Test Data Generation for Mutation Testing Based on Markov Chain Usage Model and Estimation of Distribution Algorithm

Changqing Wei, Xiangjuan Yao, Dunwei Gong, *Senior Member, IEEE* and Huai Liu, *Senior Member, IEEE*

***Abstract*—Mutation testing, a mainstream fault-based software testing technique, can mimic a wide variety of software faults by seeding them into the target program and resulting in the so-called mutants. Test data generated in mutation testing should be able to kill as many mutants as possible, hence guaranteeing a high fault-detection effectiveness of testing. Nevertheless, the test data generation can be very expensive, because mutation testing normally involves an extremely large number of mutants and some mutants are hard to kill. It is thus a critical yet challenging job to find an efficient way to generate a small set of test data that are able to kill multiple mutants at the same time as well as reveal those hard-to-detect faults. In this paper, we propose a new approach for test data generation in mutation testing, through the novel applications of the Markov chain usage model and the estimation of distribution algorithm. We first utilize the Markov chain usage model to reduce the so-called mutant branches in weak mutation testing and generate a minimal set of extended paths. Then, we regard the problem of generating test data as the problem of covering extended paths and use an estimation of distribution algorithm based on probability model to solve the problem. Finally, we develop a framework, TAMMEA, to implement the new approach of generating test data for mutation testing. The empirical studies based on fifteen object programs show that TAMMEA can kill more mutants using fewer test data compared with baseline techniques. In addition, the computation overhead of TAMMEA is lower than that of the baseline technique based on the traditional genetic algorithm, and comparable to that of the random method. It is clear that the new approach improves both the effectiveness and efficiency of mutation testing, thus promoting its practicability.**

***Index Terms*—Mutation testing, weak mutation, test data generation, Markov chain usage model, coverage of extended path, estimation of distribution algorithm.**

I. Introduction

MUTATION testing [1] is one main category of fault-based software testing techniques. It intentionally seeds various types of faults by applying a series of mutation operators to the source code to generate the mutants of the target program, which are named as mutants. Numerous studies [2] have consistently shown that the mutants have similar characteristics to real-life bugs, so mutation testing has been widely used as a measurement approach for the evaluation of a testing technique's fault-detection effectiveness: If a method can achieve a high mutation score (that is, the capability of killing a large number of mutants), it will be considered as effective in revealing different types of faults.

In addition to its popular usage in measuring testing effectiveness, mutation testing also provides a mechanism for generating test data that are supposed to deliver a high fault-detection capability [3]. Many test data generation methods have been proposed in this context from different perspectives. One major research direction is to make use of some code coverage information. Papadakis and Malevris [4], [5], for example, developed a branch coverage based technique to generate test data, aiming at killing mutants. However, the technique can only guarantee that one mutant is killed at one run, which may incur a high cost in test data generation. In order to solve this problem, Shan et al. [6], [7] used the iterative relaxation method to generate test data that can kill multiple mutants associated with the faults seeded at the same location in the code. Nevertheless, this method cannot generate test data that can kill mutants associated with different locations, and it requires the manual analysis of the program paths; therefore, the cost of this method is still high. Zhao and Gu [8] proposed a path coverage directed method based on the genetic algorithm to generate test data for killing mutants associated with different locations. However, such a method also requires the manual analysis to obtain paths for test data generation, which normally contain a large number of mutant branches. In a word, the existing methods are confronted

Manuscript received 1 April 2023; revised 13 January 2024; accepted 22 January 2024. Date of publication 24 January 2024; date of current version 15 March 2024. This work was supported in part by the National Natural Science Foundation of China under Grant 62373357 and Grant 42230704, in part by the Graduate Innovation Program of China University of Mining and Technology under Grant 2021WLKXJ074, in part by the 2021 Postgraduate Research and Practice Innovation Program of Jiangsu Province under Grant KYCX21_2136, in part by the Major Project of Natural Science Research of the Jiangsu Higher Education Institutions of China under Grant 21KJA520006, and in part by the Xuzhou Science and Technology Plan Project under Grant KC21007. Recommended for acceptance by L. Mariani. *(Corresponding author: Xiangjuan Yao.)*

Changqing Wei and Xiangjuan Yao are with the School of Mathematics, China University of Mining and Technology, Xuzhou 221116, China (e-mail: 1536113693@qq.com; yaoxj@cumt.edu.cn).

Dunwei Gong is with the College of Automation and Electronic Engineering, Qingdao University of Science and Technology, Qingdao, Shandong 266061, China (e-mail: dwgong@vip.163.com).

Huai Liu is with the Department of Computing Technologies, Swinburne University of Technology, Melbourne, VIC 3122, Australia (e-mail: hliu@swin.edu.au).

Digital Object Identifier 10.1109/TSE.2024.3358297

with several challenges, e.g., the high load in manual analysis and the repeated execution of many mutant branches, which cause a huge computation cost in test data generation, thus hindering the practicality of mutation testing.

To address the above challenges, we propose a new approach for efficiently generating test data that are effective in killing multiple mutants associated with the faults seeded in different locations. Specifically, the approach leverages the *Markov chain usage model* (MCUM), to reduce mutant branches and generate a set of extended paths. Treating the problem of generating test data as the problem of covering extended paths, the approach further makes use of an *estimation of distribution algorithm* (EDA) based on probability model to generate test data of killing multiple mutants at the same time. We develop a framework, ***TAMMEA***, to implement such an approach to ***T***est d***A***ta generation for ***M***utation testing based on ***M***arkov chain usage model and ***E***stimation of distribution ***A***lgorithm. We conducted a series of experiments to evaluate the performance of TAMMEA based on fifteen Java programs, for which mutants are constructed by using MuClipse [9], an Eclipse plug-in for MuJava [10].

Our work makes the following three major contributions:

- We, for the first time, apply MCUM into mutation testing. In TAMMEA, we use MCUM to provide a systematic mechanism for obtaining a reduced set of extended paths, which are the basis for test data generation.
- Considering the unique stochastic features of MCUM, we select EDA, instead of the traditional genetic algorithm, to solve the problem of covering extended paths and thus to quickly and accurately generate test data for killing as many mutants at the same time as possible.
- Our empirical studies demonstrate that TAMMEA can improve the efficiency of test data generation process in mutation testing as well as guarantee the high fault-detection effectiveness of generated test data. The performance improvement helps promote the cost-effectiveness and applicability of mutation testing.

The rest of this paper is organized as follows. In Section II, we introduce the background knowledge for this paper. Section III presents the TAMMEA framework, including the method of constructing MCUM, the automatic generation of extended paths based on MCUM, and the EDA-based test data generation for covering extended paths. Our experimental design and setting are described in Section IV. In Section V, we discuss and analyze the experimental results. The related work is summarized in Section VI. Finally, Section VII concludes the paper and points out some future work.

II. Background

A. Mutation Testing

Basically speaking, mutation testing is a fault-based software testing technique. In mutation testing, some minor changes are made in the code of the target program, thus creating a set of mutants. The change rules in the process of mutation testing are normally referred to as the *mutation operators* [11]. For a test datum t and a mutated statement s', researchers use these three conditions to determine whether or not it kills a mutant:

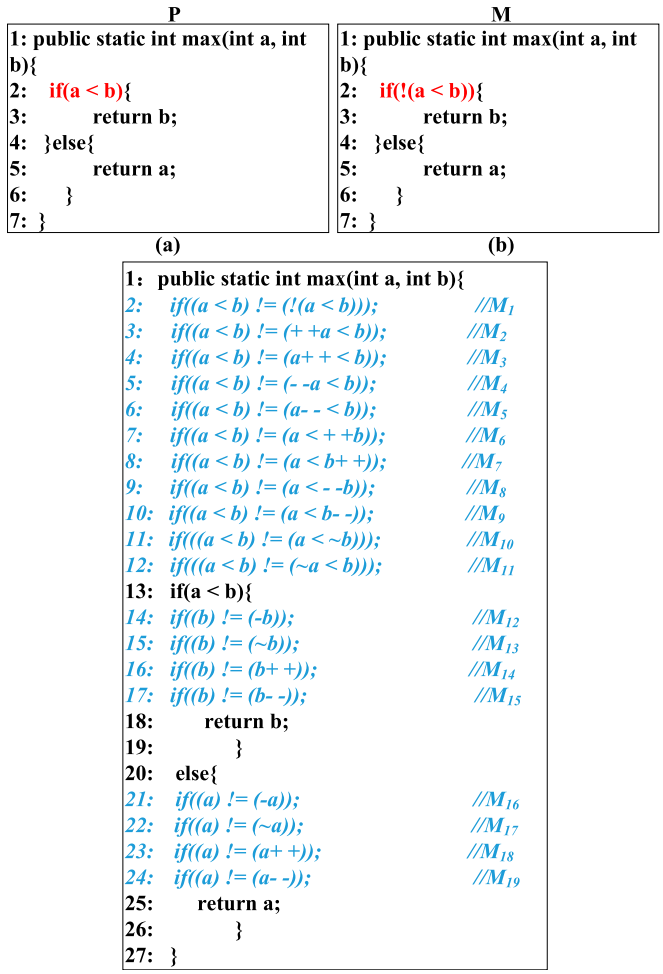


Fig. 1. A simple Java program for finding the maximum number of two integers.

(1) *accessibility* — t must be able to reach the position of s'; (2) *necessity* — t must change the program state after executing s'; and (3) *sufficiency* — t must cause the output of the mutant to be different from that of the original program.

One main purpose of mutation testing is to generate test data that kill as many mutants as possible. There exist some methods [6], [12], [13] for generating test data that can satisfy all three conditions, which is termed as strong mutation testing [14]. Nevertheless, it is normally very expensive and time-consuming to implement strong mutation testing due to the high cost incurred by the large number of mutants to be executed. As such, some researchers, such as Papadakis and Malevris [4], investigated the use of weak mutation testing (only satisfying *accessibility* and *necessity*) for test data generation. One major approach to weak mutation testing is to use *mutant branches* [15] to replace mutants. A meta-program is constructed that can be compiled only once based on the *mutant schemata* [16] technique. This mechanism effectively transforms the problem of killing mutants into the problem of making mutant branches hold true value.

Let us look at the following example for illustration. Suppose that a simple Java program P (given in Fig. 1(a)) attempts to find the maximum number of two integers. The statement

"$if(a < b)$" in P is changed to "$if(!(a < b))$", thus constructing a mutant M, shown in Fig. 1(b). According to the weak mutation criterion, a mutant branch "$if((a < b)$!= $(!(a < b)))$" can be generated to replace M. Similarly, many mutants can be generated, and the corresponding mutant branches can be added into a meta-program (Fig. 1(c)).

Specifically, some mutants cannot be killed by any test data in mutation testing and they are called *equivalent mutants* [11], [17]. For example, when the statement "$returnb$;" from P (Fig. 1(a)) is changed to "$returnb$++;", the corresponding mutant can be considered as an equivalent mutant.

Mutation testing can help measure the quality of a test suite, normally using the metric *mutation score* (MS), which is defined as:

$$MS = \frac{N_{kill}}{N_M - N_{eq}} \times 100\%, \tag{1}$$

where N_{kill}, N_M, and N_{eq} refer to the number of mutants killed by the test suite, the total number of mutants, the number of equivalent mutants, respectively. Note that we can calculate N_{kill} based on the strong or weak mutation testing, which means that MS can have two categories: MS_{weak} and MS_{strong}.

Previous studies [18], [19] have shown that the test data generated by weak mutation testing can also meet the strong mutation testing to a large extent. Therefore, the low cost weak mutation testing has become an important approach for promoting the application of mutation testing.

B. Markov Chain Usage Model

Markov chain has been widely used in various fields, including physics, biology, computer science, and pattern recognition. It also plays an important role in software testing to improve its performance [20]. In particular, MCUM (Markov chain usage model) attempts to utilize the Markov chain (MC) to simulate the usage of software program especially for the assessment and improvement of software reliability [20], [21]. The MC studied in this paper is a Markov process with discrete time and state [22], which has been widely used for statistical software testing [21]. To illustrate, suppose that in a program P, every statement is treated as a state. If the execution process of P is regarded as a stochastic process with Markov property, discrete time and state, then it can be called a MC, which is denoted as $\{X_n = X(n), n = 0, 1, 2, \cdots\}$. The Markov property hereby means that in the state of time point t_0, the probability of the occurrence of the next time point t_1 is only related to the state of the time point t_0, and has nothing to do with the state before t_0.

A simple MCUM (without considering incentives) generally consists of a structural model (SM) and state transition probability (STP) for each edge [22]. SM [23] is composed of start node, finite state nodes, edges, and end node. The basic steps for constructing MCUM is as follows.

First of all, we construct SM by continuously improving the granularity [23]. Suppose that the granularity of the initial software usage model has only three states: (1) Start state; (2) Usage state; (3) End state. According to these three states, we can continuously refine the granularity and finally build the required SM.

Then, we determine the STP with a certain probability distribution method. There are different methods of determining STP, such as the allocation without knowledge (AWK), the knowledge allocation (KA), and the expected allocation (EA) [21]. For AWK, the testers will determine the STP for each edge by the average allocation method (AAM) without understanding the program. Note that AAM refers to that if there are n output edges in the current state, the STP of each edge is $1/n$. This paper only focuses on AWK due to its convenience and strong operability. Readers interested in other methods (KA and EA) can refer to the literature [21].

As a remark, in this study, MCUM is actually a directed graph (SM) with weights and each of weights represents an STP. The larger the weight, the greater the probability of being executed next time.

C. Estimation of Distribution Algorithm

Estimation of distribution algorithm (EDA) [24] is the evolutionary method based on probability model, which is different from the classical genetic algorithm (GA) — In EDA, the probability distribution of genes replaces crossover and mutation [25]. EDA guides the direction of population evolution by collecting excellent solution sets and statistical information of their overall gene probability model, which effectively resolves the issues related to the randomness of genetic operations in GA, and thus improves the ability of global search.

Researchers have proposed many EDAs based on whether there is a dependency between genetic variables, mainly including the algorithm with independence of variables and the algorithm with interdependence of binary variables (or multivariables). Furthermore, the EDA based on independence of variables have three methods: the population based incremental learning algorithm (PBIL) [20], the distribution algorithm based on univariate edge [26], and the genetic algorithm based on compression [27]. In this study, we will use PBIL [20] to generate test data for weak mutation testing. Since the birth of PBIL, it has been frequently used in various application fields [20]. The steps of PBIL can be summarized as follows:

- **Step 1:** Set population size N. Let $g = 0$, where g is the value of generation. Initialize the sampling probability vector $P^{(g)} = (p_1^{(g)}, p_2^{(g)}, \cdots, p_l^{(g)})$, where l is the length of genes and each individual consists of l genes.
- **Step 2:** Generate the population of generation g, $Pop = \{X_1, X_2, \cdots, X_N\}$ by sampling N times according to $P^{(g)}$, where the j-th gene of X_i is denoted as X_i^j.
- **Step 3:** Calculate the fitness and select M $(M < N)$ best excellent individuals from N individuals, denoted as $X_{k_1}, X_{k_2}, \cdots, X_{k_M}$. Update $P^{(g)}$ based on Formula (8) (Section III.C.1).
- **Step 4:** Let $g = g + 1$ and go to **Step 2** until obtaining the optimal solution.

As an emerging intelligent technology, EDA has been widely used in many areas. This study, for the first time, applies it

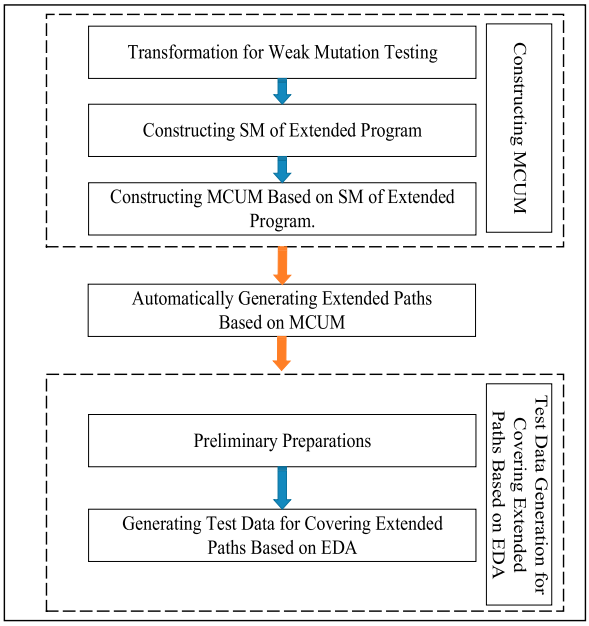


Fig. 2. Framework of TAMMEA.

into the field of mutation testing, particularly to improve its efficiency in test data generation.

III. Proposed Approach

The main aim of this study is to improve the effectiveness and efficiency of test data generation in mutation testing. We propose adopting MCUM for constructing extended paths based on the mutant branches in weak mutation testing and use EDA to solve the problem of covering the extended paths, thus generating test data. After that, we implement such an approach to the ***T***est d***A***ta generation for ***M***utation testing based on ***M***CUM and ***EDA*** in a so-called ***TAMMEA*** framework, which is depicted in Fig. 2.

As observed from Fig. 2, TAMMEA is composed of three main modules: (1) The construction of MCUM; (2) The MCUM-based automatic generation of extended paths; and (3) The EDA-based test data generation for covering extended paths.

For module (1) (Section III-A), we propose a method of constructing MCUM, which mainly includes the transformation for weak mutation testing (Section III.A.1), the construction of SM for the extended program (Section III.A.2), and the construction of MCUM based on the SM of extended program (Section III.A.3).

For module (2) (Section III-B), we give two definitions (Definition 2 and Definition 3, given in Section III.B.1) and develop an automatic method (Section III.B.2) for generating extended paths based on the MCUM from module (1).

In module (3) (Section III-C), we first undertake some preliminary preparations, including initialization of population, individual encoding and decoding, construction of fitness function and updating probability vector (Section III.C.1). Then, we use an EDA-based method (Section III.C.2) to generate test data for covering the extended paths from module (2).

TABLE I
The Mutant Branches

Mutants	Mutant Branches
M_1	$if((a<b)!=(!(a<b)))$
M_2	$if((a<b)!=((+ + a<b)))$
M_3	$if((a<b)!=((a+ + <b)))$
M_4	$if((a<b)!=((- - a<b)))$
M_5	$if((a<b)!=((a- - <b)))$
M_6	$if((a<b)!=((a<+ + b)))$
M_7	$if((a<b)!=((a<b+ +)))$
M_8	$if((a<b)!=((a<- - b)))$
M_9	$if((a<b)!=((a<b- -)))$
M_{10}	$if((a<b)!=((a<\sim b)))$
M_{11}	$if((a<b)!=((\sim a<b)))$
M_{12}	$if((b)!=((-b)))$
M_{13}	$if((b)!=((\sim b)))$
M_{14}	$if((b)!=((b+ +)))$
M_{15}	$if((b)!=((b- -)))$
M_{16}	$if((a)!=((-a)))$
M_{17}	$if((a)!=((\sim a)))$
M_{18}	$if((a)!=((a+ +)))$
M_{19}	$if((a)!=((a- -)))$

Next, we will use Example 1 to illustrate the specific steps of the three modules in the following sections.

Example 1: For program P (Fig. 1(a)) for finding the maximum number of two integers, we use $MuCplise$ [10] (a mutation testing tool) to generate the mutant M (as shown in Fig. 1(b)) and obtain 19 mutants ($M_1 \sim M_{19}$) similar to the mutant M. Based on these mutants, we construct mutant branches (as shown in Table I) by the transformation method for weak mutation testing [15].

A. Construction of MCUM

This section will present the method of constructing MCUM.

1) Transformation for Weak Mutation Testing: In mutation testing, if we judge whether a mutant is killed based on the original program and the state of the mutated point of a mutant, the corresponding mutation testing criterion is called *weak mutation testing* [19]. Based on this, we give a transformation method for weak mutation testing.

For a source program P, we get a mutated statement s^m after implementing a mutation operator on a statement s^o of P. If s^m replaces s^o, we obtain a mutant M. After that, we create a number of mutants by applying a series of mutation operators to different statements of P. Let M_{all} denote the set of all mutants generated by $MuClipse$ [10] for Java programs. Note that the equivalent mutants [11], [17] have been removed from M_{all} (we describe how we remove equivalent mutants in Section IV-D).

Furthermore, we use a test datum t to execute s^o in P and s^m in M, respectively. If t's execution cause different states of P and M, we say t kills the mutant M under the weak mutation testing. After the execution of s^o and s^m, we regard the different states of P and M as the true value of predicate "s^o != s^m". If "s^o != s^m" is treated as a branch predicate, a so-called mutant branch [15] like "$if(s^o$!= $s^m)$" can be created. Note that we use the log file $mutation\text{-}log$ from $MuClipse$ [10] to construct mutant branches. In a word, we transform the problem of killing M into the problem of covering the true value of the branch $if(s^o$!= $s^m)$.

```
//Mutation operators:Code lines:methods:
//Operation before mutation =>Operation after mutation
AOIU_1:10:int_max(int,int):b => -b
AOIU_2:12:int_max(int,int):a => -a
AOIS_1:9:int_max(int,int):a => ++a
AOIS_2:9:int_max(int,int):a => --a
AOIS_3:9:int_max(int,int):a => a++
AOIS_4:9:int_max(int,int):a => a--
AOIS_5:9:int_max(int,int):b => ++b
AOIS_6:9:int_max(int,int):b => --b
AOIS_7:9:int_max(int,int):b => b++
AOIS_8:9:int_max(int,int):b => b--
AOIS_9:10:int_max(int,int):b => b++
AOIS_10:10:int_max(int,int):b => b--
AOIS_11:12:int_max(int,int):a => a++
AOIS_12:12:int_max(int,int):a => a--
COI_1:9:int_max(int,int): a < b => !(a < b)
LOI_1:9:int_max(int,int):a => ~a
LOI_2:9:int_max(int,int):b => ~b
LOI_3:10:int_max(int,int):b => ~b
LOI_4:12:int_max(int,int):a => ~a
```

Fig. 3. The $mutation\text{-}log$ from 19 mutants.

```
1:  public static int max(int a, int b){
2:      if((a < b) != (!(a < b)));
3:      if((a < b) != (a - 1 < b));
4:      if(((a < b) != (a + 1 < b)));
5:      if((a < b) != (a < b + 1));
6:      if((a < b) != (a < b - 1));
7:      if(((a < b) != (a < ~b)));
8:      if(((a < b) != (~a < b)));
9:      if(a < b){
10:         if((b) != (-b));
11:         if((b) != (~b));
12:         return b;
13:                 }
14:     else{
15:         if((a) != (-a));
16:         if((a) != (~a));
17:         return a;
18:                 }
19:  }
```

Fig. 4. The P^{new} obtained by inserting mutant branches.

Since each mutant M in M_{all} is associated with a unique mutant branch, we get the set of all mutant branches, denoted as B^M. Furthermore, we insert all non-equivalent mutant branches in B^M in front of their corresponding statements in P one by one, so that we get an **extended program**, denoted as P^{new}.

Note that each mutant branch inserted in P is only used to determine whether the corresponding mutant is killed, but not to change any operation in P. For example, we need to use v+1 (v-1) instead of $++v$ ($--v$) and make use of v substitute for $v++$ and $v--$ when we encounter $++v$ ($--v$) and $v++$ ($v--$).

The following illustrates the concrete steps of transformation for weak mutation testing based on Example 1.

1) For a source program P from Fig. 1(a), we use $MuCplise$ [10] to generate 19 mutants ($M_1 \sim M_{19}$) after applying 4 mutation operators (AOIS, AOIU, COI and LOI). Note that here we only give the form of M_1 (Fig. 1(b)), because other mutants are similar to M_1 and will not be shown.
2) On the basis of 19 mutants, we use $mutation\text{-}log$ (Fig. 3) from $MuClipse$ [10] to construct mutant branches (Table I).
3) After removing 8 equivalent mutants (such as M_3 and M_5) using methods from prior literature [15], we insert the remaining 11 non-equivalent mutant branches in front of their corresponding statements in P one by one, so that we get an P^{new} (Fig. 4). Note that here we use v+1 (v-1) instead of $++v$ ($--v$).

2) Construction of SM of Extended Program: After obtaining the extended program P^{new} (Section III.A.1), we construct the SM of P^{new} for the construction of MCUM.

In P^{new}, if we directly insert the mutant branches in front of the original statements, it will lead to a large number of non-executable paths as well as a huge amount of mutant branches that require repeated executions when generating test data. All these incur a high cost, such as long computation time. To address these issues, we give the following two specific rules for reducing the size of P^{new} and thus obtaining a new program P_{after}.

- **Combination Rule (COMR)**. Given mutant branches generated by the mutated statements in the same position, we combine them based on contradictory or non-contradictory. Here, we accurately determine the contradiction relationship of different predicates by some existing satisfiability module theory (SMT) solvers, such as $Z3$[1]. In using $Z3$, we first construct a solver $Solver()$ and add different predicates to it. Then we use a check function $check()$ in $Solver()$ to automatically determine whether there is a feasible solution. If so, the different predicates are not contradictory. Note that if the mutant branches are corresponding to those mutants that can be killed as long as accessibility is satisfied (such as those with lines 2, 11 and 16 in Fig. 4), they need to be reserved separately for construction of MCUM and not participate in the combination.
- **Distribution Rule (DR)**. Given the combined mutant branches obtained by COMR, they can generally be divided into branch predicate mutant branches (BPMBs) and non-branch predicate mutant branches (NBPMBs). For NBPMBs, we can allocate them according to the paths they belong to. For BPMBs, we can allocate them based on whether they contradict with the branch predicates — if they are not contradictory, then they are placed on the true branch; otherwise, they are placed on the false branch (shown in Fig. 6).

The main difference of P_{after} from P^{new} is that it inserts mutant branches into the relevant positions of predicates, which is reflected as inserting a subgraph constructed by mutant branches into the corresponding edges in the CFG of P. Thus, we first need to construct subgraphs that are composed of mutant branches to build a concise SM based on the following two rules, namely Connection Rule (CONR) and Restriction Rule (RR).

- **Connection Rule (CONR)**. Given the mutant branches (including the combined mutant branches) from the same or different positions, we connect non-contradictory mutant branches (or combined mutant branches). Note that we

[1]http://research.microsoft.com/projects/z3

```
1:  if((a < b) != !(a < b))
2:  if((b) != (-b))
3:  if(((a < b) != (a < ~b))&&((a < b) != (~a < b))&&((a < b) !=
(a + 1 < b))&&((a < b) != (a < b - 1)))
4:  if((b) != (~b))
5:  if((a) != (-a))
6:  if(((a < b) != (a - 1 < b))&&((a < b) != (a < b + 1)))
7:  if((a) != (~a))
```

Fig. 5. The mutant branches and the combined mutant branches based on COMR.

```
public static int max(int a, int b){
   if(a < b){
1:        if((a < b) != (!(a < b));
2:        if(((b) != (-b)));
3:        if(((a < b) != (a < ~b))&&((a < b) != (~a <b))&&((a
< b) != (a + 1 < b))&&((a < b) != (a < b - 1)));
4:        if((b) != (~b));
          return b;
            }
    else{
5:      if((a) != (-a)) ;
6:      if(((a < b) != (a - 1 < b))&&((a < b) != (a < b + 1)));
7:      if((a) != (~a));
          return a;
            }
 }
```

Fig. 6. A reduced program P_{after} based on extended program P^{new}.

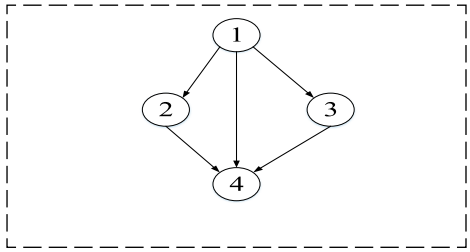


Fig. 7. The G_1 contained in the true branch of $if(a < b)$.

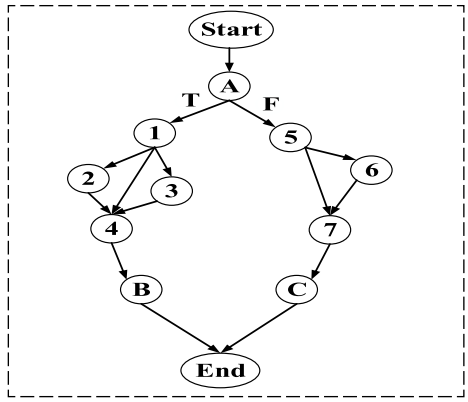


Fig. 8. The SM of extended program P^{new}.

mainly consider the following two main reasons for connecting the non-contradictory mutant branches (NCMBs): 1) There must be test data covering the true branches of two NCMBs in P_{after}; 2) Generating test data of killing more mutants is equivalent to generating test data of covering more true branches of mutant branches. Note that the combined mutant branches (CMBs) are the maximum CMBs in the subgraph except for the mutant branches of the start node and the end node. In addition, the connection here refers to the direct connection and the process of connecting is not affected by the order of connection.

- **Restriction Rule (RR)**. Given the CFG of P, we restrict the connection mode of the original nodes in CFG to remain unchanged, and only directly connect the original nodes to the start and end nodes of the embedded subgraph. In other words, the mutant branches represented by the start node and end node of the subgraph must be the data with true branches, and only one edge of the start node and end node is directly connected to the original node. In fact, such nodes (such as the numbers 1, 4, 5 and 7 in Fig. 8) exist and they happen to be the mutant branches corresponding to those mutants that can be killed as long as accessibility is satisfied. Note that the mutant branches represented by nodes 1 and 4 are used if multiple branches are combined into one.

After obtaining the subgraphs composed of mutant branches, we add them into the corresponding edges of the CFG of P, thus obtaining the final SM for the construction of MCUM.

The following illustrates the concrete steps of construction of SM of extended program P^{new} based on Example 1.

1) For 11 mutant branches (Lines 2-8, 10-11 and 15-16) of P^{new} (Fig. 4), we implement COMR on 8 of 11 mutant branches and obtain the final results as shown in Fig. 5. Note that the 3 mutant branches (Line 2, 11 and 16 in Fig. 4) do not participate in the combination and two combined mutant branches (Lines 3 and 6 in Fig. 5) are obtained based on $Z3$ during the implementation of COMR.
2) Based on the mutant branches from Fig. 5, we reduce and update P^{new} (Fig. 4) by DR for obtaining P_{after} (Fig. 6).
3) For the true branch of the branch statement $if(a < b)$ from Fig. 6, we connect the mutant branches 1 and 2 (or 2 and 4) without contradiction according to CONR. In this way, we obtain 1→2→4 (meaning the true branch covering mutant branches 1, 2 and 4 at the same time). Furthermore, we get 1→3→4 and 1→4. Similarly, for the mutant branches 5, 6, and 7 contained in the false branch of $if(a < b)$, we get 5→6→7 and 5→7.
4) We combine 1→2→4, 1→3→4 and 1→4 to obtain a subgraph G_1 (Fig. 7). Similarly, we obtain a subgraph G_2 composed of 5→6→7 and 5→7.
5) We insert G_1 and G_2 into the corresponding edges of the CFG of P based on RR, and construct the final SM (Fig. 8) for the construction of MCUM. Note that the nodes A, B and C in Fig. 8 correspond to lines 2, 3 and 6 of P in Fig. 1(a), respectively.

3) Construction of MCUM Based on SM of Extended Program: After obtaining SM (Section III.A.2), we give **state transition probability** (STP), which is defined as below:

Definition 1: Given an extended program P^{new}, $T = \{\tau_0, \tau_1, \cdots\}$ is a set of discrete moments and each statement (including mutant branch) of P^{new} is recognized as a state. During the execution of P, we obtain a discrete state space $I = \{a_1, a_2, \cdots, a_\omega\}$ (where ω is the number of statements). Suppose that X_i is the state of moment τ_i, $i = 0, 1, \cdots$. Let $A_i = \{X_i = b_i\}$, where $b_i \in I$. Then:

$$p_{ij} = p(A_j|A_0A_1\cdots A_i) = p(A_j|A_i).$$

where $i < j$. We call p_{ij} as the state transition probability from b_i to b_j, denoted as STP. STP has the following important property:

$$\sum_{j=1}^{\omega} p_{ij} = 1,$$

where $i = 1, 2, \cdots, \omega$.

In order to determine STP, we give a method to determine the probability and the specific steps are as follows.

First, we give the preliminary probability of SM based on average allocation method (Section II-B) for each edge of SM (Fig. 8) of P^{new}.

Then, we assume that the current state is i (not the end state) and j is i directly reaching the possible state in the process of traversing the SM (Fig. 8), then we adjust the size of STP based on Formula (2) until i is the end state. Note that nodes are called states in the SM and Formula (2) is given as follows.

$$\begin{cases} \Delta p_{ij} = \frac{1}{n_i} p_{ij} & j = 1, 2, \cdots, n_i \\ p_{ij} = p_{ij} + \Delta p_{ij} & j \neq h \\ p_{ij} = p_{ij} - \sum_{k=1,k\neq h}^{n_i} \Delta p_{ik} & j = h \\ \sum_{j=1}^{n_i} p_{ij} = 1 \end{cases}, \quad (2)$$

where p_{ij} is the STP from the i-th state to the j-th state, n_i is the total number of states transferred from the i-th state to the j-th state, h is any reachable state of n_i states.

Note that when using Formula (2) to update the probability, the STP that has been updated does not need to be updated repeatedly and p_{ij} should be greater than or equal to 0 (that is, p_{ij} - $\sum_{k=1,k\neq h}^{n_i} \Delta p_{ik} \geq 0$), otherwise it is meaningless.

In the end, we check whether there is any state that has not been traversed. If so, continue to adjust the size of STP based on Formula (2), otherwise stop the cycle to output the last STP.

The following illustrates the concrete steps for determining STP based on Example 1.

1) Based on the SM (Fig. 8) of P^{new}, we first adopt AAM (Section II-B) to preliminarily give an STP (Fig. 9) for each edge of the SM. For example, if the current state is A and it has two transferable states (1 and 2), we easily get that the STPs of A to 1 and A to 2 are 1/2 and 1/2.
2) In the process of traversing the SM (Fig. 8), we assume that the current state i is 1 and h is 4. After that, we easily obtain $n_i = 3$ and calculate the STPs of 1 to 2, 1 to 3 and 1 to 4 by Formula (2) based on preliminary STPs (as shown in Fig. 9) are 1/9, 4/9 and 4/9. Similarly, when i is 4 and h is 7, n_i is 2 and the STP of 5 to 7 (5 to 6) is 1/4 (3/4).
3) Repeat (2) until the final result (Fig. 10) is output without any remaining status that has not been updated. Note that we don't need to update the STP that has been updated.

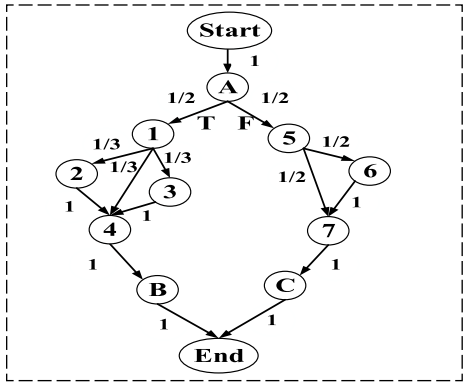


Fig. 9. The preliminary probability of SM based on AAM.

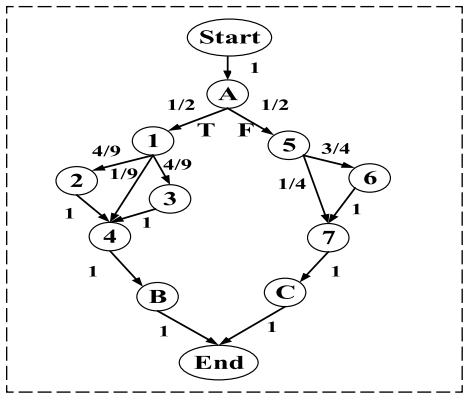


Fig. 10. A final result of STPs based on Formula (2) for Example 1.

After giving the STP, we will give a general rule for constructing MCUM as follows:

- **General Rule**. Given an extended program, the following two conditions should be met: (1) A SM must be constructed for the extended program; (2) Each edge of SM of an extended program must have a STP.

According to **General Rule**, MCUM can be quickly constructed for small-scale programs, but it is not easy to construct MCUM quickly for large-scale programs. Therefore, we propose a semi-automatic method of constructing MCUM, which is named as SA-MCUM and the pseudo code to realize this process is shown in Algorithm 1.

In Algorithm 1, we first obtain the adjacency matrix W based on CFG of P and the shortest path S generated by the $Dijkstra$ algorithm [10] (Lines 3 to 5) and transform W into W_1, whose most elements, except for diagonal 0, become ∞ in the process of obtaining S. Note that we automatically generate the CFG by some existing tools, such as $Visustin$[2]. After that, we construct the MCUM of S by four specific rules and the general rule (Line

[2]https://visustin.en.softonic.com

Algorithm 1 SA-MCUM

Input: Source program P and CFG of P
Output: The complete MCUM
1: Set N_{unused} = N_{all}
2: **while** $N_{unused} \neq 0$ **do**
3: Transform the CFG into the adjacency matrix W
4: Convert W to W_1
5: Obtain the shortest path S for CFG by $Dijkstra$ algorithm [10]
6: Generate the MCUM for S by four specific rules and a general rule
7: Update the CFG by the obtained shortest path S
8: $N_{unused} \longleftarrow N_{unused}$ - N_{used}
9: **end while**

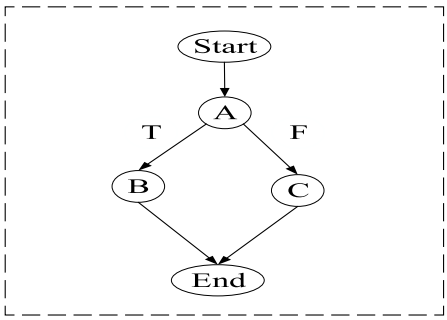


Fig. 11. The CFG of P from Example 1.

6) and update the CFG with S (Line 7). During the process of updating the CFG, if S contains only one branch predicate pr, all nodes between pr and the end node in S will be deleted from the CFG; otherwise, all nodes between the last pr and the end node in S will be deleted from the CFG. Finally, we repeat the previous process after updating N_{unused} until N_{unused} = 0 is satisfied and output the complete MCUM (Lines 2 to 9). Here, N_{unused} represents the remaining mutant branches (or paths) and is initially set to N_{all} (Line 1), which means all mutant branches (or paths). In addition, N_{used} indicates mutant branches or paths used in the current iteration (Line 8).

The following illustrates the concrete steps for constructing MCUM based on Example 1.

1) For the program P in Fig. 1(a), we modify the CFG according to $Visustin$ (as shown in Fig. 11), and convert it into adjacency matrix W.
In Fig. 11, T (F) represents true (false) branch and the nodes A, B and C represent predicates (or statements). According to Fig. 11, we can obtain its adjacency matrix W as follows:

$$W = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix},$$

where 0 means that the directly connected edge does not exist between two nodes; by contrast, 1 means that the directly connected edge exists between two nodes.

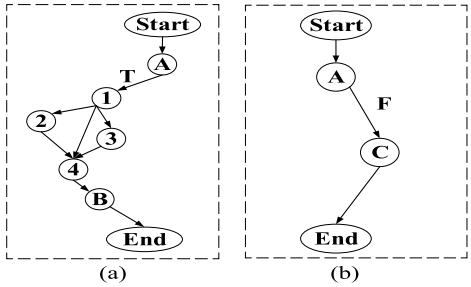


Fig. 12. The SM of S_1^{sh} and the updated CFG of P.

2) We transform W into W_1 and obtain the shortest path S_1^{sh} (S_1^{sh} = <Start→A→B→End>) by the $Dijkstra$ algorithm [10]. Note that the zero elements of W are all converted to ∞ except for zero element of the diagonal.

$$W_1 = \begin{bmatrix} 0 & 1 & \infty & \infty & \infty \\ \infty & 0 & 1 & 1 & \infty \\ \infty & \infty & 0 & \infty & 1 \\ \infty & \infty & \infty & 0 & 1 \\ \infty & \infty & \infty & \infty & 0 \end{bmatrix}$$

3) According to four specific rules (COMR, DR, CONR and RR), we obtain the SM of S_1^{sh} (as shown in Fig. 12(a)).
4) Update CFG of P after deleting the specified nodes based on the obtained shortest path. For example, S_1^{sh} has only one branch predicate pr, so we delete other nodes besides the end node after the predicate pr for updating CFG of P (Fig. 12(b)). Note that when there are multiple branch predicates in the shortest path, you need to delete the nodes after the last predicate.
5) Judge whether there are any remaining paths based on the updated CFG of P, if so, repeat (1) to (4) until there are no remaining paths to output the SM; otherwise, directly output the SM. For example, if we find that the path still exists in the updated CFG of P, we obtain the shortest path S_2^{sh} (S_2^{sh} = <Start→A→C→End>) and repeat (1) to (4) until there are no remaining paths output the complete SM (as shown in Fig. 8).
6) According to the constructed SM of the extended program (Fig. 8), we add the STP to each edge based on our method (Section III.A.3) and finally get the MCUM of the extended program (as shown in Fig. 10).

B. Automatic Generation of Extended Paths Based on MCUM

Prior to presenting the method for automatically generating the extended paths based on MCUM, let us first introduce two foundational definitions.

1) Definitions: Based on extended program P^{new}, we will define the **extended path** as follows:

Definition 2: Suppose the reduced program P_{after} (Fig. 6) of P^{new} and S^{sq} is an executed sequence from the MCUM of P^{new}, we easily obtain the executed path corresponding to S^{sq} in P_{after} and S^{sq} (S^{sq} corresponds to the executed path one by one) is called the extended path, denoted as S^{ep} (such as S_1^{ep} in Table II). Here, S^{sq} = $<s_{start} \rightarrow s_1^o \rightarrow s_1^b \rightarrow s_2^b \rightarrow s_2^o \rightarrow$

TABLE II
THE FIVE EXTENDED PATHS BEFORE REDUCTION

S^{ep}	Expressions of S^{ep}
S_1^{ep}	<$Start \rightarrow A \rightarrow 1 \rightarrow 2 \rightarrow 4 \rightarrow B \rightarrow End$>
S_2^{ep}	<$Start \rightarrow A \rightarrow 1 \rightarrow 3 \rightarrow 4 \rightarrow B \rightarrow End$>
S_3^{ep}	<$Start \rightarrow A \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow C \rightarrow End$>
S_4^{ep}	<$Start \rightarrow A \rightarrow 5 \rightarrow 7 \rightarrow C \rightarrow End$>
S_5^{ep}	<$Start \rightarrow A \rightarrow 1 \rightarrow 4 \rightarrow B \rightarrow End$>

$\cdots \rightarrow s_{m-1}^0 \rightarrow s_{m-1}^b \rightarrow s_m^b \rightarrow s_m^o \rightarrow s_{end}>$, where s_{start} is the $Start$ node, s_{end} is the End node, s_i^b (i = 1, 2, $\cdots$, m) is the i-th node representing mutant branch (or the combined mutant branches) and s_i^o (i = 1, 2, $\cdots$, m) is the i-th node representing original statement of P_{after}.

After obtaining the extended path, we will give the STP matrix for generating the extended paths and the STP matrix is defined as:

Definition 3: Suppose there are n states (sentences) for the MCUM of P, if the STP (Definition 1) between any two states i (i = 1, 2, $\cdots$, n) and j (j = 1, 2, $\cdots$, n) in the MCUM of P is known, then the STP matrix ($n \times n$), denoted as $\mathbf{P_{matrix}}$ is as follows:

$$\mathbf{P_{matrix}} = \begin{bmatrix} p_{11} & p_{12} & \cdots & p_{1n} \\ p_{21} & p_{22} & \cdots & p_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ p_{n1} & p_{n2} & \cdots & p_{nn} \end{bmatrix},$$

where p_{ij} ($0 \leq p_{ij} \leq 1$ and i, j = 1, 2, $\cdots$, n) is the STP from state i to state j, and $\sum_{j=1}^{n} p_{ij} = 1$ (property of Definition 1). Note that it is necessary to set $p_{n1} \in P_{matrix}$ to 1 in order for the model to run circularly, and the STP of no connecting edge between other states in the model is set to 0.

For example, the following is the specific form of P_{matrix} corresponding to the MCUM (Fig. 10) of P^{new} for Example 1 based on Definition 3.

$$P_{matrix} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{4}{9} & \frac{4}{9} & 0 & \frac{1}{9} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{3}{4} & 0 & \frac{1}{4} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix},$$

where each row (or column) in P_{matrix} corresponds to the state from top to bottom (or from left to right) in MCUM (Fig. 10) one by one. Each element in P_{matrix} represents the STP and $\sum_{j=1}^{12} p_{ij} = 1$ ($0 \leq p_{ij} \leq 1$, and i = 1, 2, $\cdots$, 12).

2) MCUM-Based Generation of Extended Paths: Based on Definition 2 and Definition 3, we give a new method for automatically generating extended paths based on MCUM, which is named as EP-MCUM. Our method makes full use of roulette in GA [28] for generating extended paths, as detailed in Algorithm 2.

Algorithm 2 EP-MCUM

Input: The MCUM of extended program
Output: The set Φ of extended paths
1: Set $\Pi = \phi$, $\Phi = \phi$ and $|s|_{unused} = |s|_{all}$
2: **while** $|s|_{unused} \neq 0$ **do**
3: Set $s_{next} = s_{Start}$
4: Traverse the MCUM
5: **while** $s_{next} \neq s_{End}$ **do**
6: Construct a sequence $\{H_j\}$ based on Formula (3)
7: Update the current state based on random function
8: **if** $H_{j-1} < R_i < H_j$ **then**
9: Select the edge corresponding to p_{ij} as the exit edge of state i, and add it to Π
10: $i \longleftarrow i + 1$
11: **else**
12: Generate the random number R_i again
13: **end if**
14: **end while**
15: Add Π to Φ and $\Pi = \phi$
16: Update the STP according to formula (2) and the MCUM
17: $|s|_{unused} \longleftarrow |s|_{unused} - |s|_{used}$
18: **end while**

In Algorithm 2, we first initialize some parameters before traversing MCUM, such as Π (an extended path) = ϕ, Φ (the set of extended paths) = ϕ and $|s|_{unused}$ (the number of the unused states) = $|s|_{all}$ (number of all states). After that, we traverse the MCUM from the start state (s_{Start}) of the MCUM and add the start state to Π (Lines 3 and 4). When the next state (s_{next}) is not the end state (s_{End}) (Line 5), we repeatedly construct a sequence $\{H_j\}$ (j = 1, 2, $\cdots$, n), generate the random number R_i ($0 < R_i < 1.0$), and judge whether R_i falls into the interval $[H_{j-1}, H_j]$ (Lines 6 to 8). If R_i falls into the interval $[H_{j-1}, H_j]$, we select s_{next}, the state with the edge corresponding to p_{ij} as the exit edge of state i, and add it to the extended path Π while incrementing s_{next} by one; otherwise, we regenerate the random number R_i and repeat the previous steps (Lines 9 to 13). Note that $\{H_j\}$ is constructed by Formula (3) for arbitrary state i and $|s|_{used}$ is the number of the used states.

$$H_j = \begin{cases} 0 & \text{if } j = 0 \\ \sum_{k=1}^{j} p_{ik} & \text{if } j = 1, 2, \cdots, n \end{cases} \tag{3}$$

After obtaining Π, we judge whether there are some remaining states (Line 2). If so, we add Π to Φ and set $\Pi = \phi$, and then update the STP according to Formula (2) and the MCUM. These steps are repeated until the number of remaining states is zero, when the algorithm exits with the output Φ.

To better implement Algorithm 2, we transform MCUM into P_{matrix} (Definition 3) in this paper. Note that traversing MCUM is equivalent to the traversing state P_{matrix}, so we transform the problem of solving MCUM into the problem of solving P_{matrix}.

We further improve the effectiveness of generating extended paths in two aspects, including controlling the number of extended paths and the redundancy of extended paths. For this purpose, we add an indicator $r_{coverage}$ and an empty matrix

TABLE III
THE THREE EXTENDED PATHS AFTER REDUCTION

S^{ep}	Representations of S^{ep}
S_1^{ep}	$<Start \rightarrow A \rightarrow 1 \rightarrow 2 \rightarrow 4 \rightarrow B \rightarrow End>$
S_2^{ep}	$<Start \rightarrow A \rightarrow 1 \rightarrow 3 \rightarrow 4 \rightarrow B \rightarrow End>$
S_3^{ep}	$<Start \rightarrow A \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow C \rightarrow End>$

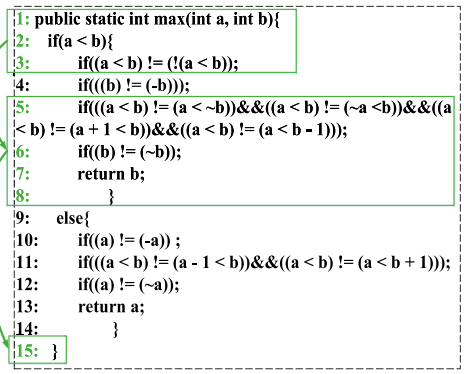


Fig. 13. The executed path corresponding to the S_1^{ep}.

A_ϕ during the generation of extended paths. $r_{coverage}$ refers to the coverage of states (nodes), which is calculated using Formula (4).

$$r_{coverage} = \frac{N_{coverage}}{N} \times 100\%, \tag{4}$$

where $N_{coverage}$ is the number of covering states (nodes) in the MCUM, and N is the number of all states (nodes) on the MCUM.

Based on this, we give a specific method to implement the process. First, we add $r_{coverage}$ in the process of programming. If $r_{coverage}$ is 100%, the number of paths will be output. Otherwise, the path generation cycle will continue until $r_{coverage}$ reaches 100%. After that, we add an empty matrix A_ϕ to store and reduce all the generated paths. Note that we still use $r_{coverage}$ to reduce the redundancy of the generated extended paths in the process of reduction.

In the following, we illustrate the process of generating extended paths based on P_{matrix} for Example 1.

1) Initialize some parameters, such as $\Phi = \phi$ and $|s|_{all}$ is 12.
2) For P_{matrix} corresponding to the MCUM (Fig. 10) of the extended program P^{new} for Example 1, we obtain five extended paths (as shown in Table II) based on Algorithm 2.
3) Further, we store five extended paths ($S_i^{ep} \in S^{ep}$ and $i = 1, \cdots, 5$) from Table II in an empty matrix A_ϕ and reduce them based on $r_{coverage}$ (Formula (4)). After reduction, we finally get three extended paths as shown in Table III.
4) Based on the three extended paths (Table III), we easily obtain the executed paths corresponding to them in P^{new}. Here we use the extended path S_1^{ep} for illustration and show the executed path corresponding to S_1^{ep} (the green numbers in Fig. 13).

C. Test Data Generation for Covering Extended Paths Based on EDA

After obtaining the set of extended paths, we transform the problem of generating test data for mutation testing into the problem for the coverage of the extended paths. To accurately and quickly generate test data for covering extended paths, we use a classical EDA, PBIL [20], [25], to generate test data in this paper. Compared with other methods [26], [27], PBIL uses the probability of knowledge learning through evolution to guide generation, which makes the search more directional to avoid falling into local optimum, and thus obtain better results. In addition, PBIL is also simple and easy to implement, and frequently used in various application fields [20], [25].

In view of this, we propose a method based on EDA to automatically generate test data of covering extended paths, mainly including pretreatment and the method of generating test data for covering extended paths based on EDA.

1) Preliminary Preparations: PBIL is a classic EDA, which requires some preliminary preparations before being used for test data generation, including the initialization of population, the individual encoding and decoding, the construction of the fitness function and the updating of the probability vector.

Population Initialization. Different from GA, we use the sampling probability vector to generate the population for EDA. Let $g = 0$ and $p_i^{(g)} = 0.5$ ($i = 1, 2, \cdots, l$), where g is the value of generation, l is the length of genes and each individual consists of l genes. Based on this, the sampling probability vector $P^{(g)}$ is initialized as $P^{(0)} = (0.5, 0.5, \cdots, 0.5)$ and the initial population is obtained based on $P^{(0)}$.

Individual encoding and decoding. Individual encoding is the process of transforming parameters or solutions in the search space into individuals (such as binary string) in the genetic space, while individual decoding is the opposite process to individual coding. In this paper, we adopt different encoding and decoding methods based on the type of program input (such as binary, integer, real, and character) before giving test data generation. For binary number types, we will use binary encoding directly and convert the binary to a decimal integer for calculating the branch distance. For integer and real number types, we use binary encoded individual and decode through Formula (5) to obtain data in the input field [29] for calculating the branch distance.

Given any individual X (a test datum), if the interval of variable X_i from X is $[a_i, b_i]$ and the encoding of X_i is $m_L m_{L-1} \cdots m_2 m_1$ (L is the encoding length), the decoding of X_i is as follows:

$$X_i = a_i + \left(\sum_{j=1}^{L} m_j \times 2^{j-1} \right) \times \frac{b_i - a_i}{2^L - 1} \tag{5}$$

Note that whether to perform rounding operations depends on the type of variable X_i. If it is an integer type, it is necessary to perform rounding operations, otherwise it is not necessary to perform rounding operations.

For Example 1, we want to get a test datum X from the input field [-10, 10]×[-10, 10] for program P (Fig. 1(a)).

TABLE IV
EXPRESSIONS OF SIMPLE PREDICATE BRANCH FUNCTIONS [8]

pr_{simple}	F_{simple}	r_{el}
$\varepsilon_1 > \varepsilon_2$	$\varepsilon_2 - \varepsilon_1$	$<$
$\varepsilon_1 \geq \varepsilon_2$	$\varepsilon_2 - \varepsilon_1$	$\leq$
$\varepsilon_1 < \varepsilon_2$	$\varepsilon_1 - \varepsilon_2$	$<$
$\varepsilon_1 \leq \varepsilon_2$	$\varepsilon_1 - \varepsilon_2$	$\leq$
$\varepsilon_1 == \varepsilon_2$	$abs(\varepsilon_1 - \varepsilon_2)$	$==$
$\varepsilon_1 ! = \varepsilon_2$	$-abs(\varepsilon_1 - \varepsilon_2)$	$<$

Notes:: pr_{simple} is a simple predicate expression, where ε_1 and ε_2 are arithmetic variables, $\{<, \leq, >, \geq, ==, ! =\}$ is a set of simple predicates, F_{simple} is a simple branch function, the equivalent expression $F_{simple} r_{el} 0$, where $r_{el} \in \{<, \leq, ==\}$ is the equivalent predicate after transformation.

TABLE V
EXPRESSIONS OF BRANCH FUNCTIONS FOR COMPLEX PREDICATES

$pr_{complex}$	$Examples$	$F_{complex}$
&& (Only)	$pr^1_{simple} \&\& \cdots pr^n_{simple}$	$max(F^1_{simple}, \cdots, F^n_{simple})$
$\|\|$ (Only)	$pr^1_{simple} \| \| \cdots pr^n_{simple}$	$min(F^1_{simple}, \cdots, F^n_{simple})$
&& and $\|\|$	$pr^1_{simple} \&\& \cdots \| \| pr^n_{simple}$	$min(max(F^1_{simple}, F^2_{simple}), \cdots, F^n_{simple})$

Notes: $pr_{complex}$ is a complex predicate, $pr^i_{simple} (i = 1, 2, \cdots, n)$ is a simple predicate arithmetic expressions, $F_{complex}$ is a complex branch function, $F^i_{simple} (i = 1, 2, \cdots, n)$ is a simple branch function.

TABLE VI
THE RESULTS OF DESIGNING MBFS FOR TYPE I

Mutation Operators	Descriptive Mutation Operator	Mutant Branch Function (F)
AOIU	Arithmetic Operator Insertion Unary	$-abs(v_1(-v_1))$
AODU	Arithmetic Operator Deletion Unary	$-abs(-v_1(v_1))$
AOIS	Arithmetic Operator Insertion Short-Cut	$-abs(v_1 - v_2)$
AODS	Arithmetic Operator Deletion Short-Cut	$-abs(v_1 - v_2))$
AORS	Arithmetic Operator Replacement Short-Cut	$-abs(v_1 - v_2))$
LOI	Logical Operator Insertion	$-abs(v_1 - v_2)$ and $v_2 = -(v_1 + 1)$
LOD	Logical Operator Deletion	$-abs(v_2 - v_1)$ and $v_1 = -(v_2 + 1)$

Notes: v_1 is variable, constant or array element, and v_2 is the result of mutation for v_1

First, we obtain the 8-bit binary encoding of the two variables of X is (10000000, 00000000). Then, we can get the test datum X based on Formula (5), which is (0, -10) (After rounding operations).

For character types, we refer to the method in prior literature [30], which first maps each character in a string to its corresponding $ASCII$ code, and then represents the $ASCII$ code corresponding to each character using binary encoding. After that, we convert the binary to a decimal integer to calculate the branch distance. Note that we obtain the corresponding string based on $ASCII$ decoding.

Construction of Fitness Function. Fitness function is an important part of EDA. To generate test data of meeting certain criteria (such as path coverage), researchers have proposed many methods for constructing the fitness functions, such as Korel's method [31] and Zhang's method [32]. Different from them, this paper needs to build the mutant branch functions to generate test data of covering the extended paths, which undoubtedly increases the difficulty of constructing the fitness function. Thus, we develop a new method for constructing the fitness function, MCFF, the specific steps of which are elaborated below.

MCFF is mainly implemented by constructing a series of branch functions, inserting the constructed branch functions into the reduced program P_{after} (Fig. 6) for the extended program P^{new} and constructing the fitness function based on the constructed branch functions.

First, we give a method of constructing a series of branch functions, including both the non-mutant branch functions (NMBFs) and the mutant branch functions (MBFs). The specific steps are as follows:

- For NMBFs, we build the branch functions through Table IV [8] (or Table V) and show the equivalent expression of the NMBF F(F_{simple} or $F_{complex}$) in Equation (6). For example, we express the NMBF of the statement $if(a < b)$ in Line 2 from Fig. 13 based on Table IV as $F(a, b)$ = a - b and regard the equivalent form of $if(a < b)$ as $if(F(a, b) < 0)$.

$$F = \begin{cases} 0 & \text{if } r_{el} \text{ is true} \\ |F| & \text{otherwise} \end{cases}, \tag{6}$$

where $|F|$ is the absolute value of F ($|F| \neq 0$).

- For MBFs, we give a method of constructing them based on *accessibility* and *necessity* [14]. On the one hand, we directly refer to Table IV and Table V for *accessibility*. On the other hand, we design the MBFs from Type I and Type II for *necessity*. For Type I (AOIU, AODU, AOIS, AODS, AOIS, LOI and LOD), we obtain the MBF from Table VI. For example, the mutant branch statement $if(b$!= $\sim b)$ ($\sim$ is LOI) in Line 6 from Fig. 13 based on Table VI can be expressed as $F(a, b)$ = -|a - b| (b = -(a + 1)) and the equivalent form of $if(b$!= $\sim b)$ is $if(F(a, b) < 0)$. For Type II (AORB, ASRS, ROR, COI, COD and COR), we construct the MBF from Table VII and Table V [8] [32]. For example, the statement in Line 11 from Fig. 13 is expressed as $F(a, b)$ = $max(F_1(a, b), F_2(a, b))$ with Table VII, where $F_1(a, b)$ = -|a - 1 - b| and $F_2(a, b)$ = -|a - (b + 1)|. After that, the equivalent form of statement in Line 11 from Fig. 13 is $if(F(a, b)$ == 0). Note that we regard the equivalent expression of $if((a < b)$!=$(a - 1 < b))$ as $if((a$ - 1) == $b)$ for *necessity* [8] and show the equivalent expression of the MBF F in Equation (6).

Then, we insert the constructed branch functions into P_{after} (Fig. 6) for P^{new}. If an extended path randomly selected from all extended paths generated based on MCUM has m branch (non-mutant branch or mutant branch) statements, then we insert m branch functions (F_1, F_2, $\cdots$, F_m) in front of m branch statements from P_{after} and readers refer to the prior literature [31] for more detailed details.

In the end, we construct the fitness function based on the inserted NMBFs and MBFs and give the formula (Formula (7)) of the fitness function as follows.

$$Fit = 1 - e^{-\sum_{i=1}^{m} F_i}, \tag{7}$$

TABLE VII
THE RESULTS OF DESIGNING MBFS FOR TYPE II

Mutation Types - Mutation Operators	Descriptive Mutation Operator	Mutant Branch Function (F)
Arithmetic Operator Replacement Binary-AORB	$\varepsilon_1 r_{op} \varepsilon_2$	$-abs((\varepsilon_1 r_{op} \varepsilon_2) - (\varepsilon_1 r^1_{op} \varepsilon_2))$
Assignment Operator Replacement-ASRS	$\varepsilon_1 r_{op} = \varepsilon_2$	$-abs((\varepsilon_1 r_{op} \varepsilon_2) - (\varepsilon_1 r^1_{op} \varepsilon_2))$
Relational Operator Replacement-ROR	$<\Longrightarrow>$	$\varepsilon_2 - \varepsilon_1$
	$<\Longrightarrow\geq$	$\varepsilon_2 - \varepsilon_1$
	$<\Longrightarrow==$	$\varepsilon_2 - \varepsilon_1$
	$<\Longrightarrow\leq$	$abs(\varepsilon_1 - \varepsilon_2)$
	$<\Longrightarrow!=$	$\varepsilon_2 - \varepsilon_1$
	...	...
Conditional Operator Insertion-COI	$E_1 \Longrightarrow E_2$	F_{E_1}
Conditional Operator Deletion-COD	$E_1 \Longrightarrow E_2$	F_{E_2}
Conditional Operator Replacement-COR	$E_1 \Longrightarrow E_2$	-abs($F_{E_1} - F_{E_2}$)

Notes: ε_1 and ε_2 is variable, constant, or array element, r_{op} and r^1_{op} (r_{op} and $r^1_{op} \in \{+, -, *, \div, \%\}$) are original binary operation and the binary operation after mutation, $\Longrightarrow$ is mutation, E_i ($i = 1, 2$) represents a variable or relational expressions, F_{E_i} represents branch function of E_i ($i = 1, 2$)

where F_i (i = 1, 2, $\cdots$, m) is the i-th branch function (NMBF or MBF) and $Fit \in$[0,1) is used to evaluate the extent to which the test data covers the extended path. The smaller the value of Fit, the more nodes in the target extended path will be covered by the generated test data (more mutants will be killed). In particular, when Fit = 0, the generated test data completely covers the target extended path, which means that we have generated test data covering the target extended path.

Update Probability Vector. Different from GA, EDA needs to constantly update the probability vector $P^{(g)}$ (g is the value of generation) to obtain the offspring. Assuming that the initial population is N, l is the length of individual genes, M ($M < N$) best excellent individuals are selected from N individuals, denoted as X_{k_1}, X_{k_2}, $\cdots$, X_{k_M}. $P^{(g)}$ is updated by Formula (8).

$$p_i^{(g+1)} = (1 - \alpha) \cdot p_i^{(g)} + \alpha \cdot \frac{1}{M} \cdot \sum_{j=1}^{M} X^i_{k_j}, \tag{8}$$

where i = 1, 2, $\cdots$, l and $\alpha \in (0, 1]$ is the learning rate.

Because α not only affects the convergence speed of PBIL, but also affects the search ability of the global optimal solution of PBIL. Therefore, we give Formula (9) to determine the size of α [33].

$$\alpha = r_0 \times \frac{[t/t_0]}{N_{max}} + r_1 \times (-1)^{Tmod10}, \tag{9}$$

where $a_1 = r_0 \times \frac{[t/t_0]}{N_{max}}$ is a learning factor that changes with evolutionary algebras, $a_2 = r_1 \times (-1)^{Tmod10}$ is a learning factor that changes with the degree of evolution, r_0 and r_1 are the benchmarks for a_1 and a_2, N_{max} is the maximum iteration number of the algorithm, t is the evolutionary algebra, and t_0 is the interval algebra of evolution, T is an algebra in which the population optimal individuals are continuously not improved, mod is a modular operation, and $[t/t_0]$ is the largest integer not greater than t/t_0.

2) Generating Test Data for Covering Extended Paths based on EDA: After the above preparations, we propose a method, namely TDG-EDA, to generate test data for covering extended paths. The specific steps of TDG-EDA are shown in Algorithm 3.

Algorithm 3 TDG-EDA

Input: The set Φ of extended paths
Output: The set Ψ of test data
1: Set g = 0 , T_0 = 20 and $|\Phi|_{unused} = |\Phi|$
2: Initialize the probability vector $P^{(g)}$ and the population size N_{size}
3: **while** $|\Phi|_{unused} \neq 0$ **do**
4: Select an extended path from Φ without repetition
5: **while** $g < N_{max}$ **do**
6: Get the g-th generation population from $P^{(g)}$
7: Obtain test data (individuals) by the decoding method
8: Calculate the fitness Fit from formula (7)
9: **if** Fit == 0 **then**
10: Add the optimal solution (test data) to Ψ
11: **else**
12: Obtain stable algebra T based on Fit
13: **if** $T == T_0$ **then**
14: Break
15: **else**
16: Get N_{best} ($N_{best} < N_{size}$) optimal solutions
17: Calculate α according to Formula (9)
18: Update $P^{(g)}$ from Formula (8)
19: $g \longleftarrow g + 1$
20: **end if**
21: **end if**
22: **end while**
23: $|\Phi|_{unused} \longleftarrow |\Phi|_{unused}$ - $|\Phi|_{used}$
24: **end while**

In Algorithm 3, we first initialize some parameters, such as the probability vector $P^{(g)}$ is (0.5, $\cdots$, 0.5) when the generation g is 0, T_0 = 20, the population size N_{size} is 100 and $|\Phi|_{unused}$ (the number of unused extended paths) is $|\Phi|$ (Lines 1 and 2). After that, we choose an extended path from Φ without repetition (Line 4) when $|\Phi|_{unused} \neq 0$. Furthermore, we obtain the g - th generation population by $P^{(g)}$ and get test data (individuals) in input field by the decoding method (Lines 5∼7) when $g < N_{max}$ (the maximum number of iterations) (Line 5). After that, we calculate the fitness Fit by formula (7) and judge whether Fit == 0 is satisfied. If Fit == 0, we add the test data to Ψ (Lines 6∼10). Otherwise, we obtain a stable algebra T based on Fit and determine whether the threshold T_0 is reached. If so, we exit the cycle. Otherwise, we calculate α according to Formula (9) and update $P^{(g)}$ based on Formula (8) after selecting N_{best} ($N_{best} < N_{size}$) excellent individuals, and then repeat steps in Lines 6∼8 to calculate Fit until Fit == 0, add the test data to Ψ; otherwise, repeat steps in Lines 6∼21 until $g \geq N_{max}$, add the test data to Ψ and go to Line 3. After that, we also judge whether $|\Phi|_{unused} \neq 0$ (Line 3). If $|\Phi|_{unused} \neq 0$, we re-select an extended path from Φ and repeat steps in Lines 5∼22. Otherwise, we exit the loop to output the set Ψ of test data until all extended paths are used (Lines 3∼24). Note that $|\Phi|_{used}$ is the number of the used extended paths (Line 23).

In the following, we illustrate the process of generating rest data for covering extended paths based on TDG-EDA for Example 1.

1) For the program P (Fig. 1(a)) from Example 1, we obtain the extended program P^{new} (Fig. 4) through transformation for weak mutation testing. After that, we obtain the reduced program P_{after} (Fig. 6) by COMR and DR in Section III.A.2 and construct the MCUM (as shown in Fig. 10) of P_{after} by **Algorithm 1** (Section III-A).
2) According to the MCUM (Fig. 10) of P_{after} from Step (1), we obtain three EPs (S_1^{ep}, S_2^{ep} and S_3^{ep}) in Table III from **Algorithm 2** (Section III.B.2). Further, we find the three actual executed paths corresponding to the three EPs in P_{after}. Here we only give the actual path (the green numbers in Fig. 13) of S_1^{ep} in P_{after} (S_2^{ep} and S_3^{ep} are similar to S_1^{ep}).
3) Based on three EPs in Table III from Step (2), we construct the fitness function (Fit) corresponding to each EP according to MCFF (Section III.C.1) and Formula (7). For example, we get the Fit of S_1^{ep} by MCFF and the specific steps are as follows: 1) Construct a series of branch functions (NMBFs or MBFs) (the blue code in ① and ② from Fig. 14) for S_1^{ep} from Tables IV∼VII; 2) Insert the constructed branch functions in front of the corresponding statements in P_{after} (the blue code in Fig. 14) and ③ in Fig. 14 is the equivalent form of Formula (6); 3) Obtain the Fit (as shown in Formula (10)) of S_1^{ep} by Formula (7). S_2^{ep} and S_3^{ep} are similar to S_1^{ep}, so we won't explain them here.

$$Fit = 1 - e^{-\sum_{i=1}^{2} F_i} = 1 - e^{-F(a,b)} \tag{10}$$

4) After giving the fitness functions of the three EPs, we use **Algorithm 3** to generate test data covering the three EPs from input field ($[-10, 10] \times [-10, 10]$). However, here we must initialize some parameters for **Algorithm 3**, such as $P^{(g)}$ is (0.5, $\cdots$, 0.5) when the generation g is 0, T_0 = 20, N_{size} = 4, the individual length L is 8, the number of integer variables (a, $b \in [-10, 10]$) is 2, r_0 (r_1) is 0.5, N_{max} (the maximum number of iterations) is 10, and N_{best} (excellent individuals) is 2. After that, we execute Steps (5)∼(8) from **Algorithm 3** to generate test data.
5) We choose an extended path S_i^{ep} (i = 1, 2, 3) of the three EPs without repetition. After that, we get the initial population (N_{size} = 4) from $P^{(0)}$ and obtain 4 test data ({(-2, 2), (0, -6), (3, -2), (9, 5)}) for S_i^{ep} in input field ($[-10, 10] \times [-10, 10]$) by binary encoding and decoding [20], [25]. Note that each datum (a, b) is composed of integer variables a and b.
6) When 4 test data from Step (5) execute the program P_{insert} from S_i^{ep} (i = 1, 2, 3), we calculate Fit by Formula (10). In this way, we transform the problem of test data generation into the problem of finding the solution of the minimum fitness. Here we only give the P_{insert} (Fig. 14) of S_1^{ep} in P_{after}.
7) Determine whether the 4 test data can cover S_i^{ep} (i = 1, 2, 3) according to the size of Fit. If Fit is 0, we output the test data covering S_i^{ep}; otherwise, we update $P^{(g)}$ with Formula (8), and then repeat steps (5) and (6) to calculate Fit until Fit is 0 or N_{max} is 10, output the corresponding test data to exit the cycle. For example, we obtain 4 test data ({(8, 2), (-1, 0), (-2, 5), (-1, -1)}) of the 3-th generation and calculate the 4 Fits after executing the program for S_1^{ep}. Only the value of Fit for the datum $(-1, 0)$ is 0, so $(-1, 0)$ is the test datum covering S_1^{ep}.
8) Similar to S_1^{ep}, we obtain test data $(-3, 8)$ and $(-6, -6)$ covering S_2^{ep} and S_3^{ep} based on Steps (5)∼ (7). In this way, we get a set of test data T = $\{t_1, t_2, t_3\}$ covering three EPs (S_1^{ep}, S_2^{ep} and S_3^{ep}), where t_1 = $(-1, 0)$, t_2 = $(-3, 8)$ and t_3 = $(-6, -6)$.

```
1: public static int max(int a, int b){
     int F1(a,b)=a-b;
2:   if(a < b){                                                    ①
       int F2(a,b)=a-b;
3:     if((a < b) != (!(a < b)));
4:     if((b) != (-b)));
       int F3(a,b)=Math.max(Math.abs(a+1-b),
       Math.abs(b));
5:     if(((a < b) != (a < -b))&&((a < b) != (-a <b))&&
       ((a < b) != (a + 1 < b))&&((a < b) != (a < b - 1)));
       int F4(a,b)=-Math.abs(a-(-(a+1));//b=-(a+1)                 ②
6:     if((b) != (-b));
7:     return b;
8:       }
9:   else{
10:    if((a) != (-a)) ;
11:    if(((a < b) != (a - 1 < b))&&((a < b) != (a < b + 1)));
12:    if((a) != (-a));
13:    return a;
14:      }
     if (F1(a,b)<0){
       F1(a,b) =0;
      }else{
       F1(a,b) =Math.abs(F1(a,b));
      }
     if (F2(a,b)<0){
       F2(a,b) =0;
      }else{
       F2(a,b) =Math.abs(F2(a,b));
      }
     if (F3(a,b)==0){
       F3(a,b) =0;                                               ③
      }else{
       F3(a,b) =Math.abs(F3(a,b));
      }
     if (F4(a,b)<0){
       F4(a,b) =0;
      }else{
       F4(a,b) =Math.abs(F4(a,b));
      }
       int F(a,b) =F1(a,b)+F2(a,b)+
       F3(a,b)+F4(a,b);
       return F(a,b);
15: }
```

Fig. 14. The program P_{insert} of inserting branch (non-mutant branch and mutant branch) functions of S_1^{ep}.

IV. Experimental Studies

We conduct a series of experiments to evaluate the effectiveness and applicability of TAMMEA. The experimental settings are presented in the following.

A. Research Questions

We design our experiments to answer the following three research questions:

- **RQ1:** How effective are the test data generated by TAMMEA in killing mutants under strong mutation criterion? As presented above, TAMMEA mainly leverages the concepts of weak mutation testing in test data generation. It is

thus necessary to examine whether the test data generated by TAMMEA are also effective under the strong mutation testing.

- **RQ2:** Can TAMMEA improve the efficiency of test data generation for mutation testing?
  In addition to a high fault-detection effectiveness, a testing method also needs to deliver a high efficiency for guaranteeing its practicality. In our experiments, we evaluate the efficiency of TAMMEA based on different metrics, including the number of test data/paths/iterations required for killing most mutants. For a method, if it involves fewer test data/paths/iterations in the process of generating test data, its efficiency will be higher.
- **RQ3:** Can TAMMEA significantly reduce the time cost without jeopardizing the effectiveness of mutation testing?
  The overall testing time is the most straightforward measure for the cost-effectiveness of a testing technique. In this study, we examine the execution time that TAMMEA spends in killing mutants.

B. Variables and Measures

1) Independent Variables: The independent variable in our study is the test data generation method. By nature, we choose the proposed TAMMEA framework in the study. In addition, we select eight baseline techniques, namely Random, Randoop [34], CATG [35], GA [8], HGA [36], HC [37], GFC [38] and FUZGENMUT [16] as the baselines for comparing with our approach.

Random: Random is to randomly sample the input space to generate a large number of test data for killing non equivalent mutants. Random is often used as the most fundamental comparison method in previous studies [39], [40].

Furthermore, we also compared our method with two widely used test case generation tools (Randoop [34] and CATG [41]) by researchers.

Randoop: Randoop [34] is a tool of automatically generating test data for Java programs based on feedback directed random testing. Unlike Random, Randoop[3] improves random test generation by combining feedback obtained from executing test inputs when creating test inputs, which can effectively avoid randomness and increase code coverage.

CATG: CATG [35] is an open-source symbolic execution-based test tool for Java, which converts test inputs and initial databases generated by Java programs into test cases for the tested application. Note that CATG[4] introduces a novel annotation mechanism to enable programmers to trim the search space for consistency testing.

Different from the previous three methods, the other five baseline techniques use evolutionary algorithms to improve the efficiency and accuracy of test data generation.

GA: GA [8] is a test data generation method based on genetic algorithm for reducing the cost of mutation testing and the main steps of this method are as follows: (1) Construct the branch functions based on the *accessibility* and *necessity*; (2) Insert the branch functions into the source program; (3) Use a path oriented GA to obtain test cases for killing mutants.

HGA: HGA [36] is proposed for generating test data automatically using data flow testing approach for mutation testing and the specific steps are as follows: (1) Find all paths corresponding to the mutant based CFG; (2) Construct fitness based on the mutation score in the mutation testing; (3) GA is used to generate test data.

GFC: GFC [38] is a method of generating test cases based on data flow constraints and the specific steps are as follows: (1) The fitness function is modeled by combining control flow constraints and data flow constraints; (2) The fitness function is applied to GA to guide the evolution and selection of test cases.

HC: HC [37] is an automated test generation approach based on hill climbing for strong mutation testing and it uses the AUSTIN [42] tool to generate test data and is only intended to cover program statements or branches by using a hill climbing (HC) technique called alternate variable method (AVM).

FUZGENMUT: FUZGENMUT [16] makes use of fuzzy clustering and multi-population genetic algorithm (MGA) to improve the effectiveness and efficiency of mutation testing from two perspectives. The main steps of FUZGENMUT are as follows: (1) Use fuzzy clustering to classify mutants into different clusters; (2) Construct a fitness function and transform the test data generation problem into an optimization problem; (3) Apply MGA to generate test data to kill mutants in different clusters in parallel.

The main reason behind the selection of these five baseline techniques (GA, HGA, HC, GFC and FUZGENMUT) is that the basic steps of them are quite similar to those of TAMMEA. In addition, evolutionary algorithms (such as GA) has been used for generating test data in mutation testing [35], [43].

2) Dependent Variables: The dependent variables in this study are mainly the metrics to be used for measuring the performance of the techniques under study.

For RQ1, we use the mutation score (MS), a fundamental metric in mutation testing, which is calculated by the Formula (1) given in Section II. MS has been widely used as the metric for measuring the fault-detection effectiveness (reflected as the mutant-killing capability).

For RQ2, we select three metrics for the evaluation of testing efficiency, including (i) the number of test data generated by each of all techniques, (ii) the number of paths and the number of iterations required by each of six techniques (such as GA, HGA, HC, GFC, FUZGENMUT and TAMMEA). Intuitively speaking, a testing method (such as Random, Randoop, CATG, GA, HGA, HC, GFC, FUZGENMUT and TAMMEA) has a higher efficiency if it can use less test data. In addition, an evolutionary algorithm based method (such as GA, HGA, HC, GFC, FUZGENMUT and TAMMEA) has a higher efficiency if it can achieve its goal (e.g., generating test data) with a smaller number of iterations and/or require the coverage of fewer paths to kill the similar number of mutants.

For RQ3, we measure the overall execution time each technique required in test data generation. The shorter the execution time, the better the method.

[3]https://randoop.github.io/randoop/
[4]https://github.com/ksen007/janala2

TABLE VIII
SUBJECT PROGRAMS AND MUTANTS

ID	Program	LOCs	Methods	Mutants		Description
				Total	Equivalent	
J1	TrashAndTakeOut	30	2	111	29	Basic arithmetic operations
J2	Mid	26	1	115	18	Return the middle value of three integers
J3	FourBalls	28	1	213	49	Relative weight of four spheres
J4	Triangle	36	1	325	40	Return the type of a triangle with three integer inputs
J5	Cal	46	2	314	43	Calculate the days between two dates in the same year
J6	HelpFormatter	416	39	291	42	A formatter of help messages for current command line options
J7	WordUtils	173	12	243	34	Operations on Strings that contain words
J8	NumberUtils	636	47	1406	212	Provides extra functionality for Java Number classes
J9	Dfp	1702	113	2133	258	Decimal floating point library for Java
J10	FastMath	2311	100	6486	976	Faster, more accurate, portable alternative to Math and StrictMath for large scale computation
J11	ArrayUtils	3258	319	7551	795	Operations on arrays, primitive arrays (like int[]) and primitive wrapper arrays
J12	Jdkmath	11540	173	31576	3265	Alternative to Math and StrictMath
J13	Numbers	20205	340	64955	5023	Number types (complex, quaternion, fraction) and utilities (arrays, combinatorics)
J14	Colt	31407	3106	93985	7749	Project for high performance scientific and technical computing
J15	Lang	53261	2106	132197	9722	Provides extra functionality for classes in java.lang
	Sum.	125075	6362	341901	28255	

C. Subject Programs and Generation of Mutants

We select 15 Java programs that vary in size and come from different application areas as the subjects of our experiments. Further, the test inputs for the 15 benchmarks are different and the variable types of the test inputs mainly include binary, integer, real, and string types. Table VIII gives the basic information of 15 Java programs. Among them, five (J1 to J5) are small-sized programs that implement basic scientific calculations [44], [45] and the other ten programs (J6 to J15) are some popular utilities [12], which can be downloaded from http://commons.apache.org.

We apply $MuClispe$ [10] [11], a commonly used mutation testing tool, to generate mutants for all 15 Java programs. Note that we only focus on method-level mutation operators during the process of generating mutants in the article. Although there are 15 method-level mutations operators in $MuClispe$, only 13 operators are used in this paper, because two operators are not related to branch functions. The information about mutant generation is given in columns 5 to 6 of Table VIII.

D. Experimental Procedure

We use $MuClipse1.3$ (Eclipse SDK 4.2.2) and $MATLAB$ $(2012a)$ to conduct our experiments on a machine with Intel(R) Core(TM) i5-3230M CPU @ 2.60GHz, 4.00GB ROM, Windows 8 Operating System.

As mentioned above, we use $MuClipse$ [10] to generate all mutants and determine whether a mutant is killed. In addition, it is a common criterion in mutation testing that the source program serves as the test oracle for verifying whether a test case kills a mutant. In other words, we determine whether a mutant is killed by comparing the output of the source program (the test oracle) with that of the mutant.

Considering that the equivalent mutants have no significance in generating test data, we remove them from all mutants before conducting the experiment. For this purpose, we use a semi-automatic method proposed by Yao et al. [15] and the specific steps are as follows: 1) $MuClipse$ is used to run all mutants on the test suite generated via Randoop [34] (which is now a plugin of $MuClipse$) and live mutants (that is, the mutants that are still alive after executing all test data) are obtained; 2) Each of these live mutants is then manually checked to determine whether they are equivalent to the base program. During the process of determining equivalent mutants, Randoop generates a large number of test data (larger than the number of test data generated by Randoop used in experiment for RQ1). So, the number of live mutants (for determining equivalent mutants) is very small. In this article, Randoop generates 5000 test data for J1∼J7, 15000 test data for J8∼J13 and 25000 test data for J14∼J15. After that, we can obtain that the live mutants for each program as shown in Table IX. From Table IX, we obtain that the number of live mutants (as shown in column 4 of Table IX) obtained by each program after using Randoop is very small, which means that the effort of manual analysis is very low when we use the method proposed by Yao et al. [46] to quickly manually analyze equivalent mutants (as shown in column 5 of Table IX).

TABLE IX
LIVE MUTANTS AND EQUIVALENT MUTANTS AFTER USING RANDOOP

ID	Total	Test Data	Live	Equivalent	Live/Total (%)	Equivalent/Total (%)
J1	111	5000	29	29	26.13	26.13
J2	115	5000	18	18	15.65	15.65
J3	213	5000	50	49	23.47	23.00
J4	325	5000	42	40	12.92	12.31
J5	314	5000	45	43	14.33	13.69
J6	291	5000	42	42	14.43	14.43
J7	243	5000	35	34	14.40	13.99
J8	1406	15000	217	212	15.43	15.08
J9	2133	15000	260	258	12.19	12.10
J10	6486	15000	980	976	15.11	15.05
J11	7551	15000	800	795	10.59	10.53
J12	31576	15000	3269	3265	10.35	10.34
J13	64955	15000	5027	5023	7.74	7.73
J14	93985	25000	7754	7749	8.25	8.24
J15	132197	25000	9727	9722	7.36	7.35
Sum.	341901	105000	28295	28255	8.28	8.26

According to the log file $mutation\text{-}log$ provided by $MuClipse$ and the weak mutation transformation rules, mutant branches are constructed for the generated mutants, and they are successively inserted into the source program P to obtain the new extended program P^{new}.

TABLE X
THE RANGE OF $Dif(M)$ UNDER EACH MUTANT CATEGORY

Mutant Category	The Range of $Dif(M)$
Easy	[0, 0.25]
Middle	(0.25, 0.5]
Difficult	(0.5, 0.75]
Very difficult	(0.75, 1]

TABLE XI
THE PARAMETERS USED BY TAMMEA AND OTHER FIVE TECHNIQUES

Parameters	Parameter Values	Parameter Description
$P^{(0)}$	(0.5, $\cdots$, 0.5)	Initial probability vector
Pop	100	Population size
p_m	0.95	Mutation probability
p_c	0.05	Crossover probability
r_0/r_1	0.5	Benchmarks of α
N_{best}	5	Number of excellent individuals
$N_{sub-size}$	5	Sub-population size
N_{size}	500	Maximum number of iterations

All nine techniques (Random, Randoop, CATG, GA, HGA, HC, GFC, FUZGENMUT and TAMMEA) are applied to 15 subject programs, and each program is tested by every technique for 20 times to get the average values of the experimental results. Furthermore, we conduct the statistical testing (via software SPSS) on the results to determine whether there is significant difference between TAMMEA and the other eight baseline techniques. In the process of answering RQ1, the number of test data generated by Random and Randoop is equal to TAMMEA, and the test data are randomly sampled 100 times in the input field to obtain an average mutation score. In addition, we also refer to the method proposed by Dang et al. [16] to classify all non-equivalent mutants for each tested program into four categories (easy, middle, difficult, and very difficult) for RQ1 and the specific steps are as follows: 1) Obtain all non-equivalent mutants and use Randoop to sample a certain number of test data (as shown in Table IX) in the input domain; 2) Classify all non-equivalent mutants based on $Dif(M)$ after executing all non-equivalent mutants, where $Dif(M)$ is the difficulty of killing mutant M and its value range under each mutant category is shown in the third column of Table X. Note that $Dif(M)$ = 1-$N_{kill}(M)$/N_{sample} ($0 \leq Dif(M) \leq 1$ and the larger $Dif(M)$, the more difficult M is to be killed), where $N_{kill}(M)$ is the number of test data that kill mutant M, N_{sample} is the number of test data sampled in the input domain ($N_{kill}(M) \leq N_{sample}$).

To ensure the smooth implementation of the experiment, Table XI provides additional parameter settings for TAMMEA and the five baseline techniques (GA, HGA, HC, GFC and FUZGENMUT) except for Random. In addition, we set the default values for all parameters of Randoop except for the maximum execution time (240 seconds in this paper) and use the default values for all parameters of CATG in our experiment.

E. Threats to Validity

1) Threat to Internal Validity: The main threat to internal validity is related to the implementation of TAMMEA, which requires a certain degree of programming work, e.g., the construction of MCUM in Section III. With the increasing number of mutants, TAMMEA will be affected by the process of constructing MCUM. In order to avoid the impact of this problem, we propose a semi-automatic method of constructing MCUM to reduce the impact of the process on TAMMEA. In addition, the source code for implementing all test data generation techniques has been cross-checked by different individuals to minimize the errors in programming.

2) Threat to External Validity: The major threat to external validity is concerned with the generalization of our results. Although we have selected Java programs with different sizes and from distinct fields in this study, our method handles only primitive types (include binary, integer, real, and string) for test data generation and we only apply method-level mutation operators in our experiments. Thus, we cannot generalize the obtained conclusions to class-level mutation operators and program inputs of non-primitive types. The mutant generation process is also limited to some selected methods, which may also affect the generality of the study.

3) Threat to Construct Validity: The threat to construct validity is related to the measurement. The metrics used in this study, such as MS, the number of test data, and execution time, are very straightforward and have been widely used in numerous studies.

4) Threat to Conclusion Validity: Due to the randomness of some settings (e.g., p_m), we implement each test data generation strategy for at least 20 times on every subject program to obtain the mean value of the experimental results. Since the collected data consistently show very small standard deviations, we are confident that the results are statistically reliable.

V. EXPERIMENTAL RESULTS

In the following, we present experimental results of TAMMEA, as compared with eight baseline methods (Random, Randoop, CATG, GA, HGA, HC, GFC and FUZGENMUT), and correspondingly, answer the three research questions raised in Section IV-A.

A. Answer to RQ1: Mutant-Killing Capabilities

Fig. 15 presents the effectiveness of the nine techniques under the strong mutation criterion, in terms of the mutation score MS_{strong}.

From Fig. 15, we could observe that TAMMEA (blue bar) was superior to the other eight baseline techniques (including 2 popular tools, namely Randoop and CATG) in their mutant-killing capabilities. Although TAMMEA mainly relied on weak mutation criterion to generate test data, TAMMEA could achieve $MS_{strong} > 96\%$ for all subject programs, whereas each of the other eight baseline techniques (Random, Randoop, CATG, GA, HGA, HC, GFC and FUZGENMUT) had at least two subject programs with $MS_{strong} < 96\%$. Furthermore, FUZGENMUT had a MS_{strong} of more than 96% in each of all subject programs except for J4, J6 and J7, and its performance was only second to TAMMEA on the whole, while

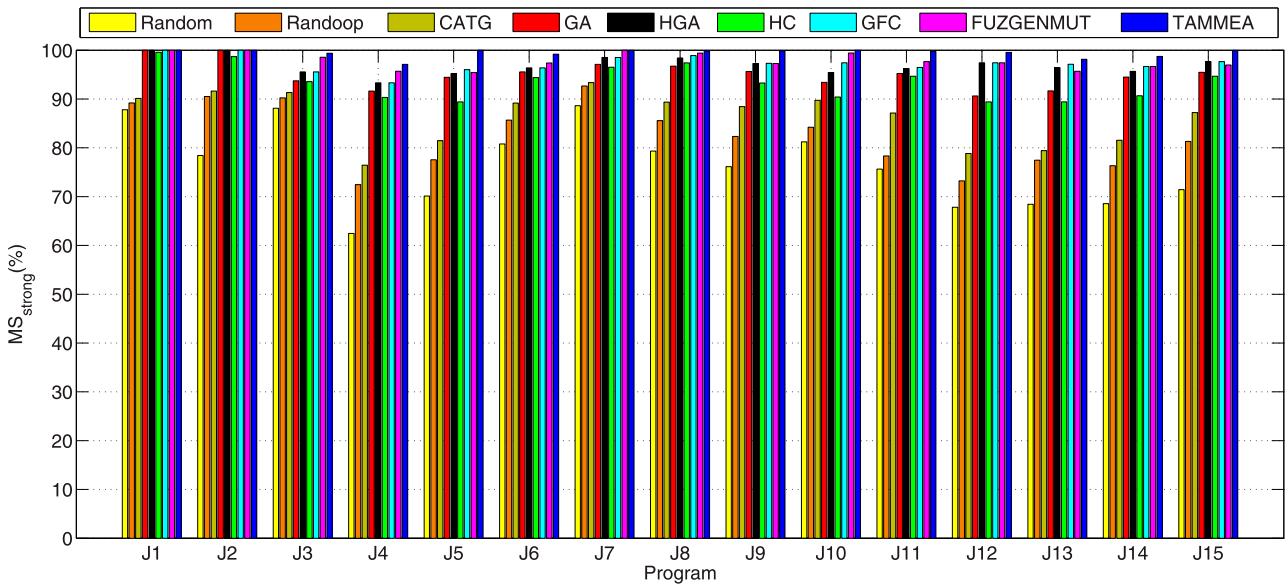


Fig. 15. The effectiveness of comparison methods under the strong mutation testing.

TABLE XII
STATISTICS OF MS_{strong} (%) FOR TAMMEA AND EIGHT BENCHMARK METHODS

Methods	Samples (No.)	Minimum	Maximum	Sum	Mean Value	Standard Deviation	Variance	Skewness	Kurtosis
Random	15	62.45	88.64	1144.91	76.33	8.15	66.42	0.10	-0.97
Randoop	15	72.45	92.67	1237.13	82.48	6.48	42.02	0.02	-1.21
CATG	15	76.45	93.37	1295.22	86.35	5.34	28.56	-0.64	-1.01
GA	15	90.62	100.00	1425.75	95.05	2.73	7.47	0.34	-0.05
HGA	15	93.33	100.00	1453.57	96.90	1.83	3.36	0.12	-0.09
HC	15	89.42	99.58	1402.41	93.49	3.48	12.09	0.33	-1.14
DFC	15	93.33	100.00	1458.78	97.25	1.72	2.94	-0.33	0.96
FUZGENMUT	15	95.42	100.00	1467.57	97.84	1.64	2.68	0.03	-1.35
TAMMEA	15	97.09	100.00	1491.53	99.44	0.85	0.72	-1.91	3.40

TABLE XIII
T-TEST RESULTS OF COMPARING TAMMEA AND EIGHT BENCHMARK METHODS

Comparison Methods	Mean Value	Standard Deviation	Standard Error	95% Confidence Interval		p-Value (Bilateral)
				Lower Limit	Upper Limit	
TAMMEA vs. Random	23.11	7.67	1.98	18.86	27.36	0.00
TAMMEA vs. Randoop	16.96	6.04	1.56	13.61	20.31	0.00
TAMMEA vs. CATG	13.09	4.78	1.23	10.44	15.74	0.00
TAMMEA vs. GA	4.39	2.34	0.61	3.09	5.68	0.00
TAMMEA vs. HGA	2.53	1.46	0.38	1.72	3.34	0.00
TAMMEA vs. HC	5.94	3.14	0.81	4.20	7.68	0.00
TAMMEA vs. DFC	2.18	1.29	0.33	1.47	2.90	0.00
TAMMEA vs. FUZGENMUT	1.60	1.31	0.34	0.87	2.32	0.00

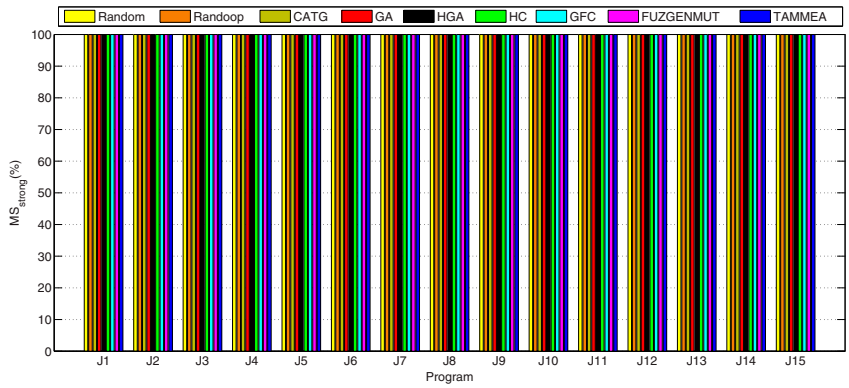
Random had the worst performance because its MS_{strong} of each program was less than 90%.

For further investigation, we presented some statistics of MS_{strong} for the nine methods, as shown in Table XII. The following observations could be made from the Table XII: (1) The maximum value of MS_{strong} for each method was 100% except for four methods (Random, Randoop, CATG and HC), and the minimum value of MS_{strong} for TAMMEA was 97.09%, which was higher than the eight methods. (2) The average MS_{strong} of TAMMEA was 99.44%, which was also higher than the other eight methods, while Random had the lowest mean value of MS_{strong} (76.33). (3) The standard deviation and variance of TAMMEA were 0.85 and 0.72, respectively, lower than those of the other eight methods. (4) The results of skewness and kurtosis implied that MS_{strong} of all nine methods approximately obeyed the normal distribution. In a word, these statistics data reinforced that TAMMEA was better than the other eight baseline methods in terms of fault-detection effectiveness.

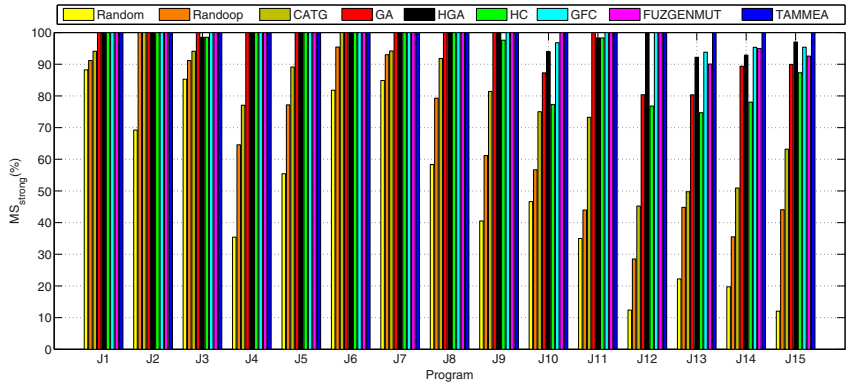
Based on the above results for MS_{strong}, we also used SPSS for T-test to verify whether TAMMEA was more significant than the other eight baseline techniques and the T-test results were shown in Table XIII. The p-value in the last column of Table XIII was less than 0.05 for each pair of comparison (such as TAMMEA vs. Random, TAMMEA vs. GA, etc.), which implied the significant difference between TAMMEA and the other eight baseline techniques. In other words, the average MS_{strong} obtained by TAMMEA was significantly higher than the other eight methods in case of the same sample.

In particular, the observation made in our experiments (test data generated by TAMMEA could kill more mutants than other techniques) was bounded by the same termination constraints such as the same number of test data for Random, and the number of required paths had reached for GA-related techniques. Other techniques could kill all mutants killed by TAMMEA if they were allowed to generate more test data. Since our study was focused on improving the efficiency of mutation testing, the above same termination constraints were required in our experiments for a fair comparison of efficiency between TAMMEA and other techniques.

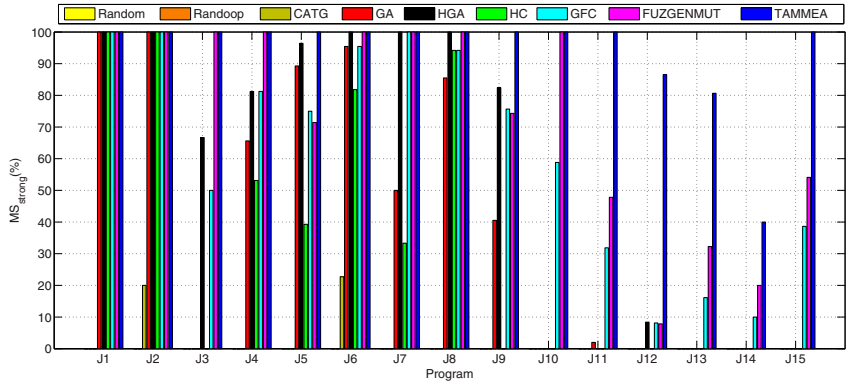
To discover the reasons why TAMMEA could kill more mutants under the same constraints, we further divided all non-equivalent mutants for each tested program into four categories (easy, middle, difficult, and very difficult) based on the method



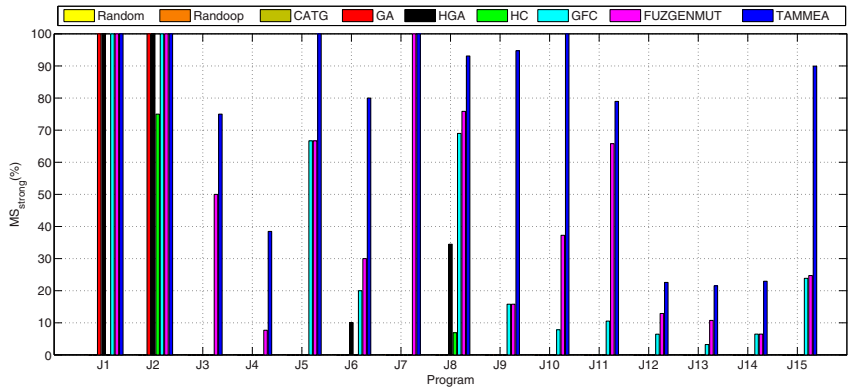
(a) Easy



(b) Middle



(c) Difficult



(d) Very difficult

Fig. 16. The comparison results of effectiveness for TAMMEA and other eight baseline techniques under different categories of mutants.

```
public class FourBalls{
1: public static int [] relativeWeight( int cual, int mA,
int mB, int mC, int mD ){
2:      int[] r = new int[3];
3:      if (cual == 1) {
4:         r[0] = mA / mB;
5:         r[1] = mA / mC;
6:         r[2] = mA / mD;
7:      } else {
8:         if (cual == 2) {
9:            r[0] = mB / mA;
10:           r[1] = mB / mC;
11:           r[2] = mB / mD;
12:        } else {
13:        if (++cual == 3) { //if (cual == 3) {
.....
}
```

Fig. 17. A mutant M of J3 after using $AOIS$ mutation operator.

proposed by Dang et al. [16] (as shown in Section IV-D) and obtained the effectiveness of each method for each category, as shown in Fig. 16.

From Fig. 16, we could observe that: 1) For the easy category (as shown in Fig. 16(a)), each of the nine methods could achieve $MS_{strong} = 100\%$ on each tested program, which indicated that each method had equal effectiveness in this type. 2) For the middle category (as shown in Fig. 16(b)), TAMMEA (blue bar) could achieve $MS_{strong} = 100\%$ for all subject programs, whereas each of the other eight methods had at least two subject programs with $MS_{strong} < 100\%$ and Random had the worst performance because its MS_{strong} of each program was less than 90%. 3) For the difficult category (as shown in Fig. 16(c)), TAMMEA could achieve $MS_{strong} = 100\%$ for each tested program except for J12, J13, and J14, whereas each of the other eight methods had at least four programs with $MS_{strong} < 100\%$. Furthermore, TAMMEA could obtain $MS_{strong} >= 40\%$ for J12, J13, and J14, while each of the other eight methods obtained $MS_{strong} < 40\%$. 4) For the very difficult category (as shown in Fig. 16(d)), TAMMEA could achieve a higher MS_{strong} than each of the other eight methods for each program except for (J1, J2, and J7) and FUZGENMUT was only second to TAMMEA on the whole, whereas each of the other seven methods had at least two subject programs with $MS_{strong} = 0\%$ and three methods (Random, Randoop, and CATG) had the worst performance because its MS_{strong} of each program was 0%. Based on the above analysis of results, we could draw a conclusion: under the same constraints, our method could kill more mutants (these mutants were usually deeply buried in the code or deeply nested in the path) for the difficult type (or very difficult type) than other methods, which made our method have a higher MS_{strong}.

In addition, we used code from J3 (other tested programs were similar to J3) as an example based on the previous experimental results to specifically illustrate the characteristics of mutants for the very difficult category. We randomly selected a mutant M (as shown in Fig. 17) from all mutants for the very difficult category and M was obtained by using mutation operator $AOIS$ to mutate the statement $if(cual == 3)$ in J3.

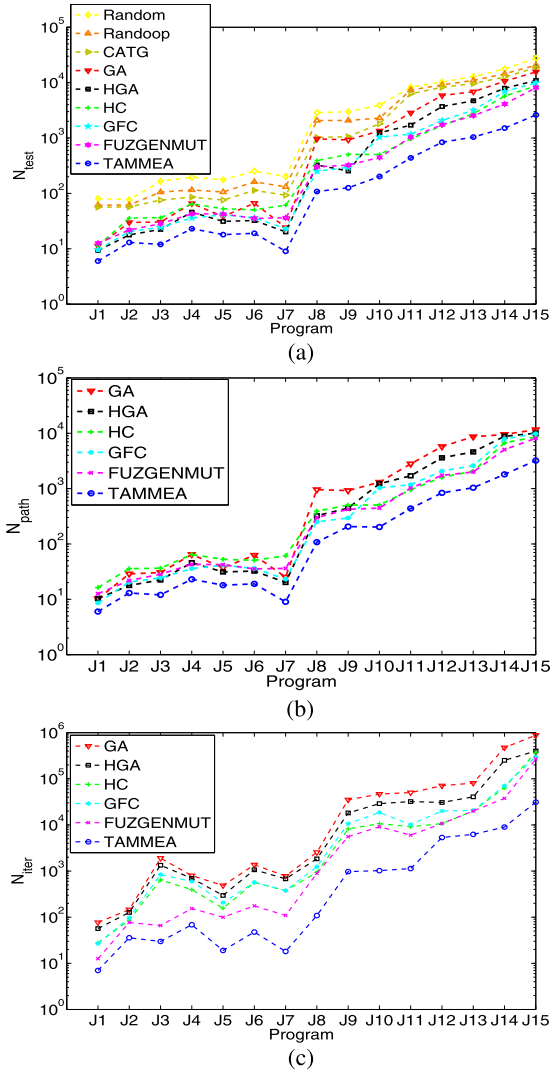


Fig. 18. The comparison results of efficiency for TAMMEA and other eight baseline techniques.

From Fig. 17, we could observe that M was deeply buried in the code or deeply nested in the path, which meant that our method could touch and kill it while other methods could not touch and kill it. Similar to M, other mutants from the very difficult category also met the same characteristics.

In summary, the test data generated by TAMMEA had higher fault-detection effectiveness than those of the other eight baseline techniques under strong mutation criterion.

B. Answer to RQ2: Efficiency

Fig. 18 presents the comparison results of efficiency for TAMMEA and the eight baseline techniques (Random, Randoop, CATG, GA, HGA, HC, GFC and FUZGENMUT) under the situation of killing all non-equivalent mutants for the weak mutation criterion. From Fig. 18, N_{test} represented the average number of test cases under the condition of killing non-equivalent mutants for each of the seven methods, while N_{path} and N_{iter} referred to the average number of iterations and the

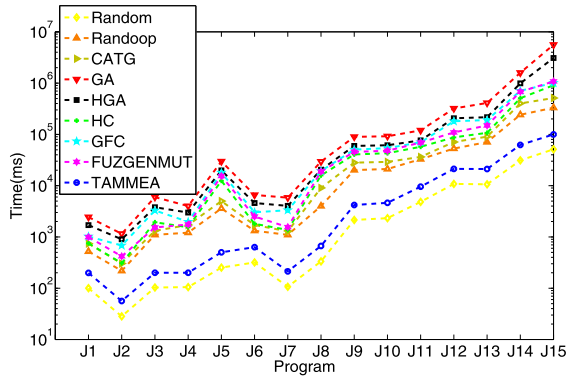


Fig. 19. The results of execution time for TAMMEA and other eight baseline techniques.

average number of paths incurred by each of the six methods except for three methods (Random, Randoop and CATG).

From Fig. 18, we could observe that: (1) From Fig. 18(a), the N_{test} (blue dotted line) generated by TAMMEA for each of 15 subject programs was lower than those of the other eight baseline techniques and Random performed worst in all programs. (2) From Fig. 18(b), the N_{path} (blue dotted line) obtained by TAMMEA for each of 15 subject programs was lower than those of the other five baseline techniques except for three methods (Random, Randoop and CATG), while FUZGENMUT was second only to TAMMEA and GA performed worst in all programs. (3) From Fig. 18(c), the N_{iter} (blue dotted line) obtained by TAMMEA for each of 15 subject programs was lower than those of the other five baseline techniques except for three methods (Random, Randoop and CATG), but GA performed worst and had very little difference from HGA in all programs.

In summary, under the same condition of meeting the weak mutation criterion, TAMMEA could generate fewer test cases than the other eight baseline techniques, and TAMMEA also incurred fewer examine fewer paths and iterations than the other five baseline techniques (GA, HGA, HC, GFC and FUZGENMUT). These implied that TAMMEA had much higher efficiency than the other eight baseline techniques.

C. Answer to RQ3: Execution Time

Fig. 19 summarizes the results of execution time for TAMMEA and the other eight baseline techniques.

From Fig. 19, we obtained the following results: 1) For all 15 subject programs, TAMMEA spent less time generating test data on each program than the other seven baseline techniques (Randoop, CATG, GA, HGA, HC, GFC and FUZGENMUT); 2) For all 15 subject projects, TAMMEA spent slightly more time generating test data on each program than Random (less than 2 times in the same unit time). Note that the reason why Random executed quickly was that it did not require evolutionary iterations after generating data, while TAMMEA (GA, HGA, HC, GFC and FUZGENMUT) required iterations to find test data that meets the requirements.

TABLE XIV
THE TIME OF GENERATING TEST DATA WITH TAMMEA UNDER EACH PROGRAM

ID	*Time* (ms)	ID	*Time* (ms)	ID	*Time* (ms)
J1	199.18	J6	631.36	J11	9538.36
J2	56.24	J7	212.78	J12	21238.37
J3	200.23	J8	666.06	J13	21138.38
J4	200.37	J9	4196.83	J14	62201.68
J5	500.67	J10	4626.97	J15	99914.69

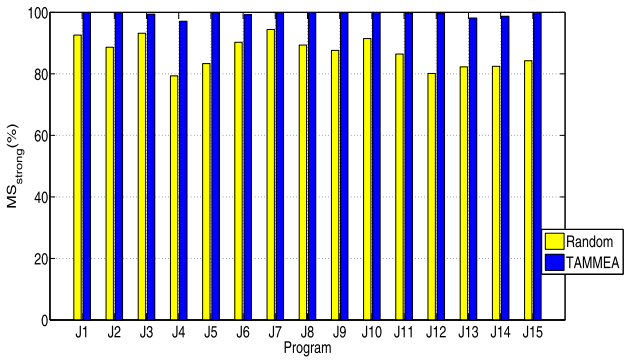


Fig. 20. The comparison results of MS_{strong} for TAMMEA and Random under the same time constraint.

TABLE XV
T-TEST RESULTS OF COMPARING TAMMEA AND RANDOM UNDER THE SAME TIME CONSTRAINT

Comparison Methods	Mean Value	Standard Deviation	Standard Error	95% Confidence Interval		p-Value (Bilateral)
				Lower Limit	Upper Limit	
TAMMEA vs. Random	12.38	4.45	1.15	9.92	14.85	0.00

In addition, we further compared the effectiveness of our method and Random under the same time constraints (note that the time of generating test data for Random was the same with that of TAMMEA as shown in Table XIV), and the specific experimental results are shown in Fig. 20.

From Fig. 20, we could observe that TAMMEA (blue bar) was superior to Random in their mutant-killing capabilities under the same time constraint. Although TAMMEA mainly relied on weak mutation criterion to generate test data, TAMMEA could achieve $MS_{strong} > 96\%$ for all subject programs, while Random had the worst performance because its MS_{strong} of each program was less than 95%. In other words, our method is still more effective than Random under the same time constraint.

Based on the above results for MS_{strong}, we also used SPSS for T-test to verify whether TAMMEA was more significant than Random and the T-test results were shown in Table XV. The p-value in the last column of Table XV was less than 0.05 for each pair of comparison (TAMMEA vs. Random), which implied the significant difference between TAMMEA and Random. In other words, the average MS_{strong} obtained by TAMMEA was significantly higher than Random in case of the same sample.

In summary, TAMMEA took about the same order of magnitude of time as Random, which was better than all other techniques involved; on the other hand, TAMMEA could obtain less test data with higher defect detection capabilities without spending too much time.

VI. Related Work

In this section, we discuss the existing studies related to the present work, especially those for the test data generation under the strong and weak mutation testing.

A. Test Data Generation Method Based on Strong Mutation Testing

Offutt [47] conducted some pioneer work to find an effective method to generate test data for strong mutation testing. Since then, researchers have proposed many methods, the main categories of which include the symbolic execution method (SEM), the search-based method (SBM), and the hybrid search-based method (HSBM).

For SEM, some researchers used a constraint condition to generate test data based on the analysis of data control flow and SEM [48]. Furthermore, Offutt et al. [49] proposed a method based on backtracking search to dynamically narrow the search domain for improving the efficiency of test data generation. Similarly, Sen et al. [50] also used SEM to analyze the path of test data during execution, but they could not give an automatic method for generating test data.

For SBM, Fraser et al. [12] used GA to generate test data for killing mutants, but it was mainly applied to Java programs. Souza et al. [37] developed an automatic test data generation method with the hill climbing algorithm to carry out strong mutation of genotypes, but it did not have a high efficiency due to the diversity of the population. Du et al. [38] proposed a method combined with data flow constraint for generating mutation test data, which fully considered data flow constraints and weighed the importance of the data dependent nodes.

For HSBM, Khan et al. [13] developed the generation of automatic test cases with mutation analysis and hybrid genetic algorithm, which was better with hybrid genetic algorithm and produced maximum mutation score. In addition, Zheng et al. [51] proposed a minimization algorithm for generating test data required by parallel programs in the mutation testing process, but it was time-consuming and inefficient.

B. Test Data Generation Method Based on Weak Mutation Testing

Different from the strong mutation testing, weak mutation testing only requires two conditions of accessibility and necessity. The current techniques in this context include the branch coverage method (BCM) [4], [5], the iterative relaxation method (IRM) [6], and the search-based method (SBM) [8].

Papadakis et al. [4] proposed a BCM of generating test data for weak mutation testing, which improved the efficiency of test data generation. However, it involves a lot of mutant branches, which increases the difficulty in implementation. To solve the problem, Papadakis et al. [4] proposed the use of genetic algorithm, but a large number of mutants lead to many targets and only one mutant branch was covered each time to generate a test datum, which literally increased the cost of mutation testing. To further resolve the issue, Zhang [32] applied the idea of set evolution to generate test data, which was obviously inefficient due to the execution of a large number of mutant branches.

As for IRM and SBM, Shan et al. [6] proposed a method of combining the mutant branches and the over IRM, which improved the quality of test data and the efficiency of generating mutation test data. However, the symbolic execution and constraint solving methods limited the applicability of the method, and no automatic approach was developed to select the target path. In addition, Du et al. [36] proposed a SBM based on genetic algorithm, which optimized the test data. However, the efficiency of generating test data using this method was relatively low.

VII. Conclusion

Mutation testing provides valuable guides for generating test data that are effective in detecting various types of faults, which are mimicked by a huge amount of mutants. However, the test data generation in mutation testing tends to incur a high cost due to the large number of mutants normally involved in the testing process. For example, large portion of generated test data may be redundant in terms of mutant-killing capabilities, and many mutants are required to be executed repeatedly before finding fault-revealing test data.

In this paper, we propose a new approach for generating test data by integrating MCUM and EDA into the weak mutation testing. In the TAMMEA framework that implements the new approach, we first apply MCUM to reduce mutant branches and thus to generate a set of extended paths, the coverage of which is considered as identical to the problem of generating test data. After that, we use an EDA with probability model to solve the problem, which can help generate test data that are particularly capable of killing multiple mutants at the same time. The experimental results show that TAMMEA can generate a relatively small set of test data of killing most mutants even under the strong mutation testing and the execution time of TAMMEA is in the same order of magnitude as that of Random, which clearly demonstrate that TAMMEA can largely improve the cost-effectiveness of mutation testing, and thus enhance its practicality.

Despite the encouraging results of this study, some further optimizations are required in the future. On the one hand, it is worthwhile to conduct in-depth investigations on the fine-tuned mechanism for determining STP for improving the performance of our approach. On the other hand, the input considered in this paper is mainly conventional type, such as integer or string. Basic data types can be used as objects themselves. So, in a sense, it can also be said that our programs take objects as inputs. However, if the input is of a complex type, our method needs to be modified to some extent, such as the probability model in the estimation of distribution algorithm. In this case,

the applicability and efficiency of our method need further verification. Due to limitations in the length of this article, we will consider it as one of our future research work.

REFERENCES

[1] R. A. Demillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, Apr. 1978.

[2] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 654–665.

[3] B. H. Smith and L. Williams, "On guiding the augmentation of an automated test suite via mutation analysis," *Empirical Softw. Eng.*, vol. 14, no. 3, pp. 341–369, 2009.

[4] M. Papadakis and N. Malevris, "Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing," *Softw. Qual. J.*, vol. 19, no. 4, pp. 691–723, 2011.

[5] M. Papadakis and N. Malevris, "Mutation based test case generation via a path selection strategy," *Inf. Softw. Technol.*, vol. 54, no. 9, pp. 915–932, 2012.

[6] J. Shan, Y. Gao, M. Liu, J. Liu, L. Zhang, and J. Sun, "A new approach to automated test data generation in mutation testing," *Chin. J. Comput.*, vol. 31, no. 6, pp. 1025–1034, 2008.

[7] J. Shan, J. Wang, Z. Qi, and J. Wu, "Improved method to generate path-wise test data," *J. Comput. Sci. Technol.*, vol. 18, no. 2, pp. 235–240, 2003.

[8] X. Zhao and B. Gu, "Generation of test cases in mutation testing using genetic algorithm," *J. Comput. Appl.*, vol. 29, no. 0z1, pp. 262–264, 2009.

[9] A. J. Offutt and R. H. Untch, "Mutation 2000: Uniting the orthogonal," in *Proc. 1st Workshop Mutation Anal. (MUTATION)*, 2001, pp. 34–44.

[10] Y. Ma, J. Offutt, and Y. R. Kwon, "MuJava: An automated class mutation system," *Softw. Testing, Verification Rel.*, vol. 15, no. 2, pp. 97–133, 2005.

[11] X. Chen and Q. Gu, "Mutation testing: Principal, optimization and application," *J. Frontier Comput. Sci. Technol.*, vol. 6, no. 12, pp. 1057–1075, 2012.

[12] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," *IEEE Trans. Softw. Eng.*, vol. 38, no. 2, pp. 278–292, 2012.

[13] R. Khan, M. Amjad, and A. K. Srivastava, "Generation of automatic test cases with mutation analysis and hybrid genetic algorithm," in *Proc. 3rd Int. Conf. Comput. Intell. Commun. Technol. (CICT)*, 2017, pp. 1–4.

[14] R. A. DeMilli and A. J. Offutt, "Constraint-based automatic test data generation," *IEEE Trans. Softw. Eng.*, vol. 17, no. 9, pp. 900–910, Sep. 1991.

[15] X. Yao, G. Zhang, F. Pan, D. Gong, and C. Wei, "Orderly generation of test data via sorting mutant branches based on their dominance degrees for weak mutation testing," *IEEE Trans. Softw. Eng.*, vol. 48, no. 4, pp. 1169–1184, Apr. 2022.

[16] X. Dang, D. Gong, X. Yao, T. Tian, and H. Liu, "Enhancement of mutation testing via fuzzy clustering and multi-population genetic algorithm," *IEEE Trans. Softw. Eng.*, vol. 48, no. 6, pp. 2141–2156, Jun. 2022.

[17] C. Wei, X. Yao, D. Gong, and H. Liu, "Spectral clustering based mutant reduction for mutation testing," *Inf. Softw. Technol.*, vol. 132, 2021, Art. no. 106502.

[18] W. E. Howden, "Weak mutation testing and completeness of test sets," *IEEE Trans. Softw. Eng.*, vol. 8, no. 4, pp. 371–379, Jul. 1982.

[19] A. J. Offutt and S. D. Lee, "How strong is weak mutation?" in *Proc. Symp. Testing*, 1991, pp. 200–213.

[20] J. A. Whittaker and J. H. Poore, "Markov analysis of software specifications," *ACM Trans. Softw. Eng. Methodol.*, vol. 2, no. 1, pp. 93–106, 1993.

[21] J. A. Whittaker and M. G. Thomason, "A Markov chain model for statistical software testing," *IEEE Trans. Softw. Eng.*, vol. 20, no. 10, pp. 812–824, Oct. 1994.

[22] Y. Wang, F. Ye, X. Zhu, and C. Wu, "A method for software reliability test case design based on Markov chain usage model," in *Proc. Int. Conf. Qual., Rel., Risk, Maintenance, Saf. Eng.*, 2013, pp. 1207–1210.

[23] J. Whittaker and J. Poore, "Statistical testing for cleanroom software engineering," in *Proc. 25th Hawaii Int. Conf. Syst. Sci.*, 1992, pp. 428–436.

[24] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA, USA: Addison-Wesley, 1988.

[25] Z. Shude and S. Zengqi, "A survey on estimation of distribution algorithms," *Acta Automatica Sinica*, vol. 33, no. 2, pp. 113–124, 2007.

[26] D. Goldberg, *The Design of Innovation: Lessons from and for Competent Genetic Algorithms*. Norwell, MA, USA: Kluwer, 2002.

[27] J. J. Grefenstette, "Optimization of control parameters for genetic algorithms," *IEEE Trans. Syst., Man, Cybern.*, vol. 16, no. 1, pp. 122–128, Jan. 1986.

[28] H. Lei and L. Chen, "Test case generation based on Markov chain usage model," *J. Univ. Electron. Sci. Technol. China*, vol. 40, no. 5, pp. 732–736, 2011.

[29] Y. Dong and J. Peng, "Automatic generation of software test cases based on improved genetic algorithm," in *Proc. Int. Conf. Multimedia Technol.*, 2011, pp. 227–230.

[30] P. Hu and D. Hu, "Research on automatic generation of string type test data based on genetic algorithms," *J. Huaihua Univ.*, vol. 27, no. 8, 2008, Art. no. 3.

[31] B. Korel, "Automated software test data generation," *IEEE Trans. Softw. Eng.*, vol. 16, no. 8, pp. 870–879, Aug. 1990.

[32] G. Zhang, "Test case generation for mutation testing based on set evolution and dominance relation," Ph.D. dissertation, China Univ. Mining Technol., Xuzhou, China, 2017.

[33] Y. Yu, "Research on key technologies of dynamic hardware and software partitioning," Ph.D. dissertation, Tianjin Univ., Tianjin, China, 2011.

[34] C. Pacheco and M. D. Ernst, "RANDOOP: Feedback-directed random testing for Java," in *Proc. Companion 22nd Annu. ACM SIGPLAN Conf. Object-Oriented Program., Syst., Lang., Appl.*, 2007, pp. 1–2.

[35] M. Masud, A. Nayak, M. Zaman, and N. Bansal, "Strategy for mutation testing using genetic algorithms," in *Proc. Can. Conf. Elect. Comput. Eng.*, 2005, pp. 1049–1052.

[36] Y. Du, Y. Pan, H. Ao, N. O. Alexander, and Y. Fan, "Automatic test case generation and optimization based on mutation testing," in *Proc. IEEE 19th Int. Conf. Softw. Qual., Rel. Secur. Companion (QRS-C)*, 2019, pp. 522–523.

[37] F. C. M. Souza, M. Papadakis, Y. L. Traon, and M. E. Delamaro, "Strong mutation-based test data generation using hill climbing," in *Proc. IEEE/ACM 9th Int. Workshop Search-Based Softw. Testing (SBST)*, 2016, pp. 45–54.

[38] X. Du, Q. Qiang, and L. Huang, "A method combined with data flow constraint for generating mutation test cases," *Softw. Eng. Appl.*, vol. 7, no. 2, pp. 99–109, 2018.

[39] S. Liu and Y. Li, "An auotmatic data generation test method," *Comput. Eng.*, vol. 13, no. 5, pp. 26–32, 1987.

[40] D. Gong, F. Pan, T. Tian, S. Yang, and F. Meng, "A feedback-directed method of evolutionary test data generation for parallel programs," *Inf. Softw. Technol.*, vol. 124, no. 2, pp. 1–15, 2020.

[41] H. Tanno, X. Zhang, T. Hoshino, and K. Sen, "TesMa and CATG: Automated test generation tools for models of enterprise applications," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, 2015, pp. 717–720.

[42] K. Lakhotia, M. Harman, and H. Gross, "AUSTIN: A tool for search based software testing for the C language and its evaluation on deployed automotive systems," in *Proc. 2nd Int. Symp. Search Based Softw. Eng.*, 2010, pp. 101–110.

[43] K. K. Mishra, S. Tiwari, A. Kumar, and A. K. Misra, "An approach for mutation testing using elitist genetic algorithm," in *Proc. 3rd Int. Conf. Comput. Sci. Inf. Technol.*, 2010, pp. 426–429.

[44] A. P. Mathur and W. E. Wong, "An empirical comparison of data flow and mutation-based test adequacy criteria," *Softw. Testing Verification Rel.*, vol. 4, no. 1, pp. 9–31, 2010.

[45] L. Zhang, S. Hou, J. Hu, T. Xie, and H. Mei, "Is operator-based mutant selection superior to random mutant selection?" in *Proc. ACM/IEEE 32nd Int. Conf. Softw. Eng.*, 2010, pp. 435–444.

[46] X. Yao, M. Harman, and Y. Jia, "A study of equivalent and stubborn mutation operators using human analysis of equivalence," in *Proc. 36th Int. Conf. Softw. Eng. (ICSE)*, Hyderabad, India, 2014, pp. 919–930.

[47] A. J. Offutt, *Automatic Test Data generation*. Eastern United States: Georgia Institute of Technology, 1988.

[48] G. Ji, "Survey on genetic algorithm," *Comput. Appl. Softw.*, vol. 21, no. 2, pp. 69–73, 2004.

[49] A. J. Offutt, Z. Jin, and J. Pan, "The dynamic domain reduction procedure for test data generation," *Softw. Pract. Experience*, vol. 29, no. 2, pp. 167–193, 1999.

[50] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," *ACM SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 263–272, 2005.
[51] W. Zheng, C. Feng, X. Wu, Y. Huang, and L. Fang, "Mutation test based test case minimization for concurrent program," *Comput. Sci.*, vol. 44, no. 11, pp. 109–113, 2017.

Changqing Wei received the master's degree in applied mathematics from China University of Mining and Technology, in 2020. Currently he is working toward the Ph.D. degree with the School of Mathematics, China University of Mining and Technology. His main research interests include search-based software testing.

Xiangjuan Yao received the Ph.D. degree in control theory and control engineering from China University of Mining and Technology, in 2011. She is a Professor with the School of Mathematics, China University of Mining and Technology. Her main research interests include intelligence optimization and search based software testing.

Dunwei Gong (Senior Member, IEEE) received the Ph.D. degree in control theory and control engineering from China University of Mining and Technology, in 1999. He is a Professor with the College of Automation and Electronic Engineering, Qingdao University of Science and Technology. His main research interests include intelligence optimization and control.

Huai Liu (Senior Member, IEEE) received the B.Eng. degree in physioelectronic technology and M.Eng. degree in communications and information systems, both from Nankai University, China, and the Ph.D. degree in software engineering from the Swinburne University of Technology, Australia. He is a Lecturer with the Department of Computing Technologies, Swinburne University of Technology, Melbourne, Australia. He has worked as a Lecturer with Victoria University and a Research Fellow with RMIT University. Prior to working in higher education he worked as an Engineer in the IT industry. His current research interests include software testing, cloud computing, and end-user software engineering.