IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received July 17, 2020, accepted July 28, 2020, date of publication August 5, 2020, date of current version August 18, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3014290

Multi-Task Optimization-Based Test Data Generation for Mutation Testing via Relevance of Mutant Branch and Input Variable

XIANGYING DANG[1,2,3], XIANGJUAN YAO[4], DUNWEI GONG[1], (Member, IEEE), TIAN TIAN[5], AND BAICAI SUN[1]

[1]School of Information and Control Engineering, China University of Mining and Technology, Xuzhou 221116, China
[2]School of Information Engineering (School of Big Data), Xuzhou University of Technology, Xuzhou 221000, China
[3]Key Laboratory of Intelligent Industrial Control Technology of Jiangsu Province, Xuzhou 221000, China
[4]School of Mathematics, China University of Mining and Technology, Xuzhou 221116, China
[5]School of Computer Science and Technology, Shandong Jianzhu University, Jinan 250101, China

Corresponding author: Dunwei Gong (dwgong@vip.163.com)

This work was supported in part by the National Natural Science Foundation of China under Grant 61773384, in part by the National Key Research and Development Program of China under Grant 2018YFB1003802-01, in part by the Fundamental Research Funds for the Central Universities under Grant 2020ZDPYMS40, in part by the Major Project of Natural Science Research of the Jiangsu Higher Education Institutions of China under Grant 18KJA520012, and in part by the Xuzhou Science and Technology Plan Project under Grant KC19197.

ABSTRACT Mutation testing is a powerful software testing technique. However, it is difficult to obtain test data for killing a large number of mutants, especially for hard-to-kill mutants. Mutant branch is formed by an original statement of a program under test and its mutated statement. The true branch of a mutant branch is covered by a test datum, suggesting the corresponding mutant is killed under the criterion of weak mutation testing. This article focuses on efficiently generating test data for a large number of mutant branches. When generating test data using a search-based method, the size of the search domain is a determining factor affecting the search performance. The key observation is that only partial input variables affect whether a mutant will be killed, so the search domain can be reduced by deleting irrelevant variables. Along this line, we first group the mutant branches based on their relevant input variables, followed by formulating a multi-task optimization model of test data generation for the grouped mutant branches, in which the relevant input variables are taken as the decision variables. Finally, a multi-population genetic algorithm with individual sharing is employed to generate test data by multi-tasking. The experiments based on eight programs of various sizes show that removing related variable helps reduce the search domain, and the efficiency of test data generation by grouping and multitasking is improved.

INDEX TERMS Weak mutation testing, multi-task test data generation, relevant input variables, grouping mutants, multi-population genetic algorithm.

I. INTRODUCTION

Software testing, aiming to find faults in a software product, is one of the vital processes in the life cycle of software development, which generally consists of two phases, i.e., generating test data and executing the software product with the generated test data [1], [2]. Among various testing methods, Mutation testing is a fault-based software testing technique, which is proposed by Hamlet [3] and DeMillo *et al.* [4]. For mutation testing, faults are generally seeded into a program under test deliberately by a number of simple syntactic changes to create a set of faulty programs called mutants [5]. The syntactic change rules are called as mutant operators. Equivalent mutants are syntactically different but functionally equivalent to the original program [6], [7].

Generally, there are two kinds of mutation testing criteria, i.e., strong mutation testing and weak mutation testing [5], [8], [9]. Regarding the criterion of strong mutation testing,

The associate editor coordinating the review of this manuscript and approving it for publication was Luca Ardito.

an incorrect state should be propagated to the output of a program when a test datum is employed to execute the original program and its mutant [10]. If reachability and infection are held for a mutant, the mutant is said killed under the criterion of weak mutation testing [11], [12]. In this paper, we focus on test data generation under the criterion of weak mutation testing.

Generally, when performing mutation testing, a large number of mutants will be generated. To date, various methods have been proposed to reduce the cost of mutation testing [13]–[16]. Along this line, Jia and Harman [1] summarized the cost reduction techniques into two types, reduction of the generated mutants (which combines do fewer and do faster) and reduction of the execution cost (which corresponds to do faster). In this paper, we focus on the latter, i.e., ''do faster'' by ''searching fewer possibilities''.

To efficiently generate test data for killing the mutants, many scholars proposed various methods and developed a number of tools [17]–[19], to improve the efficiency of generating test data. Papadakis *et al.* [5] summarized three main kinds of methods of generating test data for mutation testing, i.e., constraint-based test generation [20], dynamic symbolic execution [8], [21], and search-based test generation [22]–[24]. Among them, search-based methods are widely used, which commonly employ genetic algorithms(GAs) as optimizers [23], [25]–[28].

GAs are one of the representative global optimization algorithms. In GAs, a population composed of some individuals continuously searches optimal solutions in the input domain under the guidance of a specific fitness function. Along this line, Masud *et al.* [26] presented a model of mutation testing based on GAs, which has good performance in detecting faults. The above studies suggest that GAs are competent in generating test data under the criterion of mutation testing. Compared with a GA with only one population, a GA with multiple sub-populations, called a multi-population genetic algorithm (MGA), generally has more superior performance. Yao and Gong [25] proposed a MGA with individual sharing to generate test data for covering multiple paths. In MGA, each sub-population optimizes one subproblem which corresponds to one path, so the fitness functions of different sub-populations differ from each other. All sub-populations evolve in parallel. Based on the above ideas, MGA is applied to test data generation for mutation testing in this paper.

Previous studies [23], [29]–[31] discovered that search domain size is related to search performance(the ability to find test data). McMinn *et al.* [29] discovered that the coverage of a branch is affected by only some input variables. In mutation testing, we also observe that whether a mutant is killed or not is related to only some input variables. Thus, to improve the efficiency of generating test data by reducing the search domain, we can remove the irrelevant input variable. In addition, different mutants may be related to the same input variables, as a result, we can group these mutants. On that basis, a multi-task optimization model of mutation-based test data generation is formulated. Finally,

```
.....
int x1,x2,x3,x4,x5,x6;
if(x1<20 && x1+x2<50)
{
   if(x3==x4)
     x4= x4+5;
   else
     { x4= x4+x3;
       x5= x5%3; }
   if ( x6<=x4)
     { x6=x6%3;}
   .....
}
```

(a) The original program, G

```
.....
int x1,x2,x3,x4,x5,x6,t;
1 if(x1<20 &&x1+x2<50)
   {
2    if(x3==x4)
3      x4= x4+5;
     else
4      { x4= x4+x3;
         x5= x5%3; }
5    if ( x6<=x4) !=( x6<x4) ·····M1
     if ( x6<=x4)
       { t=x6;
         if (x6 %3)!=((++t) %3) ····· M2
         x6=x6 %3;}
       .....
   }
```

(b) The program G' inserted with mutant branches, $M1$,$M2$

FIGURE 1. One example.

MGA is employed to generate test data for the proposed multi-task model. In this way, we expect to improve the efficiency of test data generation for a large number of mutants.

This paper has the following three-fold contributions: (1) proposing a method of grouping the mutant branches based on the relevant input variables, (2) formulating the problem of multi-task mutation-based test data generation as an optimization one, and (3) efficiently generating test data by MGA.

This paper is organized as follows. Section II reviews the related work. The emphasis of this paper is in Section III, grouping mutant branches based on relevant variables, optimization model of multi-task mutation-based test data generation, and test data generation via MGA. The experiments and analysis are in Sections IV and V. Threats to validity are discussed in Section VI. Finally, Section VII concludes the whole paper, and directs several opportunities for future research.

II. RELATED WORKS

A. WEAK MUTATION TESTING TRANSFORMATION

Let G be a program under test with the input vector, X. After performing a mutation operator on a statement, s_i, in G, a mutated statement is obtained, denoted as s_i'. Based on the necessary conditions of mutation testing, Papadakis and Malevris [9] built the conditional statement, ''$if\ s_i \neq s_i'$'', which is defined as *the* ***mutant branch*** [32], [33], denoted as M_i. One mutant branch corresponds to one mutant. In this way, they transformed the problem of killing a mutant under weak mutation testing into the problem of covering the true branch of a mutant branch. By applying various mutation operators on the statements of G, we can obtain a mutant branch set, denoted as $M = \{M_1, M_2, \ldots, M_n\}$, where n is the number of mutant branches.

For example, Fig. 1 (a) is a part of the program under test. After performing the mutation operator on ''$x[6] \leq x[4]$'', the mutated statement, ''$x[6] < x[4]$'', is generated. Based on [9], we can obtain the conditional statement, ''$if\,(x[6] \leq x[4])! = (x[6] < x[4])$'', which is defined as mutant branch, M_1. If a test datum can cover the true branch of $if\,(x[6] \leq x[4])! = (x[6] < x[4])$, M_1 is killed under weak mutation criterion. Fig. 1(b) shows a new program under test inserted with two mutant branches, M_1 and M_2.

B. TRADITIONAL OPTIMIZATION MODEL OF MUTATION-BASED TEST DATA GENERATION

Papadakis and Malevris [9] employed the symbolic execution method to generate test data for the mutant branches, however, for many mutants, it is difficult to find test data. In view of the above problem, Zhang [32] used a set evolution method to generate test data for a large number of mutants, and the efficiency of mutation testing is greatly improved.

In addition, we formulate a problem of test data generation as an optimization one [23], the decision vector is defined as the input vector X of program G, and the input domain is defined as $D(X)$. For a mutant branch, M_i, its objective function is denoted as $f_i(X)$, if X can kill M_i, $f_i(X) = 0$, otherwise $f_i(X) = 1$. As a result, M_i is killed if and only if $f_i(X)$ takes the minimum value of zero.

Given that the value of $f_i(X)$ is either 0 or 1, such a function has the difficulty in guiding the evolution of a population. From the beginning of the program to s', the path that is easy to cover is selected as the target path, denoted P_i. $P(X)$ is defined as a path that is covered by X. The similarity between $P(X)$ and P_i, denoted as $g_i(X)$, which is the constraint function.

Based on $f_i(X)$, $g_i(X)$ and $D(X)$, the optimization model of mutation-based test data generation [23] can be described as:

$$\begin{aligned} &min\ f_i(X) \\ &s.t.\ \begin{cases} g_i(X) = 1 \\ X \in D(X) \end{cases} \end{aligned} \tag{1}$$

Eq. 1 shows that the more input variables are, the larger the search domain is. In this paper, we will improve the traditional optimization model of Eq. 1 by removing irrelevant variables.

III. THE PROPOSED METHOD

The framework of the proposed method is shown in Fig. 2. A meta-program with the mutant branches is composed following the previous research [32], [33]. Each mutant branch is corresponding to one mutant. In Section A, we first determine the relevance between the input variables and the mutant branches, based on which mutant branches are grouped. Then, a multi-task optimization model of test data generation was established for the grouped branches, which is elaborated In Subsection B. Finally, in Subsection C, the MGA is used to solve the multi-task optimization problem.

A. GROUPING MUTANT BRANCHES BASED ON RELEVANT VARIABLES

In this section, we first seek the input variables related to a mutant branch. Then, mutant branches which have the same relevant inputs are divided into a group.

- **Determining the relevance between a mutant branch and an input variable**

We discover that not every input variable will be responsible for determining whether a mutant is killed or not. Thus, the irrelevant variables should be removed, which helps to reduce the search domain.

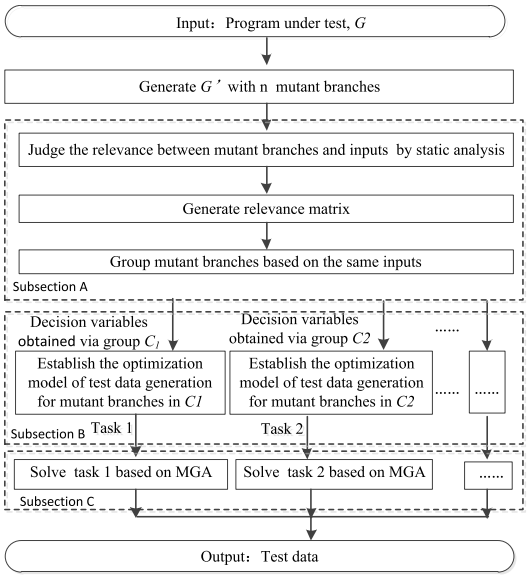


FIGURE 2. Framework of the proposed method.

Let $X = (x_1, x_2, \cdots, x_m)$, where x_j is an input variable and m is the number of input variables. $D(x_j)$ is the domain formed by all the values of input variable x_j, and the domain of all variables can be expressed as $D(X) = D(x_1) \times D(x_2) \times \cdots \times D(x_m)$.

Let $D^*(x_j) \subseteq D(x_j)$, which is a subdomain of $D(x_j)$. In some cases, although x_j is relevant to M_i in $D(x_j)$, x_j in $D^*(x_j) \subset D(x_j)$ is not relevant to mutant branch, M_i. We expect to find $D^*(x_j)$, which helps to reduce the search domain, $D(x_j)$.

If an input variable, x_j, in $D^*(x_j)$ is capable of influencing whether M_i will be killed or not, we say that x_j is a **relevant variable** of M_i in $D^*(x_j)$; otherwise, x_j is an **irrelevant variable** of M_i in $D^*(x_j)$.

In general, whether an input variable is relevant or irrelevant for a particular target (killing M_i) is an undecidable problem. However, it is possible to obtain a conservative estimate of relevance using static analysis techniques.

Zhang and Gong [31] proposed a static analysis method of determining the relevance between the input variables and a path. From this viewpoint, we transform the relevance between input variables and M_i into the relevance between input variables and a path. Generally, multiple paths can reach the mutant branch, M_i, However, it is too costly to determine the relevance between M_i and the input variables involved in all paths. Thus, from these paths, we select one path that is easy to cover. This path is also the target path, P_i in the traditional model of Eq. 1. In this way, we only analyze the relevance between M_i and the input variables involved in P_i. If each node in P_i is not influenced by x_j in $D(x_j)$, x_j is irrelevant input variable to M_i. For more technical details, please refer to [31].

To seeking $D^*(x_j)$, we also start with P_i. Eq. 1 shows that if a test datum, X, is expected to kill M_i, X must first reach M_i through P_i. From this perspective, the value of x_j in $D^*(x_j)$

must cover P_i. Thus, $D^*(x_j)$ is obtained by the constraints of conditional statements in P_i, for more details about the constraints of conditional statements, please refer to [34].

Take the program in Fig. 1(b) as an example, we illustrate how to determine the relevance between the input variables and the mutant branches. The input vector of this program is $X = \{x1, x2, x3, x4, x5, x6\}$, which is integer, and $D(X) = [1, 64]^6$. There are two paths reaching M_1, i.e., "1, 2, 3, 5" and "1, 2, 4, 5". Referring to previous research [23], we select "1, 2, 4, 5" as a target path, denoted as P_1, because `statement 4` (the false branch of "$if(x3 == x4)$") has a high execution probability, suggesting that "1, 2, 4, 5" is easier to cover than "1, 2, 3, 5".

Next, we investigate whether $x1, x2, x3, x4, x5$ and $x6$ are related to M_1 based on P_1.

Regarding $x1$ and $x2$, they only exist in `statement 1` ("$if((x1 < 20)\&\&((x1 + x2) < 50))$"). We can obtain $D^*(x1) = [1, 19]$ because of "$x1 < 20$" and $x1 \in [1, 64]$. That is, $x1$ takes different values in $[1, 19]$ which does not influence whether X kills M_1 or not. Further, we can obtain $D^*(x_2) = [1, 30]$ based on interval arithmetic between "$x1 + x2 < 50$" and $x1 \in [1, 19]$. As a result, $x1$ in $D^*(x_1)$ and $x[2]$ in $D^*(x_2)$ are not relevant to M_1. For $x3$, $x4$ and $x6$, because they can influence some nodes on path $P_1 = 1, 2, 4, 5$ based on [31], $x3$, $x4$ and $x6$ are related to M_1. Regarding $x5$, it exists in statement "$x5 = x5\%3$", which is a non-control node and has no connection with other nodes. Thus, $x5$ is not relevant to M_1.

- **Grouping mutant branches with same relevant input variables**

We discover that the relevance between input variables and mutant branches are the *many-to-many relationships*, so we can divide these mutant branches into multiple groups. For this purpose, we first establish the relevant matrix between the input variants and the mutant branches, and then divide the mutant branches based on the relevant matrix.

We establish a so-called **relevant matrix**, Λ, expressed as:

$$\Lambda = \begin{array}{c} \begin{matrix} x_1 & x_2 & \cdots & x_m \end{matrix} \\ \begin{bmatrix} \rho_{11} & \rho_{12} & \cdots & \rho_{1m} \\ \rho_{21} & \rho_{22} & \cdots & \rho_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ \rho_{n1} & \rho_{n2} & \cdots & \rho_{nm} \end{bmatrix} \end{array} \begin{matrix} M_1 \\ M_2 \\ \vdots \\ M_n \end{matrix}$$

where the value of ρ_{ij} is 1 or 0. If x_j is not relevant to M_i, $\rho_{ij} = 0$, otherwise $\rho_{ij} = 1$.

Algorithm 1 gives the details of grouping mutant branches based on Λ. Firstly, we make a copy of Λ, denoted $\overline{\Lambda}$, and investigate the elements in the first row of $\overline{\Lambda}$. If $\rho_{1j} = 0$, the corresponding columns are deleted (Lines 4 to 7). Then, we investigate each line for the reduced $\overline{\Lambda}$, if $\rho_{\gamma j} = 0$, the corresponding lines in $\overline{\Lambda}$ are deleted (Lines 11 to 14). After reducing $\overline{\Lambda}$, the values of ρ_{ij} are all 1, i.e., $\rho_{ij} = 1$, suggesting that M_i and x_j are relevant in the reduced relevant matrix. As as result, we obtain the first group, $C_1 = \{\overline{M}_{1,1}, \overline{M}_{1,2}, \cdots, \overline{M}_{1,\hbar_1}\}$, where $\hbar_1 (\leq n)$ is the number of the mutant branches. The relevant input variables are denoted as $\bar{x}_{1,1}\bar{x}_{1,2}, \cdots \bar{x}_{1,\ell_1}$, where $\ell_1 (\leq m)$ is the number of relevant input variables (Line 18). Further, $\overline{M}_{1,1}, \overline{M}_{1,2}, \cdots, \overline{M}_{1,\hbar_1}$ are deleted from M, and the corresponding lines of $\overline{M}_{1,1}, \overline{M}_{1,2}, \cdots, \overline{M}_{1,\hbar_1}$ and the corresponding columns of $\bar{x}_{1,1}\bar{x}_{1,2}, \cdots \bar{x}_{1,\ell_1}$ are deleted from Λ (Line 18). The above process repeats for the reduced Λ until $M = \emptyset$. Finally, β groups of mutant branches are obtained.

Algorithm 1 Grouping Mutant Branches

Input: M (mutant branch set), X (input vector), Λ (relevant matrix)

Output: β groups of mutant branches

1: int $k = 1$;
2: **while** $M = \emptyset$ **do**
3: $k = k + 1$; $\overline{\Lambda} = \Lambda$
4: **for** $i = 1$ to n **do** \\ n is No. of lines
5: **for** $j = 1$ to m **do** \\ m is No. of columns
6: **if** $\rho_{ij} = 0$ **then**
7: Delete the j-th colum from $\overline{\Lambda}$
8: **end if**
9: **end for**
10: **end for**
11: **for** $\gamma = 2$ to n **do**
12: **for** $j = 1$ to m **do**
13: **if** $\rho_{\gamma j} = 0$ **then**
14: Delete the γ-th line from $\overline{\Lambda}$
15: **end if**
16: **end for**
17: **end for**
18: Obtain group $C_k = \{\overline{M}_{k,1}, \overline{M}_{k,2}, \cdots, \overline{M}_{k,\hbar_k}\}$ and save related variables $\bar{x}_{k,1}\bar{x}_{k,2}, \cdots \bar{x}_{k,\ell_1}$
19: $M = M \backslash \{\overline{M}_{k,1}, \overline{M}_{k,2}, \cdots, \overline{M}_{k,\hbar_k}\}$; Delete the lines corresponding to $\overline{M}_{k,1}, \overline{M}_{k,2}, \cdots, \overline{M}_{k,\hbar_k}$ and the columns corresponding to $\bar{x}_{k,1}\bar{x}_{k,2}, \cdots \bar{x}_{k,\ell_1}$ from Λ
20: **end while**
21: Return β mutant branch groups

Refer to the example given in Fig. 1 again. Suppose we obtain the following relevant matrix.

$$\Lambda = \begin{array}{c} \begin{matrix} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 \end{matrix} \\ \begin{bmatrix} 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \end{array} \begin{matrix} M_1 \\ M_2 \\ M_3 \\ M_4 \\ M_5 \end{matrix}$$

Based on Λ, we can obtain $x3, x4$ and $x6$ are relevant to M_1 and M_2; $x1$ and $x5$ are relevant to M_3 and M_5; $x2$ are relevant to M_4. Three groups are $C_1 = \{M_1, M_2\}$, $C_2 = \{M_3, M_5\}$, and $C_3 = \{M_4\}$, respectively.

B. MULTI-TASK OPTIMIZATION MODEL OF MUTATION-BASED TEST DATA GENERATION

The traditional optimization model (Eq. 1) shows that the decision variables are the entire input variables of a program.

The domain formed by the decision variables is the search domain. If the decision variables are the relevant input variables, the search domain will be reduced, which helps to improve the efficiency of finding desired test data by the search-based methods.

Along this line, we first formulate the problem of generating test data for a mutant branch as an optimization one with a unique constraint, in which the decision variables are the relevant input variables. Then, we establish a multi-task optimization model for the multiple groups of mutant branches.

Suppose that the input variables, $\overline{x}_{k,1}\overline{x}_{k,2}, \cdots \overline{x}_{k,\ell_k}$, are relevant to $\overline{M}_{k,i}$ in C_k, the decision vector is defined as $\overline{X}_k = (\overline{x}_{k,1}\overline{x}_{k,2}, \cdots \overline{x}_{k,\ell_k})$.

For a mutant branch, $\overline{M}_{k,i}$, the optimization model of generating test data can be formulated as:

$$\begin{aligned} &\min\ (f_i^k(\overline{X}_k)) \\ &s.t.\ \begin{cases} g_i^k(\overline{X}_k) = 1 \\ \overline{X}_k \in D(\overline{X}_k) \end{cases} \end{aligned} \tag{2}$$

Eq. 2 is the improved optimization model for the traditional one (Eq. 1).

Since the mutant branch has been divided into β groups, we can divide the optimization problem of generating test data into β sub-optimization problems. For this purpose, a multi-task optimization model of mutation-based test data generation can be expressed as:

$$\begin{aligned} T^1: &\ \min(f_i^1(\overline{X}_1)) \\ &s.t. \begin{cases} g_i^1(\overline{X}_1)) = 1 \\ (\overline{X}_k) \in D(\overline{X}_1), \quad i = 1, 2, \cdots, \hbar_1 \end{cases} \\ T^2: &\ \min(f_i^2(\overline{X}_2)) \\ &s.t. \begin{cases} g_i^2(\overline{X}_2) = 1 \\ (\overline{X}_2) \in D(\overline{X}_2), \quad i = 1, 2, \cdots, \hbar_2 \end{cases} \\ &\cdots \\ T^\beta: &\ \min(f_i^\beta(\overline{X}_\beta)) \\ &s.t. \begin{cases} g_i^\beta(\overline{X}_\beta) = 1 \\ (\overline{X}_\beta) \in D(\overline{X}_\beta), \quad i = 1, 2, \cdots, \hbar_\beta \end{cases} \end{aligned} \tag{3}$$

Eq. 3 shows that group C_k corresponds to subtask T^k, and T^k needs to solve $\hbar_k$ subproblems. When the decision variable is smaller than the input variables of a program, the search domain is reduced, which is conducive to quickly finding the desired test data.

C. MULTI-TASK OPTIMIZATION-BASED TEST DATA GENERATION VIA MGA

Given that different subtasks have different optimization models and decision variables in Eq. 3, each subtask can be solved in parallel, which is conducive to improving the efficiency of test data generation. In each subtask, we only need one sub-population for all of the mutant branches in the same group, which helps to greatly reduce the number of sub-population.

To solve the optimization model (Eq. 3), we employ a multi-population genetic algorithm (**MGA**). In the following, we will first give the representation of the individuals. Following that, we design the fitness function and present genetic operators. Finally, an algorithm of test data generation by MGA is given.

For the representation of the individuals, considering that the decision vector in Eq.(4) is the relevant input vector, the individuals of the population are encoded only on the relevant input vector, $\overline{X}_k = (\overline{x}_{k,1}\overline{x}_{k,2}, \cdots \overline{x}_{k,\ell_k})$, instead of all input vector of a program, $X = (x_1, x_2, \cdots, x_m)$. Similarly, mutation and crossover in genetic algorithms are implemented on $\overline{X}_k$. In this way, the search domain of the population reduces $m - \ell_k$ dimensions.

For an irrelevant input variable, we take a value within its domain, and this value remains unchanged during the evolution press.

Refer to the example given in Fig. 1 again. This program has 6 input variables. Among them, input variables $x3, x4$ and $x6$ are relevant to M_1, which are employed as decision variables. When running MGA, the evolutionary individuals are encoded, crossed and mutated on $x3, x4$ and $x6$. When executing the program, we need input the values of all six variables. To fulfill this task, $x1, x2$ and $x5$ take any value in $D_1^*(x1) = [1, 19]$, $D_2^*(x2) = [1, 30]$, and $D_5(x5) = [1, 64]$, respectively. Throughout the evolutionary process, the values of $x1, x2$ and $x5$ remain unchanged, and only $x3, x4$ and $x6$ implement genetic operations. Suppose a variable contained in an individual be encoded by 6-bit binary. The number of decision variables is 3 in our optimization model, there are only $2^{3\times6}$ candidate solutions at each iteration. However, there are $2^{6\times6}$ candidate solutions based on the traditional model. From this perspective, our method can reduce the search domain from 6 dimensions to 3 dimensions.

The fitness function is employed to drive test data generation during the evolution press. To generate test data for a mutant branch, M_i^k, the fitness function is designed based on Eq.(3), denoted as $fit_i^k(\overline{X}_k)$, formed by the objective function $(f_i^k(\overline{X}_k))$ and the penalty function resulted from the unique constraint function $(g_i^k(\overline{X}_k))$ [23], expressed as:

$$fit_i^k(\overline{X}_k) = f_i^k(\overline{X}_k) \times (1 - g_i^k(\overline{X}_k) + c) \tag{4}$$

where $\overline{X}_k$ is the relevant input vector and c is a small constant to ensure that the value in parentheses is greater than 0. Eq. 4 shows that the smaller the value of $fit_i^k(\overline{X}_k)$ is, the better the corresponding individual is. $\overline{M}_{k,i}$ is killed if and only if $fit_i^k(\overline{X}_k)$ takes the minimum value of zero, i.e., $fit_i^k(\overline{X}_k) = 0$.

Algorithm 2 gives the details of test data generation by MGA with individual sharing to solve the multi-task optimization model of Eq. 3. The number of sub-populations in MGA is the number of subtasks, i.e., β. One sub-population optimizes the problem of test data generation for the mutant branches of one group. The individuals of each subpopulation are encoded according to their decision variables.

In Algorithm 2, we first randomly select a mutant branch in each group as the optimization target (Line 1). For example,

in group C_k, we select $\overline{M}_{k,i}$ as an optimization target. If an individual of X_k is an optimal solution (desired test data) of $\overline{M}_{k,i}$ based on genetic algorithm (Lines 4-5), we reselect an alive mutant branch in C_k as the optimization target (Line 8). If the optimal solution of $\overline{M}_{k,i}$ is not found, we calculate the fitness, implement genetic operations and generate new individuals(Lines 11-13), and until termination criteria are met. Repeating the above processes, if the optimal solutions of all mutant branches in C_k are found, we stop the evolution of this sub-population, and remove subtask T^k (Line 6). The following two termination criteria of Algorithm 2 are adopted. One is that the desired test data are generated for all subtask, i.e, the value of β becomes 0 (Line 3). The other termination criterion is that the population evolves for the maximum number of iterations.

Note that MGA with individual sharing is a key technology in Algorithm 2. When running MGA, the individuals of X_k not only judges whether it is the optimal solution for its corresponding mutant branch, $\overline{M}_{k,i}$ via genetic algorithm (Lines 1-14), but also needs to judge whether it can kill the mutant branches in other groups by executing these variants (Line 15). If some mutant branches are killed by X_k, these mutant branches are marked as killed in their groups, suggesting that these killed mutant branches are not selected as optimization targets (Lines 16-17). In this way, the individuals of different sub-populations are shared with each other, and the opportunity of finding the optimal solution for an optimization target is increased.

Algorithm 2 Multi-Task Optimization-Based Test Data Generation by MGA

Input: M (a mutant branch set), Pop (a population including β sub-populations)
Output: A test suite

1: Determine an optimization target for each group; Initialize the individuals of each sub-populations and assign the values of all parameters;
2: Individuals of each sub-population execute the program with the mutant branches;
3: **while** termination criteria are not met **do**
4: **if** a individual in X_k $(k = 1, 2, \cdots, \beta)$ is the optimal solution of $\overline{M}_{k,i}$ **then**
5: **if** this individual is the optimal solutions of all mutant branches in C_k **then**
6: $\beta = \beta - 1$; Stop the evolution of the sub-populations;
7: **else**
8: Select an alive mutant in C_k as the optimization target; goto **Line 1**;
9: **end if**
10: **else**
11: Calculate fitness based on Eq. 4 for each individual of each sub-populations;
12: Implement selection, crossover and mutation;
13: Generate the new individuals; goto **Line 2**;
14: **end if**
15: Individuals in X_k execute the program with the mutant branches for the different groups;
16: **if** some mutant branches are killed by these individuals in X_k **then**
17: these killed mutant branches are marked as killed in their groups;
18: **end if**
19: **end while**
20: Save the optimal solutions and the killed mutants;
21: Return the generated test suite;

IV. EXPERIMENTS

We conduct a series of experimental studies to evaluate the performance of the proposed method.

A. RESEARCH QUESTIONS

RQ1: **Is it necessary to remove the irrelevant input variables for the mutant branches?**
For the mutant branches in a program under test, what is the proportion of irrelevant variables? If most mutant branches depend on a small number of input variables, the search may become more effective after removing these irrelevant variables.

RQ2: **To what extent does taking relevant input variables as decision variables increase the efficiency of mutation-based test data generation?**
In the traditional optimization model shown as Eq. 1, the decision variables are all input variables of a program, while in the improved optimization model shown as Eq. 2, the decision variables are only the relevant input variables, suggesting that the search domain becomes smaller. To verify the performance of the improved optimization model, we employ the random method and a single-population genetic algorithm to generate test data.

RQ3: **To what extent does grouping and multitasking improve the efficiency of mutation-based test data generation?**
We employ **MGA** to solve the multi-task optimization problem of test data generation in Eq. 3. As a comparison, for the ungrouped mutation branches, **SGA** (single population genetic algorithm) is used to generate test data.

B. EXPERIMENTAL SETUP

The configuration of PC in the experiments is as follows: 2*Intel(R) Core(TM) i5 CPU, 4GB memory, Microsoft Windows 7, and VC++.

1) OBJECT PROGRAMS

We select eight benchmark and industrial programs as the object programs under test, which are written in C language. The scale, data type, structure, and function of the programs under test are diverse and wide. Table 1 lists the

TABLE 1. Information about programs under test.

ID	Programs	*No.* of inputs	lines of code	Function
G1	Triangle	3	35	Triangular classification
G2	Day	3	42	Calculate the order of day
G3	Totinfo	7	319	Information statistics
G4	TIFF	14	182	Tag Image File Format
G5	F2	17	418	Engine defroster
G6	Defroster	20	250	Rear window defroster
G7	Replace	$m+n+p$; $m,n,p \in \{0,1,...,5\}$	564	Pattern matching
G8	Space	236	9564	Array language interpreter
Sum	-	315	11374	-

main information about these programs. Among them, *G*1 and *G*2 are the standard benchmark programs, and have been widely used by many scholars [23], [25], [31], [33]. *G*4 manipulates the images in the Tag Image File Format (TIFF) [25], [29]. *G*5 and *G*6 are industrial case studies provided by DaimlerChrysler, which are production code for the engine and rear window defroster embedded control systems [25], [29]. *G*7 and *G*8 are created by European Space Agency [23], [29], [35], and *G*3 is created by Siemens [23], [25]. They are available from the Software-artifact Infrastructure Repository.

2) GENERATION OF MUTANT BRANCHES

We select around 30% of the statements from each program to generate mutant branches, which are summarized on Column 2 of Table 2. These statements are selected in terms of such factors as the structure complexity, the statement type, the lines of code, and the nesting depth [36].

Next, we seed faults by mutation operators on the selected statements. Offutt and King [38] proposed 22 classes of mutation operators for C programs. In the experiment, we choose 13 classes of mutation operators, *ABS, AOR, CAR, CRP, CSR, LCR, ROR, RSR, SCR, SAR, SRC, SVR*, and *UOI*, and their details refers to [1], [38], because the other 9 classes are not suitable for the programs under test, for example, mutation operator, *SDL* (statement deletion).

Further, we manually generate mutants while determining whether they are equivalent or redundant. Although our method is artificial, it is not arbitrary, but follows a procedure set in advance. There have been various studies concerning how to detect redundant or equivalent mutants [35]. Moreover, the redundant mutants should be removed [37]. Our previous research [12], [33], [35] discovered that the mutants generated in the same location are likely to have the subsumed relations. In addition, some mutants produced by some mutation operators are redundant.

After obtaining the non-equivalent mutants, we construct the corresponding mutant branches, listed in Column 4 of Table 2. Note that some of these mutant branches are not necessary to apply the proposed method, because they are easily killed. To remove these unwanted mutants, we use some test data generated by the random method to kill the non-equivalent mutants, then obtained the alive mutants as tested mutant branches. The number of tested mutant branches are shown in Column 5 of Table 2. According to the number of mutant branches, *G*1 – *G*2, *G*3 – *G*7, and *G*8 are labeled as small-, medium- and large-scale programs. The sizes of randomly generated test data corresponding to these three types of programs are 150, 2000, and 12000, respectively.

C. EXPERIMENTATION PROCESS

1) RQ1: NECESSITY OF REMOVING IRRELEVANT INPUT VARIABLES

In the experiment, we first determine the relevance between the mutant branches and the input variables based on Subsection III.A. Then, we compare the number of the relevant input variables with the number of the input variables of the program, and calculate the reduction rate of input variables. The **reduction rate** (*RR*) refers to the number of irrelevant variables to the total number of input variables, i.e.

$$RR = \frac{\text{No. of irrelevant input variables}}{\text{No. of all input variables}} \tag{5}$$

Intuitively speaking, for a program under test, its input domain is certain. The larger *RR* is, the more irrelevant variables are, and the more the search domain is reduced, the more necessary to reduce irrelevant variables.

In Fig. 1 (b), the number of relevant input variables to M_1 ($x[3], x[4], x[6]$), is three, thus we can obtain $RR = \frac{6-3}{6} = 50\%$ for M_1, suggesting the search domain is reduced by half.

2) RQ2: PERFORMANCE OF THE IMPROVED OPTIMIZATION MODEL

In the experiment, the random method and the single-population genetic algorithm solving the traditional optimization model are denoted as **RDtra** and **SGAtra**, respectively, and the two methods based on the improved optimization model are denoted as **RD** and **SGA**, respectively. We verify the performance of the improved model, that is, the efficiency of test data generation after reducing the search domain. In terms of the success rate, mutation score, time consumption, number of iterations, we focus on comparing **RD** with **RDtra** and **SGA** with **SGAtra**, and by the way observe the advantages of the single-population genetic algorithms (**SGA** and **SGAtra**) over the random methods (**RD** and **RDtra**).

The **success rate** (*SR*) is the ratio of times of finding desired test data (success) to times of running an algorithm, expressed as

$$SR = \frac{\text{times of finding desired test data}}{\text{times of running an algorithm}} \tag{6}$$

Eq. 6 shows that the higher the success rate is, the better the corresponding algorithm is.

In the experiment, the success or failure of generating test data using **RDtra**, **RD**, **SGAtra**, or **SGA** is recorded, along with the time consumption and the number of iterations if the search was successful. Generally, the shorter the time and the fewer the number of iterations required, the better and more efficient the search is.

TABLE 2. Information about programs under test.

ID	$No.$ of statements selected	$No.$ of mutants	$No.$ of non-equivalent mutants	$No.$ of mutant branches tested	$Ave.No.$ of relevant input variables	$Ave.$ of (%) RR
G1	7	47	43	29	2.03	32.33
G2	6	31	27	17	1.20	60.00
G3	103	512	402	278	4.05	42.14
G4	60	290	249	183	6.06	56.71
G5	139	498	379	310	1.67	90.18
G6	83	321	310	211	6.21	68.95
G7	183	509	403	313	10.21	43.28
G8	2690	4403	3966	2901	138.22	41.43
Sum	-	3271	5779	4242	Ave.=21.21	Ave.=54.38

Note that due to the non-determinism of the experimental results, we independently run each algorithm 60 times to generate test data, and take the average of these results. Similarly, in the following experiments, the results of the proposed method and contrast methods are obtained in the same way.

Reflecting the degree of achievement of a test suite in detecting various types of faults, **mutation score** (MS) is defined as the ratio of the number of killed mutants to the number of all non-equivalent mutants, i.e.

$$MS = \frac{\text{No. of the killed mutants}}{\text{No. of the non-equivalent mutants}} \tag{7}$$

where MS are obtained under the criterion of weak mutation testing in this paper.

In the experiment, the parameters of four algorithms are set as follows. The number of test data generated by **RDtra** and **RD** is set 3000. In **SGAtra** and **SGA**, the genetic operations are roulette wheel selection, one-point crossover, and one-point mutation. The probabilities of crossover and mutation operations are 0.9 and 0.3, respectively. There are two termination criteria. One is that the desired test data are generated for all mutants. The other is that the population evolves for the maximum number of iterations, $g = 3000$.

3) RQ3: PERFORMANCE OF GROUPING AND MULTITASKING FOR THE MUTANT BRANCHES

In the experiment, we first group the mutant branches with the same relevant variables using Algorithm 1. Then, for the grouped mutant branches, **MGA** via individual sharing is employed to generate test data based on Algorithm 2. In contrast, we employ **SGA** (used by the above set of experiments) to generate test data for the ungrouped mutant branches.

Three metrics are used for answering RQ3. One is MS. The other metrics are time consumption and the number of iterations.

In the experiment, the sub-population size of **MGA** is $size = 5$, the other parameter settings and the genetic operators are the same as **SGA**. The settings of **SGA** refer to the above set of experiments.

V. EXPERIMENTAL RESULTS AND ANALYSIS

A. ANSWER TO RQ1

Column 6 in Table 2 lists the average numbers of relevant input variables for each program. Column 7 lists the average values of reduction rate (RR), which are obtained based on Column 3 in Table 1 and Column 6 in Table 2. We can see that, from Column 7, the largest RR is 90% corresponding to $G5$ and the smallest is 32.33% corresponding to $G1$, and the average value of RR is 54.38% for all mutant branches of eight programs, suggesting that the mutant branches in the different programs have the different values of RR, and more than half of input variables are irrelevant.

This set of experiments shows that it is necessary to remove the irrelevant input variables for most mutant branches, which helps to improve the efficiency of generating test data by reducing the search domain.

B. ANSWER TO RQ2

Table 3 lists the values of success rate, SR, of the random methods (**RDtra** and **RD**) and the single population genetic algorithms (**SGAtr** and **SGA**) when generating test data using, where "Mean" and "Max" represent the mean and maximum values of SR for all mutant branches of a program. The minimum value of SR is 0, indicating that the mutant is not been killed after executing the algorithm 60 times. Given that the display with the minimum value (0) of each program is meaningless for comparison, we select the second minimum value of SR, denote as "2nd Min" in Table 3, which represents the minimum value greater than 0.

Columns 2–7 of Table 3 summarize the values of SR using the random method (**RDtra** and **RD**) in terms of "2nd Min", "Max", and "Mean", we discover that the differences between their values are relatively small. For example, "Mean" of SR using these two methods are 48.13% and 51.88%, their difference is only 3.75%. It implies that, for the improved model, SR is unaffected by the removal of irrelevant input variables. This is probably because the random algorithm randomly samples in the search domain. Even if the values of irrelevant variables are removed, the values of relevant variables are still highly uncertain in the reduced the search domain, which makes no significant difference in SR when searching for desired test data.

Columns 8–13 of Table 3 summarize the values of SR using **SGA** and **SGAtra**. Their "Mean" are 85.14% and 89.79%, especially "Max" of **SGA** is 100% for $G1$–$G5$ and $G6$. These results provide evidence that **SGA** solving the improved model is better than **SGAtra** in RS, indicating that **SGA** helps to increase the effectiveness and efficiency of finding desired test data by reducing the search domain.

TABLE 3. Success rate by the different methods based on two models.

ID	RDtra (%)			RD (%)			SGAtra(%)			SGA (%)		
	2nd Min	Max	Mean	2nd Min	Max	Mean	2nd Min	Max	Mean	2nd Min	Max	Mean
$G1$	11.67	71.67	55.33	26.67	81.67	61.67	38.33	100	86.67	38.33	100	91.67
$G2$	15.00	75.00	55.00	26.67	81.67	65.00	26.67	100	88.33	61.67	100	93.33
$G3$	18.33	68.33	48.33	16.67	80.00	50.00	23.33	96.67	88.33	50.00	100	90.00
$G4$	18.33	73.33	48.33	20.00	80.00	48.33	38.33	98.33	81.67	51.67	100	90.00
$G5$	18.33	68.33	46.67	21.67	76.67	48.33	26.67	96.67	85.00	58.33	100	88.33
$G6$	13.33	71.67	45.00	21.67	71.67	46.67	31.67	93.33	83.33	56.67	96.67	88.33
$G7$	16.67	73.33	43.33	25.00	83.33	43.33	33.33	100	86.67	60.00	100	90.00
$G8$	10.00	78.33	45.00	20.00	81.67	51.67	38.33	96.67	85.00	58.33	96.67	86.67
Ave.	15.21	72.50	48.13	22.29	79.58	51.88	32.08	97.71	85.00	54.38	99.17	89.79

In addition, we can see from Table 3 that, in terms of ''2nd Min'', ''Max'', and ''Mean'', **SGA** and **SGAtra** are significantly better than **RDtra** and **RD**. Especially, in the average values of ''Mean'', **SGA** and **SGAtra** are 37.91% (89.79%–51.88 %) and 36.87 % (85.00 % – 48.13 %) more than **RD** and **RDtra**.

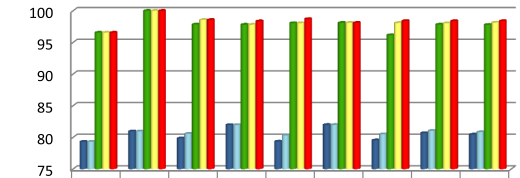
Given that the removal of irrelevant input variables does not have much impact on the performance of generating test data using the random methods, we focus on evaluating the performance of **SGAtra** and **SGA** in *SR*, the number of iterations and the time consumption. We employ the Mann-Whitney U Test based on the statistical tool, Spss. Let the significance level of the U test be 0.05.

Given that the ''Mean'' may not reveal the personalities of different mutant branches or may mask abnormal data, we select three kinds of representatives mutant branches to display in the limited space of this paper. Table 4 lists the Mann-Whitney U Test results of **SGA** comparing to **SGAtra** in three metrics for the representative mutants, where M_{i1}, M_{i2} and M_{i3} $(i = 1, 2, \cdots, 8)$ refer to the mutant that corresponds to ''Max'', the one closest to 'Mean'', and the one that corresponds to ''2nd Min''. Symbol ''+'' indicates that **SGA** is significantly better than **SGAtra**, and ''='' indicates that there is no significant difference when generating test data. In the last row of Table 4, ''The superior ratio'' refers to the number of ''+'' to the number of the experimental evaluation. For example, when evaluating 32 results of U test in terms of time consumption, 22 results show ''+'', that is, the superior ratio is $\frac{22}{32} = 68.75\%$.

Similarly, for 4242 tested mutant branches, we employ the Mann-Whitney U test to evaluate the performance of **SGA**. Fig. 4 shows the values of the superior ratio in terms of three metrics. In the last column of Fig. 4 (a–c), the average values of superior ratios are 83.9% (*SR*), 82.78% (the number of iterations) and 68.71% (the time consumption). It means that, for most of the mutant branches, **SGA** is significantly better than **SGAtra**.

The results of Table 4 and Fig. 4 provide evidence that, in terms of *SR*, the number of iterations and time consumption, for most of the mutant branches, **SGA** is better than **SGAtra**, and **SGA** is more significant for the hard-to-kill mutants (M_{i3}).

The values of *MS*, obtained by four methods, are shown as Fig. 3. As can been seen that the average values of *MS* based on **SGAtra** and **SGA** are 98.15 % and 97.80 %,



	G1	G2	G3	G4	G5	G6	G7	G8	Ave.
RDtra	79.31	80.95	79.86	81.97	79.35	81.99	79.55	80.70	80.46
RD	79.31	80.95	80.58	81.97	80.32	81.99	80.51	81.04	80.83
SGAtra	96.55	100	97.84	97.81	98.06	98.10	96.17	97.83	97.80
SGA	96.55	100	98.56	97.81	98.06	98.10	98.08	98.04	98.15
MGA	96.55	100	98.56	98.36	98.71	98.10	98.40	98.38	98.38

FIGURE 3. *MS* by the different methods based on two models.

respectively. They are significantly better than that of **RD** (80.46 %) and **RDtra** (80.83 %), suggesting that **SGA** and **SGAtra** are more capable in killing the mutants. Of course, their excellent performance is not related to the removal of irrelevant input variables, but the excellent search mechanism of genetic algorithm. In addition, rows 3 and 4 of Fig. 3 show that, in terms of the average values of *MS*, the difference of **SGA** and **SGAtr** is 0.35% (98.12%-97.80%), suggesting that **SGA** is higher than **SGAtra**, but their difference is small. It is because they have the same search mechanism based on genetic algorithms.

In summary, **SGA** has the highest *SR* and *MS*, the least time consumption and the number of iterations, suggesting that, when generating mutation-based test data, the performance that the genetic algorithms solve the improved model is significant.

C. ANSWER TO RQ3

Fig. 5 and 6 show the average values of the iterative number and time consumption obtained using **MGA** and **SGA** for the mutant branches of each program. In the average values of the iterative number, **MGA** (877.87) is 1.37 times **SGA** (1202.50). In time consumption, **MGA** (3.17s) saves 1.17 times more than **SGA** (6.88s).

To verify the significant superiority of **MGA**, we employ the Mann-Whitney U test to evaluate the results of the iterative number and time consumption. Let the significance level of the U test be 0.05. Fig. 7 shows the results of the superior ratio of **MGA** comparing to **SGA**. We can see that the average values of the superior ratios are 69.59% in the number of

TABLE 4. Mann-Whitney U Test results of SGA comparing to SGAtra in three metrics for the representative mutants.

ID	Mutants	Time consumption	$No.$ of iterations	SR
G1	M11	=	=	=
	M12	=	=	+
	M13	+	+	+
	Mean	+	=	+
G2	M21	=	=	=
	M22	=	=	+
	M23	+	+	+
	Mean	+	=	+
G3	M31	=	=	=
	M32	+	+	+
	M33	+	+	+
	Mean	+	+	+
G4	M41	=	=	=
	M42	+	=	+
	M43	+	+	+
	Mean	+	+	+
G5	M51	=	=	=
	M52	+	=	+
	M53	+	+	+
	Mean	+	=	+
G6	M61	=	=	=
	M62	+	+	+
	M63	+	+	+
	Mean	+	+	+
G7	M71	=	=	=
	M72	+	+	+
	M73	+	+	+
	Mean	+	+	+
G8	M81	=	=	=
	M82	+	+	+
	M83	+	+	+
	Mean	+	+	+
The superior ratio (%)	-	68.75	53.13	75

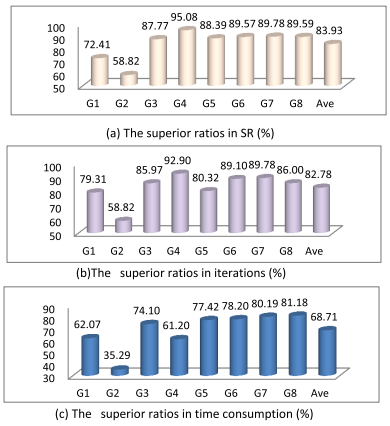


FIGURE 4. The superior ratios of SGA compareing to SGAtra in three metrics.

iterations and 65.08% in the time consumption, shown as the last column of Fig. 7 (a–b), suggesting that **MGA** is significantly superior to **SGA** for most mutant branches. Especially, for *G*8, the superior ratios are 82.42% (2391/2901) in the number of iterations and 82.21% (2385/2901) in the time consumption, suggesting that the more mutant branches are, the more significant **MGA** is. **MGA** is excellent because, at each iteration, **SGA** can generate only one test datum

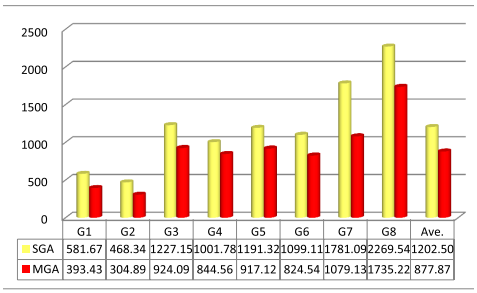


FIGURE 5. The number of iterations of MGA and SGA.

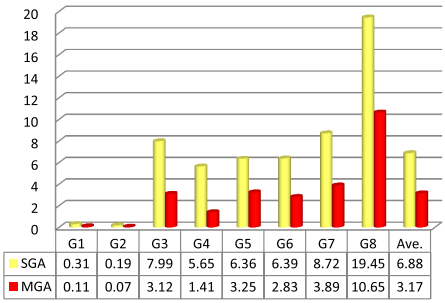


FIGURE 6. Time consumption of MGA and SGA (in second).

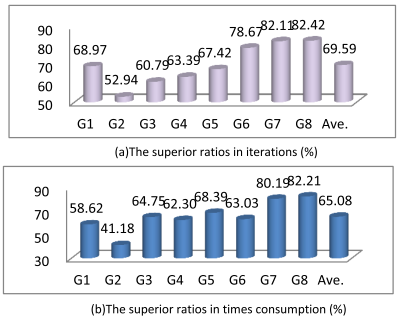


FIGURE 7. The superior ratio of MGA comparing to SGA in two metrics.

targeting at one mutant branch, whereas **MGA** can generate test data for more mutant branches by multitasking, the information sharing of the individuals can prompt a mutant branch to be executed by the multiple individuals of different sub-population when running **MGA**.

MS of **SGA** and **MGA** are listed in rows 4–5 of Fig. 3. For eight programs, *MS* of **SGA** and **MGA** are higher than those of the other three methods. The average values of *MS* of **SGA** and **MGA** are 98.15% and 98.38%, respectively, which indicates that **MGA** is slightly better than **SGA**, because they have the same mechanism for generating test data. Although **MGA** is not significantly better than **SGA**, its advantage should be improving the efficiency of generating test data, which can be verified by the results of Fig. 5–7.

In summary, **MGA** outperforms **SGA**, not only due to its higher *MS*, but also because of its shorter time and fewer iterations when generating test data. In other words, for a large number of mutant branches, the performance of generating test data by grouping and multitasking is significantly improved.

VI. THREATS TO VALIDITY

This section provides the main threats to validity of the proposed method and ways to alleviate these threats.

The first threat is the existence of equivalent mutants and redundant mutants. They can affect the quality of the test suite. To avoid human judgment bias for equivalent mutants, if a team member is not sure about whether a mutant is equivalent, we will discuss together. In this way, the determination of equivalent mutants is more accurate. In addition, non-equivalent mutants are also obtained via our previous methods [23], [35]. For the redundant mutants, they do not contribute during the testing process, although they are considered for calculating the mutation score [37]. As a result, the mutation score is inflated. To obtain the reasonable mutation score, we just used some rough methods to reduce the redundant mutants based on our previous research [12], [33], [35] and other literature [1], [5]. How to reducing the redundant mutants is not the focus of this paper, which will be involved in future studies.

The second threat comes from the cost of the static analysis of the relevance between the input variables and the mutant branches. To reduce cost, we adopt two strategies. One is that we choose a path that is easy to cover from multiple paths, and then only analyze the relevance between the variables involved in this path and a mutation branch. The other is that we remove the mutants that are easily killed, and only analyze the relevance between the hard-to-kill mutants and the variables. In these ways, the cost of the proposed method should be reduced. In the future, we will study some automatic methods to determine relevance, which helps to improve the efficiency of the proposed method.

VII. CONCLUSION

Mutation testing is an important method for evaluating the quality of a test suite. However, the cost of mutation testing is relatively high, which greatly reduces its application in real software engineering. Previous research has shown that many factors affect the efficiency of test data generation, such as a large number of mutants, the diversity of mutant operators, and the huge domain formed by the input variables. In this paper, we focus on improving the efficiency of test data generation by reducing the search domain and multitasking.

We evaluated the performance of the proposed method on eight programs from different application domains and with various scales. The experimental results show that (1) for most mutant branches, there are many irrelevant variables, suggesting it is necessary to remove irrelevant variables; (2) taking relevant input variables as decision variables in the improved model increases the efficiency of mutation-based test data generation; (3) the efficiency of test data generation by grouping mutant branch using **MGA** is improved significantly.

Although the proposed method has a higher performance in generating mutation-based test data, the cost of the static analysis we used may be relatively large, especially for complex programs. We need to further explore the characteristics of the mutants, and group the mutants under other criteria to promote the efficiency of killing mutants. In addition, in this paper, we determine the relevance between the input variables and the mutants under the weak mutation testing. For strong mutation testing, an incorrect state should be propagated to the output of a program when a test datum is employed to execute the original program and its mutant. Therefore, we need to design a better strategy to determine their relevance under strong mutation testing.

REFERENCES

[1] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 649–678, Sep. 2011.

[2] X. Yao, D. Gong, B. Li, X. Dang, and G. Zhang, "Testing method for software with randomness using genetic algorithm," *IEEE Trans. Softw. Eng.*, vol. 8, pp. 61999–62010, 2020.

[3] R. G. Hamlet, "Testing programs with the aid of a compiler," *IEEE Trans. Softw. Eng.*, vol. SE-3, no. 4, pp. 279–290, Jul. 1977.

[4] 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.

[5] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, "Mutation testing advances: An analysis and survey," *Adv. Comput.*, vol. 112, pp. 275–378, Jan. 2019.

[6] M. Kintis, M. Papadakis, and N. Malevris, "Employing second-order mutation for isolating first-order equivalent mutants," in *Proc. Annu. ACM Symp. Appl. Comput.*, vol. 25, nos. 5–7, pp. 508–535, Aug. 2015.

[7] M. Sridharan and A. S. Namin, "Prioritizing mutation operators based on importance sampling," in *Proc. IEEE 21st Int. Symp. Softw. Rel. Eng.*, Nov. 2010, pp. 378–387.

[8] M. Harman, Y. Jia, and W. B. Langdon, "Strong higher order mutation-based test data generation," in *Proc. 19th ACM SIGSOFT 13th FSE Conf. Eur.*, 2011, pp. 212–222.

[9] 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, Dec. 2011.

[10] B. Lindström and M. Märki, "On strong mutation and subsuming mutants," in *Proc. 9th IEEE Int. ICSTW Conf.*, Apr. 2016, pp. 112–121.

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

[12] D. Gong, G. Zhang, X. Yao, and F. Meng, "Mutant reduction based on dominance relation for weak mutation testing," *Inf. Softw. Technol.*, vol. 81, pp. 82–96, Jan. 2017.

[13] J. Zhang, L. Zhang, M. Harman, D. Hao, Y. Jia, and L. Zhang, "Predictive mutation testing," *IEEE Trans. Softw. Eng.*, vol. 45, no. 9, pp. 898–918, Sep. 2019.

[14] Y.-S. Ma and S.-W. Kim, "Mutation testing cost reduction by clustering overlapped mutants," *J. Syst. Softw.*, vol. 115, no. 2, pp. 18–30, May 2016.

[15] M. B. Bashir and A. Nadeem, "Improved genetic algorithm to reduce mutation testing cost," *IEEE Access*, vol. 5, pp. 3657–3674, 2017.

[16] M. Papadakis and Y. Le Traon, "Mutation testing strategies using mutant classification," in *Proc. 28th Annu. ACM Symp. Appl. Comput.*, 2013, vol. 25, no. 5, pp. 572–604.

[17] A. S. Namin, X. Xue, O. Rosas, and P. Sharma, "MuRanker: A mutant ranking tool," *Softw. Test., Verification Rel.*, vol. 25, nos. 5–7, pp. 572–604, Aug. 2015.

[18] M. Kintis, M. Papadakis, and A. Papadopoulos, "How effective are mutation testing tools? An empirical analysis of java mutation testing tools with manual analysis and real faults," *Empirical Softw. Eng.*, vol. 23, no. 4, pp. 2426–2463, 2017.

[19] A. Kiran, W. H. Butt, M. W. Anwar, F. Azam, and B. Maqbool, "A comprehensive investigation of modern test suite optimization trends, tools and techniques," *IEEE Access*, vol. 7, pp. 89093–89117, 2019.

[20] R. A. DeMilli and A. J. Offutt, "Constraint-based automatic test data generation," *IEEE Trans. Softw. Eng.*, vol. 17, no. 9, pp. 900–910, 2002.

[21] M. Papadakis and N. Malevris, "Automatic mutation test case generation via dynamic symbolic execution," in *Proc. 21st IEEE Int. ISSRE Conf.*, Nov. 2010, pp. 121–130.

[22] F. C. M. Souza, M. Papadakis, Y. Le Traon, and M. E. Delamaro, "Strong mutation-based test data generation using hill climbing," in *Proc. 9th Int. Workshop Search-Based Softw. Test.*, 2016, pp. 45–54.
[23] X. Dang, X. Yao, D. Gong, and T. Tian, "Efficiently generating test data to kill stubborn mutants by dynamically reducing the search domain," *IEEE Trans. Rel.*, vol. 69, no. 1, pp. 334–348, Mar. 2020.
[24] Y. Zhang and D.-W. Gong, "Evolutionary generation of test data for paths coverage based on scarce data capturing," *Chin. J. Comput.*, vol. 36, no. 12, pp. 2429–2440, Mar. 2014.
[25] X. J. Yao and D. W. Gong, "Genetic algorithm-based test data generation for multiple paths via individual sharing," *Comput. Intell. Neurosci.*, vol. 29, pp. 1–13, Jan. 2014.
[26] M. Masud, A. Nayak, and M. Zaman, "Strategy for mutation testing using genetic algorithms," in *Proc. Can. Conf. Electr. Comput. Eng. (ICCSCEI)*, 2005, pp. 1049–1052.
[27] A. Bajaj and O. P. Sangwan, "A systematic literature review of test case prioritization using genetic algorithms," *IEEE Access*, vol. 7, pp. 126355–126375, 2019.
[28] T. Tian, D. Gong, F.-C. Kuo, and H. Liu, "Genetic algorithm based test data generation for MPI parallel programs with blocking communication," *J. Syst. Softw.*, vol. 155, pp. 130–144, Sep. 2019.
[29] P. Mcminn, M. Harman, K. Lakhotia, Y. Hassoun, and J. Wegener, "Input domain reduction through irrelevant variable removal and its effect on local, global, and hybrid search-based structural test data generation," *IEEE Trans. Softw. Eng.*, vol. 38, no. 2, pp. 453–477, Mar. 2012.
[30] M. Harman, P. McMinn, and J. Wegener, "The impact of input domain reduction on search–based test data generation," in *Proc. Found. Softw. Eng.*, 2007, pp. 155–164.
[31] Y. Zhang and D. W. Gong, "Evolutionary generation of test data for path coverage based on automatic reduction of search space," *Acta Electron. Sinica*, vol. 40, no. 5, pp. 1011–1016, 2012.
[32] G. J. Zhang, D. W. Gong, and X. J. Yao, "Test case generation based on mutation analysis and set volution," *Chin. J. Comput.*, vol. 38, no. 11, pp. 2318–2331, 2015.
[33] X. Y. Dang, D. W. Gong, and X. J. Yao, "Feasible path generation of weak mutation testing based on statistical analysis," *Chin. J. Comput. Sci.*, vol. 39, no. 11, pp. 2355–2371, 2016.
[34] R. W. Moore, R. B. Kearfott, and M. J. Cloud, *Introduction to Interval Analysis*. Philadelphia, PA, USA: SIAM, 2009.
[35] 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.*, 2014, pp. 919–930.
[36] D. W. Gong, B. Qin, and T. Tian, "Selecting objects to be mutated based on statement importance," *Acta Electron. Sinica*, vol. 45, no. 6, pp. 1518–1522, 2017.
[37] M. Papadakis, C. Henard, M. Harman, Y. Jia, and Y. Le Traon, "Threats to the validity of mutation-based test assessment," in *Proc. 25th Int. Symp. Softw. Test. Anal.*, 2016, pp. 354–365.
[38] A. J. Offutt and K. N. King, "A fortran 77 interpreter for mutation analysis," *ACM SIGPLAN Notices*, vol. 22, no. 7, pp. 177–188, Jul. 1987.

XIANGYING DANG received the M.S. degree in computer science from the School of Information Engineering, Jiangnan University, in 2008. She is currently pursuing the Ph.D. degree with the School of Information and Electronic Engineering, China University of Mining and Technology. Her research interests include software engineering-based search and mutation testing and analysis.

XIANGJUAN YAO received the Ph.D. degree in control theory and control engineering from the China University of Mining and Technology, in 2011.

She is currently a Professor and a Ph.D. Advisor with the School of Mathematics, China University of Mining and Technology. Her main research interests include search-based software testing and evolutionary computation.

DUNWEI GONG (Member, IEEE) received the Ph.D. degree in control theory and control engineering from the China University of Mining and Technology, in 1999.

He is currently a Professor and a Ph.D. Advisor with the School of Information and Electrical Engineering, China University of Mining and Technology. His main research interests include evolutionary computation, intelligence optimization, and data mining.

TIAN TIAN received the Ph.D. degree in control theory and control engineering from the China University of Mining and Technology, Xuzhou, China, in 2014. She is currently an Associate Professor with the School of Computer Science and Technology, Shandong Jianzhu University. Her current research interest includes parallel program testing.

BAICAI SUN received the M.S. degree in control engineering from Qufu Normal University, in 2016. He is currently pursuing the Ph.D. degree with the School of Information and Control Engineering, China University of Mining and Technology. His research interests include search-based software engineering, surrogate-assisted evolutionary optimization, and machine learning.