

1) mutation score (*MS*)[16]

$$MS(T) = \frac{M_k}{M_t - M_q} \qquad (1)$$

Where M_k is the number of mutants detected by the test case, 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. M_s is a macro-criterion that it doesn't care which mutation is tested.

2) Metamorphic relations Failure-detecting Capability (*MRFS*)[16]:

$$MRFC(a_i) = \frac{M_{a_i}}{M_t - M_q} \qquad (2)$$

Where M_{ai} is the number of mutants detected by a_i, M_t is the total number of mutants, and M_q is the number of equivalent mutants that cannot be detected by any set of test data. This measurement criterion evaluate the failure- detecting capability of *MRs.*

Table 1 122mutations of IntStack

Mutation Operation	Operation Description	Number
AORB	Arithmetic Operators Replacement (binary arithmetic operator)	4
AORS	Arithmetic Operators Replacement (unary arithmetic operator)	3
AOIS	Arithmetic Operator Insertion (short-cut arithmetic operator)	40
COI	Conditional Operator Insertion	4
SDL	Statement Deletion	6
AODU	Arithmetic Operator Deletion(unary)	1
AOIU	Arithmetic Operator Insertion(unary arithmetic operator)	5
ROR	Relation Operation Replacement	27
LOI	Logical Operation Insert	14
VDL	Variable Deletion	8

3) Mutation Metamorphic Relations Failure-detecting Capability (*MMRFC*) [16]:

$$MMRFC(m) = \frac{N_m}{N} \qquad (3)$$

Where N is the number of *MRs*, N_m is the number of *MRs* that can kill mutation m.

D. Experimental results analysis

During the experimental execution, variables that in the generated *MR* shall be replaced with actual values. The selection of actual values can base on the equivalence class classifying criteria in black-box testing and path-coverage criterion in white-box testing. The replaced method sequence and corresponding parameter values constitute original test cases and follow-up test cases. Execute original e test cases

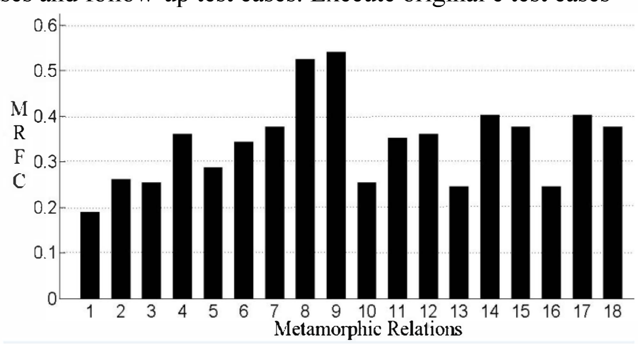

Figure 1 Error-detecting capacity of *MRs* in Paradigm

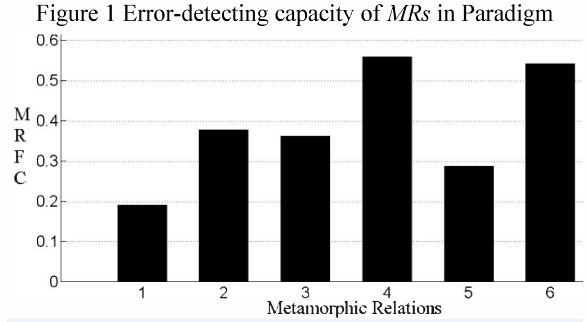

Figure 2 Error-detecting capacity of *MRs* in improved method

and follow-up test cases respectively, and then determine

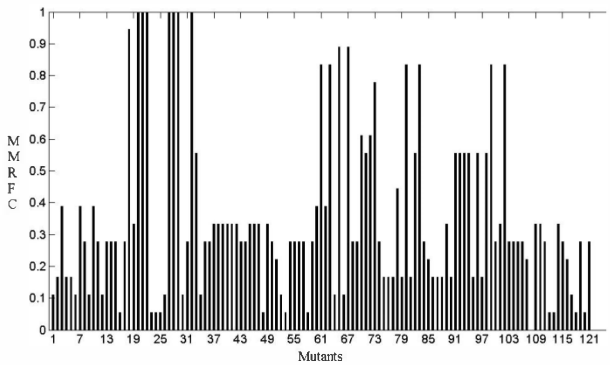

Figure 3 Mutation *MRs* Failure-detecting Capability

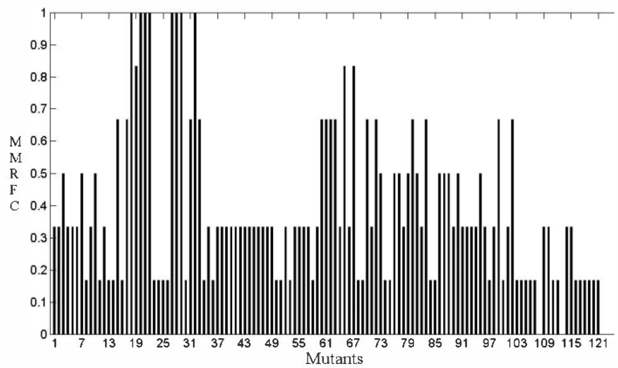

Figure 4 Mutation *MRs* Failure-detecting Capability

whether the executed results are equal. In the determination of the equivalence of the executing results, the following two conditions need to be considered: (1) if it is single-valued *MR*, we only need to determine whether the returned value in method sequence is equal to the specific value of the other side. (2) If it is the method sequence *MR*, we have to compare whether the returned value and object state equals to the other side.

Let's take the integer stack for an example, according to GFT algorithm, replace variables in axiom with the normal forms which length are 1, 2, 3 and 100 to construct eighteen *MRs*; and then construct six *MRs* according to the new algorithm. As the parameter values are merely values of pushed stack, so equivalence class division is not required; therefore, we can set $N_1 = 1$, $N_2 = 2$, ... $N_{101} = 101$, that is to obtain eighteen pairs and six pairs of test cases, and act them to the above 122 mutations. Set the threshold for mutation detection error rate to be 95%, after the test, calculate the measure index.

Mutation detection rate:

$$MS_{GFT}(T) = 118/122 = 96.7\%$$

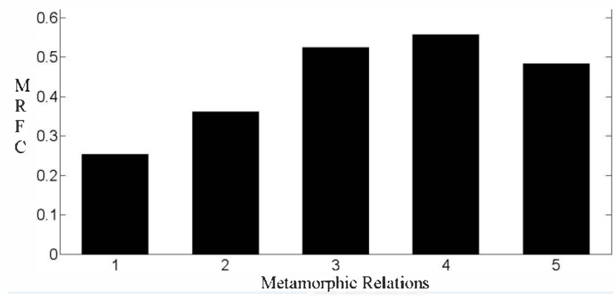

Figure 5 failure-detecting capability of composite *MR* and its corresponding component *MRs*

$$MS_{NEW}(T) = 116/122 = 95.6\%$$

Through mutation detection rate, it can be seen that the mutation-detecting capacities of the six *MRs* constructed under new guidelines and the eighteen *MRs* constructed by GFT algorithm have reached to a set threshold, while the *MR* redundancy has reduced by 66.7%, which verified the effectiveness of *MRs* constructed based on the above criteria.

2) mutants-detecting capacity *MRFC* of *MR* 122 variants.

As can been seen from Figure 1 and Figure 2, *MR* constructed by GFT contains large *MR* redundancy, the Failure-detecting Capability of most *MRs* constructed by improved method is better than MRs constructed by GFT. In the *MR* constructed by GFT, mutation detection rate in 2, 3 and 4 *MRs*, in 10, 11 and 12 MRs, in 13, 14 and 15 MRs, and in 16, 17 and 18 *MRs* are basically similar, as the difference of normal form replaced class variables with the length of 1, 2 and 3 are not insignificant, verifying that the *MR* redundancy constructed by GFT normal form is significant. The 7, 8 and 9 MRs in figure 1 and the 6 *MR* in figure 2 have higher mutation detection rate, it is because the three MRs have considered the white-box information array bounds issue and verified the Guideline 3.

As can been seen from Figure 2, the mutation-detecting capacity of the third *MR* $a_{3\text{-}41\text{-}42}$ is much higher than other *MRs*, as it contains all methods except observers in IntStack class, and it also verify the object state after new, push and pop calling, moreover, it has the ability to detect that both sides of stack length is inconsistent after the correction of new, pop and push methods, and the stack array bound mistakes caused by the correction on stack length mistakes. Compared to other *MRs*, it comprises the most complete methods, and takes the white box information, such as array bounds issue into account, verifies the guideline 4.

3) Metamorphic relations Failure-detecting Capability

MMRF

As can been seen from Figure 3 and Figure 4, in Figure 4 the mutation *MRs* failure-detecting capability of most mutation*s* is better than mutations in Figure3. In addition, mutation number 22、23、24、29、30、31、34, can be verified by all the constructed *MRs*. Because those mutants caused by object state mistakes or stack array bound mistakes after new calling. All *MRs* start with new, thus those mutations can be verified.

4) Comparative study of composite *MR* and its corresponding component *MR*

In Figure 5, the first three *MRs* are a_3、a_{41}、a_{42} respectively. The later two are composite *MRs* constructed by the first three. The fouth *MR* is $a_{3\text{-}41\text{-}42}$: $new.pop.push(N).pop.push(N_1)$ $push(N_2)...push(N_{100}).push(N).pop=new.push(N_1) \cdots push(N_{99})$ and the fifth *MR* is $a_{3\text{-}41\text{-}42}{}'$: $new.pop.push(N).pop.push(N_1)$ $push(N_2) \cdots push(N_{100}).push(N).pop = new.pop.push(N).pop.$ $push(N_1) \cdots push(N_{99})$. We find that the difference between two sides of $a_{3\text{-}41\text{-}42}$ is more larger than $a_{3\text{-}41\text{-}42}{}'$, and in Figure 5 the failure-detecting capability of $a_{3\text{-}41\text{-}42}$ is much better than $a_{3\text{-}41\text{-}42}{}'$, verifies the guideline 5. We can also find that the failure-detecting capability of $a_{3\text{-}41\text{-}42}$ is better than its corresponding component *MRs*, but $a_{3\text{-}41\text{-}42}{}'$ is not. Because of similarity between the two sides of $a_{3\text{-}41\text{-}42}{}'$, some failure-detecting capabilities of $a_{3\text{-}41\text{-}42}{}'$ are hidden. It verifies the guideline 4 and 5.

Experiment is also conducted on SavAcc class according to the same methods and procedures. Experimental results showed that, under the similar mutation detection rate, the constructed number of *MRs* by new method is significantly less than the number constructed by GFT algorithm; after recombination, the mutation-detecting rate of *MR* has improved significantly, and verified the effectiveness of guidelines constructed *MR* proposed in this paper.

V. CONCLUSION

To solve the Oracle problem of methods sequence in object-oriented software testing, a method of *MRs* constructing for object-oriented software testing basted on algebraic specification was proposed. In metamorphic testing, we use returned value and object state to verify the equivalence between two execution results. Finally the improved method was verified through constructing *MRs* of IntStack and SavAcc class. The experiment results show that *MRs* redundancy is reduced by 66.7% at the same mutation score. So the new method has a low *MRs* redundancy and improves the efficiency of software testing.

REFERENCES

[1] CHEN T Y, CHEUNG S C. YIU S M, "Metamorphic testing: A new approach for generating next test cases," Technical Report HKUST, Hong Kong: Hong Kong University, 1998.

[2] Liu H, Kuo F CH, "How Effectively Does Metamorphic Testing Alleviate the Oracle Problem," IEEE Transactions on Software Engineering, vol. 40, no.1, pp.4-22, Dec. 2014.

[3] ZHAN W, SONG H, "Achievements and Challenges of Metamorphic Testing," in proc. WCSE 2013: Fourth World Congress on Software Engineering, 2013, pp.73-77.

[4] UPULEE K, "Techniques for Automatic Detection of Metamorphic Relations," in proc. ICSTV 2014: IEEE International Conference on Software Testing, Verification, and Validation Workshops, 2014, pp. 237-238.

[5] Upulee Kanewala, James M. Bieman, "Using Machine Leaning Techniques to Detect Metamorphic Relations for Programs without Test Oracles," in proc. the Software Reliability Engineering (ISSRE 2013), 2013, pp. 1-10.

[6] Huai Liu, Xuan Liu and Tsong Yueh Chen, "A New Method for Constructing Metamorphic Relations," in proc. the Quality Software(QSIC 2012), 2012, pp. 59-68.

[7] MURPHY C, VAUGHAN M, IIAHI W, KAISER G, "Automatic detection of previously-unseen application states for deployment environment testing and analysis," in proc. the 5th International Workshop on Automation of Software Test, 2010, pp. 16-23.

[8] Zhan-wei Hui, Song Huang, "A Formal Model For Metamorphic Relation," in proc. Decomposition Software Engineering (WCSE 2013), 2013, pp. 64-68.

[9] WANG R, BEN K, "Researches on basic criterion and strategy of constructing metamorphic relations," Computer Science, vol. 39, no.1, pp.115-119, Dec 2012.

[10] CHEN H Y, TSE T.H, CHAN F.T, CHEN T.Y, "In Black and White: An Integrated Approach to Class-Level Testing of Object-Oriented Programs," ACM Transactions on Software Engineering and Methodology, vol. 10, no. 3, pp. 250-295, Dec. 1998.

[11] CHEN H Y, TSE T.H, CHEN T.Y, "TACCLE:A Methodology for Object-Oriented Software Testing at the Class and Cluster Levels," ACM Transactions on Software Engineering and Methodology, vol. 10, no. 4, pp. 56-109, Dec. 2001.

[12] CHEN H Y, TSE T.H, "Automatic Generation of Normal Forms for Testing Object-Oriented Software,'' in proc. Ninth International Conference on Quality Software, 2009, pp. 108-116.

[13] DONG G W, NIE C, XU B, "Effectively metamorphic testing based on program path analysis," Chinese Journal of Computers, vol. 32, no. 5, pp.1002-1003, Dec. 2009.

[14] OFFUTT A J, LEE A, ROT H G, et a1, "An experimental determination of sufficient mutant operators," ACM Transactions on Software Engineering and Methodology, vol. 5, no. 2, pp. 99-118, Dec. 1996.

[15] DONG G W, XU B. et al, "Survey of Metamorphic Testing,'' Journal of Frontiers of Computer Science and Technology, vol. 3, no. 2, pp. 130-143, Dec. 2009.

[16] WU P, SHI X C,TANG J J, "Metamorphic testing and special case testing: A case study," Journal of software, vol.16, no. 7, pp.1210-1220, Dec. 2005.