Hindawi
Scientific Programming
Volume 2021, Article ID 7543896, 10 pages
https://doi.org/10.1155/2021/7543896

Research Article

ImReMuDF: Redundant Mutants Identification Method Based on Definition and Reference of Variables

Zhenpeng Liu,[1,2] Xianwei Yang,[1] Yi Liu,[1] Yonggang Zhao ,[3] and Xiaofei Li[2]

[1]*School of Cyberspace Security and Computer, Hebei University, Baoding, China*
[2]*Information Technology Center, Hebei University, Baoding, China*
[3]*School of Management Engineering and Business, Hebei University of Engineering, Handan, China*

Correspondence should be addressed to Yonggang Zhao; ygzhao@hbu.edu.cn

Received 16 June 2021; Revised 2 October 2021; Accepted 6 October 2021; Published 21 October 2021

Academic Editor: Roberto Natella

Mutation testing is an effective defect-based software testing method, but a large number of mutants lead to expensive testing costs, which hinders the application of variation testing in industrial engineering. To solve this problem and enable mutation testing to be applied in industrial engineering, this paper improves the method of identifying redundant mutants based on data flow analysis and proposes the inclusion relationship between redundant mutants, so that the redundancy rate of mutants is reduced. In turn, the cost of mutation testing can be reduced. The redundant mutants identification method based on definition and reference of variables (ImReMuDF) was validated and evaluated using 8 C programs. The minimum improvement in redundant mutant identification rate was 34.0%, and the maximum improvement was 71.3% in the 8 C programs tested, and the verification results showed that the method is feasible and effective and has been improved in reducing redundant mutants and effectively reducing the execution time of mutation testing.

1. Introduction

Mutation testing is a fault-based software testing technique [1] that has received a great deal of attention in program analysis, defect detection, and test case generation [2]. Mutation testing not only has the advantages of strong troubleshooting ability, convenience, and flexibility, but also can be used to expose defects in the software, and it has the ability to measure the error found in the test data set and evaluate the adequacy of the test [3]. Mutation testing has strong fault detection capabilities [4]. Mutation testing technology has gradually become popular in the industry [5] but has not been widely used. The reason is that running and analyzing tests are expensive in terms of resources and manpower [6], the computational cost of mutation testing is high, the identification of equivalent mutations is difficult, the effective automated mutation testing tools are not perfect, etc. [7].

The concept of mutation testing was first proposed by Demillo [8], which refers to the execution of mutation operations on the original program to generate a new program (program with defects), and the new program is called a mutant. Subsequently, a large number of scientific research results appeared. In terms of reducing the number of mutants, Wong [9] proposed random selection of mutants to achieve the purpose of reducing the number of mutants. Sridharan and Namin [10] proposed the preferential selection of more informative mutation operators. Usaola et al. [11] proposed the minimization of test cases to reduce the time of mutation testing. Sun et al. [12] proposed a method for identifying redundant mutants based on data flow analysis. Chekam et al. [13, 14] proposed a dynamic symbolic execution method and a mutant priority method. Shomali and Arasteh [15] proposed the firefly optimization algorithm as a heuristic algorithm for identifying the most error-prone path in the program. Hooseini et al. [16] proposed a genetic algorithm to identify the path where the program is most likely to propagate errors as the mutation location. In terms of shortening the time of mutation testing, Krauser et al. [17] proposed parallel execution of mutants.

King and Offutt [18] proposed prioritization of mutation compilation. Although the above method optimizes the mutation testing from different aspects, it is not perfect in terms of adequacy assessment, and it still has a certain impact on adequacy. The method proposed by Sun et al. reduces the mutants caused by variables very well. It reduces the execution time of mutation testing by reducing the number of mutants, but it does not extend well to the identification of redundant mutants of multiple variables.

In order to increase the recognition rate of redundant mutants and reduce redundant mutants, this paper proposes a redundant mutants identification method based on definition and reference of variables (ImReMuDF). The main contributions of this paper are as follows:

(1) The definition and reference of two variables are proposed to increase the recognition rate of redundant mutants in the test.

(2) In the definition and reference of two variables, the inclusion relationship of the variables in the same situation is introduced. This method improves the identification of redundant mutants caused by variables, thereby achieving the purpose of reducing the number of mutants.

(3) In 8 C program experiments, the effectiveness and feasibility of this method are verified.

Compared with the definition and reference of one variable, the ImReMuDF method can be extended to the definition and reference of multiple variables to identify redundant mutants.

2. Identification of Redundant Mutants

2.1. Traditional Mutation Testing. In traditional mutation testing, errors are embedded into the program through mutant operators [19] or manually, and then the embedded defects are detected by test cases, and the adequacy of defect detection is determined by the mutation score. Given the tested program **P** and the set of test cases **T**, the tested program **P** uses mutation operators or manual embedding errors to generate a set of mutants *M*. It is very important to identify equivalent mutants and nonequivalent mutants in the mutation testing [20]. The traditional mutation testing process is shown in Figure 1. The traditional mutation testing process is as follows:

(1) The set of mutants identifies and classifies the set of equivalent mutants **I** and the set of nonequivalent mutants **L**

(2) Run the set of test cases $T(T = T \cup \{t\})$ on the set **L**

(3) Calculate the mutation score for the set of test cases **T**

(4) If the mutation testing meets the requirements, exit; if the mutation testing does not meet the requirements, add a new test case ***t*** and repeat the (2)-(3) operations

2.2. Redundant Mutant. In mutation testing, the adequacy of the test case set is evaluated by the ratio of the number of detected mutants to the total mutants. In mutation testing, a large number of redundant mutants bring a large number of repeated checks to the mutation testing. On the one hand, it increases the test time, and on the other hand, it brings expensive test costs. Therefore, a large number of redundant mutants must be removed before mutation testing. The removed mutants will not be repeatedly tested, reducing the time of mutation testing, to achieve the purpose of saving test costs.

The redundant mutants are shown in Figure 2. In Figure 2, it can be found that the mutants m_1 and m_2 mutate in the 3rd and 4th rows, respectively. Although the position of the mutation has changed, the program status below the mutation location is similar. In the mutation testing, if there is a test case that can execute the above code, it must be able to kill mutants m_1 and m_2. In the calculation of the mutation score, since whether the mutant can be killed can be obtained by the test result of the mutant m_2, only the mutant m_1 needs to be executed during the mutation testing, and the mutant m_2 is not executed.

Given the program **P** and the mutant m_1 generated by the mutation operator, if the output results of the program **P** and the mutant m_1 are not equal when running on the test case **t**, then the mutation testing is called strong mutation testing. Given the program **P** and the mutant m_2 generated by the mutation operator, if the state of the program **P** and the mutant m_2 is inconsistent when running on the test case **t**, then the mutation testing is called weak mutation testing. Compared with strong mutation testing, weak mutation testing has the advantage of shortening the test time. As can be seen from the above description, traditional mutation testing is based on strong mutation testing. This paper is to optimize the execution time of the mutation testing based on the weak mutation testing.

Definition 1 **(inclusive relationship between mutants).** Given the program **P** to be tested and an input $t = (t = \{t_{i(i=1,2,\ldots,n)}\})$ containing **n**-tuples, all possible **t** forms the input space **T** of the program **P**. Suppose there are two mutants m_1 and m_2 generated by **P**, and the output states of **P** and **m** when the input is **t** are $\mathbf{P_t}$ and $\mathbf{m_t}$, respectively; the mutant m_1 including the mutant m_2 must meet the two following conditions:

(1) $\exists t \in T,\ p_t \neq (m_1)_t \wedge p_t \neq (m_2)_t$

(2) $T(m_1) \subseteq T(m_2), T(m) = \{t \in T | p_t \neq m_t\}$

Then, it can be expressed as $m_1 \mapsto m_2$.

Based on the above analysis, when $T(m_1) \subseteq T(m_2)$, $T(m) = \{t \in T | p_t \neq m_t\}$ exists between the mutants, and m_1 and m_2 meet Definition 1, m_2 is called the redundant mutant of m_1. Assuming that there are sets of mutants M_1 and M_2 satisfying $\forall m_i \in M_2,\ \exists m_j \in M_1$, where $m_i \in m_j$, then M_2 is called the set of redundant mutants, and M_1 is the set of nonredundant mutants.

2.3. Data Flow Analysis. Data flow analysis is a technology used during compilation. And data flow analysis plays an important role in program analysis, compilation optimization, program verification, defect detection, etc. [2].

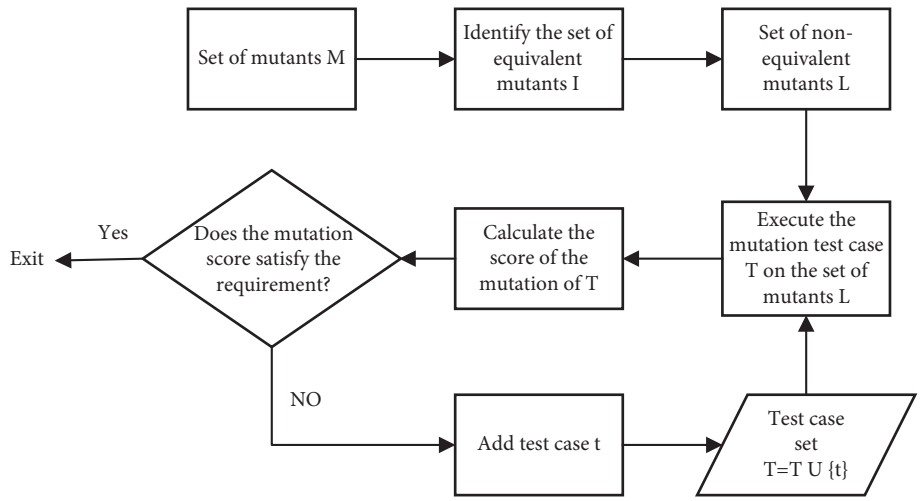

Figure 1: Traditional mutation testing process.

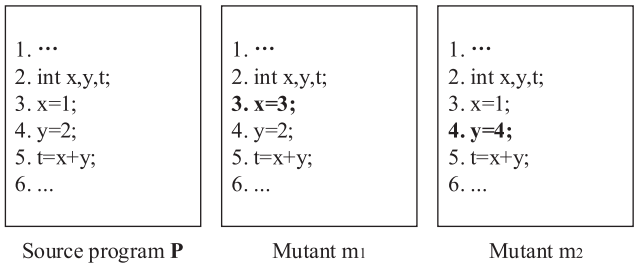

Figure 2: Examples of redundant mutant.

Data flow analysis mainly focuses on the data flow or possible values on the program execution path. Its purpose is to determine the relationship between variable definitions and references within a certain range of the program [21]. Therefore, the following two definitions are given.

Definition 2 (definition set and reference set). Definable set: in the program **P**, if **x** exists in the assignment statement **y**, and the variable value of **x** changes, then **x** is an assignment variable, which can be expressed as

$$\text{def}(y) = \{x \mid x \text{ is the assignment variable in the y statement}\}. \quad (1)$$

Reference set: in the program **P**, if $\boldsymbol{x}$ exists in the assignment statement $\boldsymbol{y}$, and the variable value of $\boldsymbol{x}$ does not change, then $\boldsymbol{x}$ is a reference variable, which can be expressed as

$$\text{ref}(y) = \{x \mid x \text{ is the reference variable in the } y \text{ statement}\}. \quad (2)$$

Definition 3 **(reference chain).** In PDG [22], there is a path from $\text{def}(u, v, s_1)$ to $\text{ref}(u, v, s_2)$, and there is no other path from $\boldsymbol{s_1}$ to $\boldsymbol{s_2}$, then there is a definition-reference Chain of variables u and v, denoted as $dr(u, v, s_1, s_2)$.

2.4. Improved Redundant Mutant Identification Steps. When performing redundant mutant detection on the mutant set M, firstly, a series of information about each mutant is obtained; then, after the identification rules, the redundant mutants in the mutant set are identified; finally, the redundant mutants are removed from the mutant set, and a new mutant set is output. The redundant mutant process of data flow analysis is as follows:

(1) The program structure of the program **P** to be tested is analyzed, and the block rule file of the program is obtained

(2) Compare the mutant program and the source program one by one to obtain the mutation location of the mutants

(3) Determine the class of the mutant according to the block rule file

(4) Analyze the data flow of the source program, and generate data flow information such as variable definition, variable reference, and definition-reference chain;

(5) Identify the set of redundant mutants according to the redundant mutant identification rules and combining the block categories of the mutants and information of the program data flow

(6) Output the set of mutants

The process of identifying redundant mutants based on data flow analysis is shown in Figure 3.

In redundant mutant based on data flow analysis, program structure analysis and mutation location analysis are applied to mutation testing tools [23]. In this paper, the data flow analysis is implemented using Frama-C [24].

Time complexity analysis of the algorithm: suppose that the number of mutants of the program to be tested is n, and the number of definition-reference chains is m. In the algorithm, each definition-reference chain needs to traverse the entire set of mutants, so the time complexity of obtaining the set of definition mutants and reference mutants is $O(n)$;

when the definition-reference chain meets the identification rules, it is necessary to traverse entire mutants in the set of reference mutant of the variable, so the time complexity is $O(m)$. Therefore, the time complexity of the algorithm is $O(n \times m)$.

3. Identification Rules of Redundant Mutant

When mutants undergo static analysis, the following three aspects need to be qualified: (1) the set of mutants with similar mutation locations; (2) the set of mutants with the same predecessor path conditions; (3) the set of mutants with similar program states (the state of the program after the mutation location). In the case of redundant mutant identification, to ensure the existence of a reachable path from variable definition to use between the source mutant and the mutation location of the redundant mutant, the solution is to set the analysis location after the mutant is used. Based on the above principles, the redundant mutant identification rules are defined in 4 dimensions (intrablock, sequential block, subfast, and intermodule) in the technique of data flow analysis.

3.1. Identification Rule 1 (D_1). Suppose that there is a definition-reference chain $dr(u, v, s_1, s_2)$ in the program, let mutant $m_{i=(1,2,3)}$ be the definition mutant set $M(\text{def}, u, v, s_1)$ of variable **u, v,** and let mutant $m_{j=(1,2,3)}$ be the reference mutant set $M(\text{ref}, u, v, s_2)$ of variable **u, v**. When $\mathbf{s_1}$ and $\mathbf{s_2}$ are in the same basic block and in the definition mutant set, $m_1 \mapsto m_2, m_3$. It can be concluded that $m_{x=(2,3,\ldots,6)}$ is a redundant mutant of m_1.

According to the identification rule $\mathbf{D_1}$ of the definition, it is qualified that its source variant m_1 and redundant mutants $m_{x=(2,3,\ldots,6)}$ belong to the same basic block. $m_{i=(1,2,3)}$ and $m_{j=(1,2,3)}$ are defined by the variable definability appearing in the $\boldsymbol{s_1}$ statement ($m_{x=(2,3,\ldots,6)}$) and the variable reference line appearing in the $\mathbf{s_2}$ statement (ref, u, v, s_2), respectively, where the $\mathbf{s_1}$ statement and the $\mathbf{s_2}$ statement belong to the $\mathbf{BasicBlock_n}$. From the data flow analysis technique, we can know that if the execution of $\mathbf{s_1}$ in $\mathbf{m_i}$ is triggered in the mutation test, it can cause the execution of $\mathbf{s_2}$ in $\mathbf{m_j}$, and then the definition-reference company can ensure the propagation of the state at $\mathbf{s_1}$ to $\mathbf{s_2}$. Based on the inclusion relationship between mutants and the definition of redundant mutant, $m_{x=(2,3,\ldots,6)}$ can be identified as a redundant mutant of m_1. Figure 4 shows an example of the $\mathbf{D_1}$ identification rule application.

In the mutants $m_{i=(1,2,3)}$, the variables x and y are defined in the 4th and 5th lines $(\text{def}(x, y, 4, 5))$, and in the mutants $m_{j=(4,5,6)}$, the variables x and y are referenced in the 6th line $(\text{ref}(x, y, 6))$. In the mutants $m_{i=(1,2,3)}$, the changes in the value of the variable in the definition mutant set $M(\text{def}, x, y, 4, 5)$ of the variables x and y will cause the value of the variable *temp* in the 6th line of the program to change. In the mutants $m_{j=(4,5,6)}$, the reference mutant set $M(\text{ref}, x, y, 6)$ of the variables x and y has a direct effect on the value of the variable *temp* in the program. Due to the definition-reference chain $dr(x, y, 4, 5, 6)$, the 6 mutants entirely have the same change state in the 6th row; that is, $m_{x=(2,3,\ldots,6)}$ are redundant mutants of m_1.

3.2. Identification Rule 2 (D_2). Assuming the existence of a definition-reference chain $dr(u, v, s_1, s_2)$, let mutant $m_{i=(1,2,3)}$ be the definition mutation set $M(\text{def}, u, v, s_1)$ of variable **u**, **v**, and let mutant $m_{j=(1,2,3)}$ be the reference mutation set $M(\text{ref}, u, v, s_2)$ of variable **u**, **v**. $\mathbf{s_1}$, belongs to the program block $\mathbf{b_n}$ ($\mathbf{b_n}$ is one of the BasicBlock, OptiomBlock, and LoopBlock), and $\mathbf{s_2}$ belongs to $\mathbf{BasicBlock_i}$, where $\mathbf{b_n}$ and $\mathbf{BasicBlock_i}$ satisfy the sequential block relationship and meet Definition 1 in $\mathbf{s_1}$. It can be concluded that $m_{x=(2,3,\ldots,6)}$ is a redundant mutant of m_1.

According to the defined identification rule $\mathbf{D_2}$, it is qualified that its mutant $m_{i=(1,2,3)}$ and mutant $m_{j=(4,5,6)}$ belong to the sequential block relationship. $m_{i=(1,2,3)}$ and $m_{j=(4,5,6)}$ are variables defined, respectively, in the $\boldsymbol{s_2}$ statement (def, u, v, s_1) and referenced in the $\mathbf{s_2}$ statement (ref, u, v, s_2), where $\boldsymbol{s_1}$ belongs to program block $\mathbf{b_n}$ ($\mathbf{b_n}$ is one of the BasicBlock, OptiomBlock, and LoopBlock) and $\mathbf{s_2}$ belongs to $\mathbf{BasicBlock_i}$. According to the definition of the sequential block, in the mutation testing, if the execution condition of $\mathbf{s_1}$ in $\mathbf{m_i}$ is triggered, it will cause the execution of $\mathbf{s_2}$ in $\mathbf{m_j}$. Then, the definition-reference chain of the variable can ensure that the state at $\mathbf{s_1}$ is propagated to $\mathbf{s_2}$. Based on the inclusion relationship between mutants and the definition of redundant mutants, $m_{x=(2,3,\ldots,6)}$ can be identified as redundant mutants of m_1. An example of the recognition rule is shown in Figure 4 below. Figure 5 shows an example of the $\mathbf{D_2}$ identification rule application.

The source program in Figure 5 is divided into blocks. It can be seen that lines 3, 4, and 5 belong to the same **BasicBlock** ($\mathbf{b_1}$), lines 6, 7, and 8 belong to the same **BasicBlock** ($\mathbf{b_2}$), and line 9 belongs to the same **BasicBlock** ($\mathbf{b_3}$). The definitions of the variables x and y are in the 4th and 5th lines, expressed as $(\text{def}(x, y, 4, 5))$, and the references to the variables x and y are in the 9th line, expressed as $(\text{ref}(x, y, 9))$. $\mathbf{b_2}$ is the sequential block of $\mathbf{b_1}$, and $\mathbf{b_3}$ is the sequential block of $\mathbf{b_2}$, so $\mathbf{b_3}$ is the indirect block of $\mathbf{b_1}$. The mutants $m_{i=(1,2,3)}$ belong to the definition mutation set $M(\text{def}, x, y, 4, 5)$ of the variables x and y. If the values of the variables x and y change, the influence of the program is propagated to line 9 by reference to the variables x and y, thereby changing the value of the variable *temp*. The mutants $m_{j=(4,5,6)}$ belong to the reference mutation set $M(\text{ref}, x, y, 9)$ of the variables x and y, and the changes in the values of the variables x and y directly affect the value of the variable *temp*. From the definition-reference chain $dr(x, y, 4, 5, 9)$, it can be seen that the 6 mutants have the same change state in the 9th row; that is, $m_{x=(2,3,\ldots,6)}$ are redundant mutants of m_1.

3.3. Identification Rule 3 (D_3). Assuming the existence of a definition-reference chain $dr(u, v, s_1, s_2)$, let mutant $m_{i=(1,2,3)}$ be the definition mutation set $M(\text{def}, u, v, s_1)$ of variables **u**, **v**, and let mutant $m_{j=(4,5,6)}$ be the reference mutation set $M(\text{ref}, u, v, s_2)$ of variables **u**, **v**. When $\mathbf{s_1}$ belongs to program block $\mathbf{b_n}$ ($\mathbf{b_n}$ is one of the BasicBlock, OptiomBlock, and LoopBlock), and $\mathbf{s_2}$ belongs to

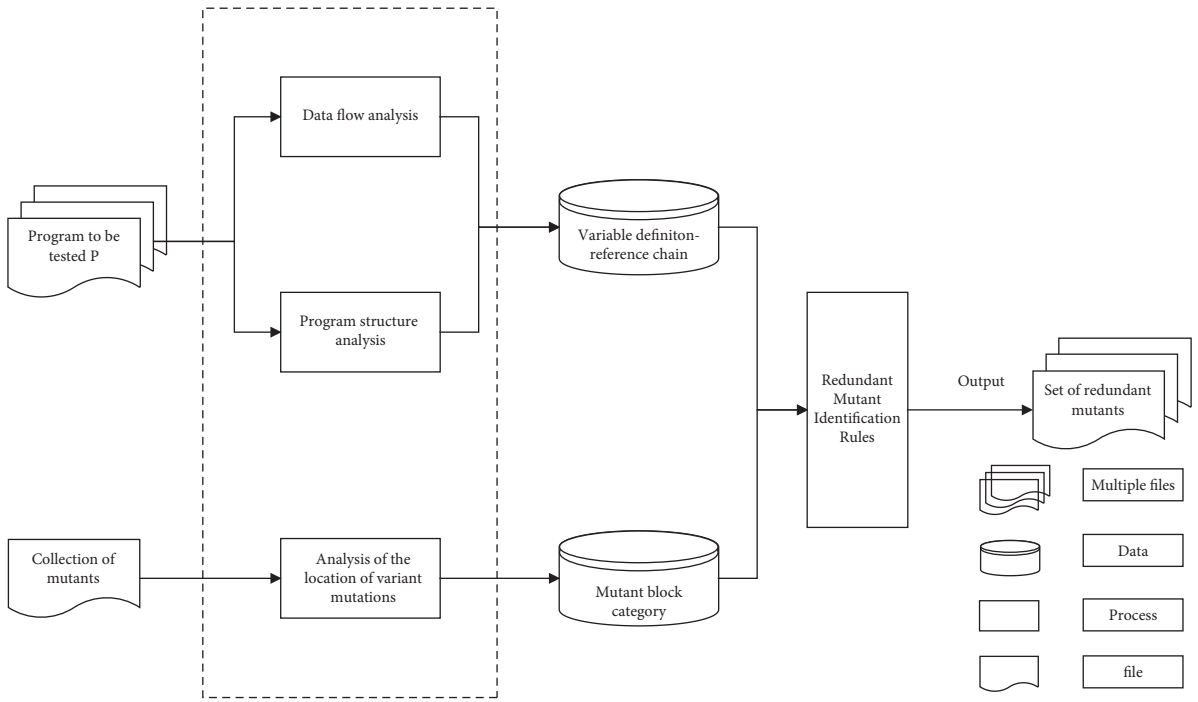

Figure 3: Redundant mutant identification by data flow analysis.

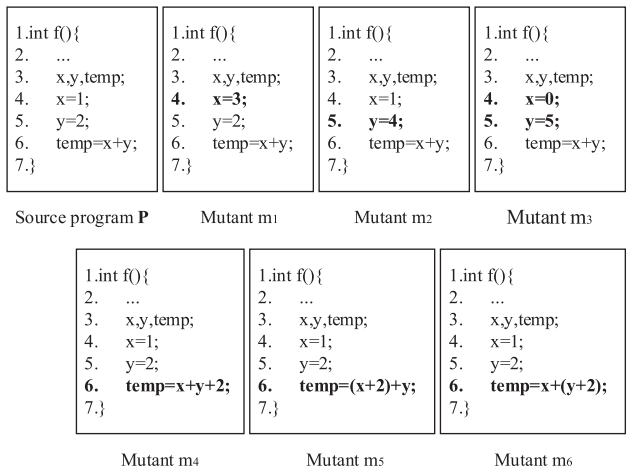

Figure 4: Example program for identifying rule D_1.

BasicBlock$_i$, **BasicBlock$_i$** is the sequential block of the subblock of $\mathbf{b_n}$ and meets Definition 1 in $\mathbf{s_1}$. It can be concluded that $m_{x=(2,3,\ldots,6)}$ are redundant mutants of m_1.

According to the identification rule $\mathbf{D_3}$, it is qualified that the relationship between the mutants $m_{i=(1,2,3)}$ and mutants $m_{j=(4,5,6)}$ is defined as a composite of the relationship of the subblock and the sequential block. $m_{i=(1,2,3)}$ and $m_{j=(4,5,6)}$ are variables defined, respectively, in the $\boldsymbol{s_2}$ statement $((\mathrm{def}, u, v, s_1))$ and referenced in the $\boldsymbol{s_2}$ statement $((\mathrm{ref}, u, v, s_2))$. $\boldsymbol{s_1}$ belongs to program block $\mathbf{b_n}$ ($\mathbf{b_n}$ is one of the BasicBlock, OptiomBlock, and LoopBlock) and $\mathbf{s_2}$ belongs to **BasicBlock$_i$**, where $\mathbf{b_n}$ and **BasicBlock$_n$** directly or indirectly satisfy the composite of the relationship of the subblock and the sequential block. It can be known from the subblock relationship that, in the mutation testing, if $\mathbf{b_n}$ is executed, its upper block will be executed, and then the sequential block **BasicBlock$_n$** will be executed. The definition-reference chain can ensure that the error state at $\boldsymbol{s_1}$ is propagated to $\mathbf{s_2}$. According to the inclusion relationship between mutants and the definition of redundant mutants, $m_{x=(2,3,\ldots,6)}$ can be identified as the redundant mutant of m_1. Figure 6 shows an example of the $\mathbf{D_3}$ identification rule application.

The source program in Figure 6 is divided into blocks. It can be seen that lines 3, 4, 5, and 6 belong to the same loop block ($\mathbf{b_1}$), lines 4 and 5 belong to the same basic block ($\mathbf{b_2}$), and line 7 belongs to the same basic block ($\mathbf{b_3}$). $\mathbf{b_1}$ is the upper block of $\mathbf{b_2}$, and $\mathbf{b_3}$ is the sequential block of $\mathbf{b_1}$. The definitions of the variables x and y are in the 4th and 5th lines, expressed as $\mathrm{def}(x, y, 4, 5,))$, and the references to the variables x and y are in the 9th line, expressed as $\mathrm{ref}(x, y, 9)$. The mutants $m_{i=(1,2,3)}$ belong to the definition mutation set $M(\mathrm{def}, x, y, 4, 5)$ of the variables x and y. The change of variables x and y changes the value of the variable sum by $\mathrm{ref}(x, y, 7)$ in line 7. $m_{j=(4,5,6)}$ directly affect the change of the value of the variable sum by referencing the variables x and y. From the definition-reference chain $dr(x, y, 4, 5, 7)$, it can be seen that the 6 mutants have the same change state in the 7th row; that is, $m_{x=(2,3,\ldots,6)}$ are redundant mutants of m_1.

3.4. Identification Rule 4 (D_4). Suppose that there exists a definition-reference chain $dr(u, v, s_1, s_2)$, let mutants $m_{i=(1,2,3)}$ be the definition mutation set $M(\mathrm{def}, u, v, s_1)$ of the variables **u, v**, and let mutants $m_{j=(4,5,6)}$ be the reference mutation set $M(\mathrm{ref}, u, v, s_2)$ of the variables **u, v**. If $\mathbf{s_1}$ is a function call statement $i \longrightarrow^{*} j$ in module **i** with the variables **u, v** as arguments, $\mathbf{s_2}$ belongs to a **BasicBlock$_j$** in module **j**,

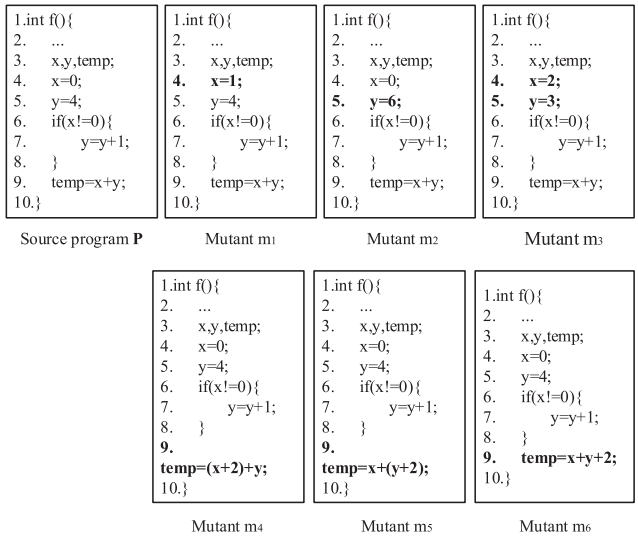

Figure 5: Example program for identifying rule D_2.

$S(\text{BasicBlock}_j) = \varnothing$ and meets Definition 1 in $\boldsymbol{s_1}$. It can be concluded that $m_{x=(2,3,\ldots,6)}$ are redundant mutants of m_1.

According to the identification rule $\mathbf{D_4}$, it is qualified that mutants $m_{i=(1,2,3)}$ and mutants $m_{j=(4,5,6)}$ belong to the cross-module relationship. $m_{i=(1,2,3)}$ and $m_{j=(4,5,6)}$ are the block, where the variable definitional appearance $((\text{def}, u, v, s_1))$ is located as $b_n \in i$, and the variable reference line appears in the $\text{BasicBlock}_n \in j$ where $((\text{ref}, u, v, s2))$ is located, and there is a call relationship $i \longrightarrow {}^* j$ between $\mathbf{b_n}$ and $\mathbf{BasicBlock_n}$, and $\mathbf{BasicBlock_n}$ has no subblock relationship in module **j**. From this, it can be deduced that, after the execution of $\mathbf{b_n}$, $\mathbf{BasicBlock_n}$ must be executed, and then the definition-reference chain can ensure that the error state at $\mathbf{s_1}$ is propagated to $\mathbf{s_2}$. Based on the inclusion relationship between mutants and the definition of redundant mutants, $m_{x=(2,3,\ldots,6)}$ can be identified as redundant mutants of m_1. Figure 7 shows an example of the $\mathbf{D_4}$ identification rule application.

The mul function is called by the f function in the third line, and the passing parameters are the variables x and y. The definition of variables x and y in the mutants $m_{i=(1,2,3)}$ is represented as $\text{def}(x, y, 3)$, and the references in the mutants $m_{j=(4,5,6)}$ are represented as $\text{ref}(x, y, 6)$. $m_{i=(1,2,3)}$ belong to the definition mutation set $M(\text{def}, x, y, 3)$ of variables x and y. The change of the variable will affect the return value of the function, and the value of the variable *mulp* in the program can be changed through the call of the function. $m_{j=(4,5,6)}$ belong to the reference mutation set $M(\text{ref}, x, y, 5)$ of the variables x and y, and the reference of variables x and y will directly affect the value of the variable *mulp*. From the definition-reference chain $dr(x, y, 3, 6)$, it can be seen that the 6 mutants have the same change state in the 6th row; that is, $m_{x=(2,3,\ldots,6)}$ are redundant mutants of m_1.

4. Experimental Analysis

4.1. Experimental Subjects. 8 C program sets (program source: http://sir.csc.ncsu.edu/portal/index.php, containing detailed information about experimental data) are used as

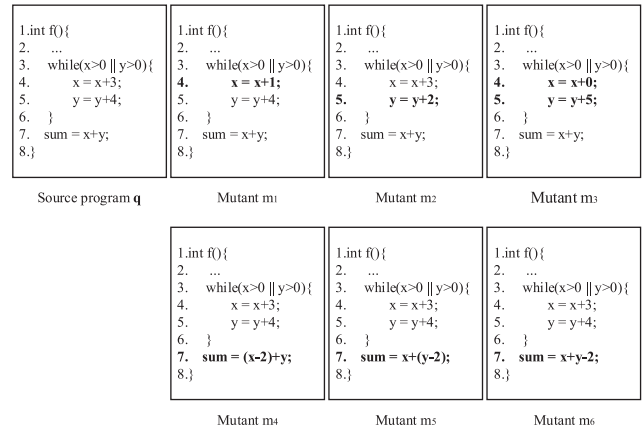

Figure 6: Example program for identifying rule D_3.

experimental objects to verify the feasibility and effectiveness of the algorithm, which can be divided into two categories: (1) Siemens program sets and the functional descriptions of the program set: print_tokens and print_tokens2 are lexical analyzers; schedule and schedule2 are schedulers; replace is pattern matching and replacement; tcas is a vehicle collision program; tot_info is data generation statistics. (2) The space assembly is the interpreter of the array definition language.

Firstly, a large number of mutants were generated for each source program using Proteum [25]; then, the data set was preprocessed to filter a set of equivalent mutants and a set of applicable mutants. The relevant information of the program set is shown in Table 1.

4.2. Experimental Steps and Results Analysis. Firstly, all sets of mutants with nonequivalent and single mutants are selected, and the data flow information of the experimental subjects is obtained using Frama-C [24]; then, the block rule file of the program to be tested is obtained using the static analysis method, and the mutation location information of the mutants is obtained; finally, the redundant mutants are identified after the identification rules $(\mathbf{D_1}\sim\mathbf{D_4})$.

The redundant mutants identified by the program for different experimental subjects were summed and counted, and the statistical results are shown in Table 2. The redundancy rate of their mutants was used to verify the effectiveness of the algorithm in reducing the execution time of the mutation testing, and the redundancy rate was calculated by the formula shown in equation (1):

$$R = \frac{\text{NRVI}}{\text{MN} - \text{MEN}} \times 100\%, \tag{3}$$

where R denotes the redundancy rate, NRVI denotes the number of identified redundant mutants, MN denotes the total number of mutants, and MEN denotes the number of equivalent mutants.

From Table 2 and Figure 8, it can be seen that there are different numbers of redundant mutants in the set of mutants of different experimental subjects. Although the redundancy rate accounted for by different experimental subjects varies greatly, this is enough to verify the existence

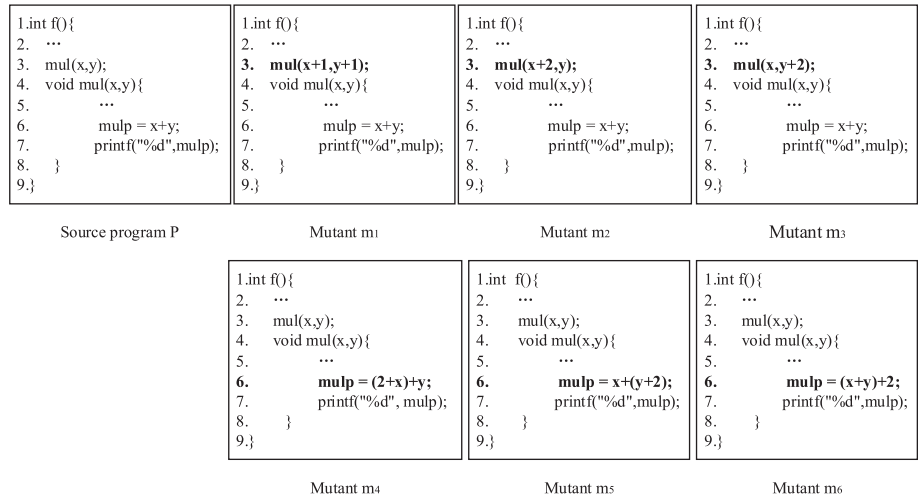

Figure 7: Example program for identifying rule D_4.

Table 1: Experimental procedure information.

Experimental subject	Number of lines of code (including commented lines of code)	Lines of code (excluding comment lines and blank lines)	Number of test cases	Total number of mutants	Number of equivalent mutants	Number of applicable mutations
print_tokens	565	343	4130	10881	583	3619
print_token2	510	355	4115	9369	545	3478
Schedule	412	296	2650	3516	204	1441
schedule2	307	263	2710	5794	471	2045
Replace	563	513	5542	21703	1556	7805
Tcas	173	137	1680	6478	442	2044
tot_info	406	281	1052	4308	678	3654
Space	9126	5982	136467	9380	1079	7986

Table 2: Redundant mutants of two variables for different experimental subjects.

Experimental subjects	Number of redundant mutants				Total redundant mutants
	D_1	D_2	D_3	D_4	
print_tokens	78	32	62	180	352
print_token2	33	0	76	112	221
Schedule	41	0	23	40	104
schedule2	18	7	27	53	105
Replace	175	0	267	131	573
Tcas	0	0	112	37	149
tot_info	86	16	213	186	501
Space	1046	132	786	0	1964

of redundant mutants. From the side, it also verifies the reason why a small number of test cases can kill a large number of mutants in mutation testing.

From Table 2 and Figure 9, it can be seen that the redundancy recognition rate varies widely among the different experimental objects, and the recognition variation of different identification rules is relatively large. Among the 8 C programs, it can be seen that the redundancy recognition rate of the space program reaches a maximum of 1964. However, the number of redundancy recognitions of the schedule is only 104. The validity analysis of the method is shown in Figure 9: This figure visualizes the proportion of redundant mutants under different identification rules. The number of redundant mutants identified in $\mathbf{D_1}$ and $\mathbf{D_3}$ is relatively high, accounting for 37% and 39% of the total number of mutants, respectively; the number of redundant mutants identified in $\mathbf{D_2}$ and $\mathbf{D_4}$ is relatively low, accounting for 5% and 19% of the total number of mutants, respectively. The following two points are summarized: (1) the larger the program size is, the more efficient the redundant mutants will be identified; (2)

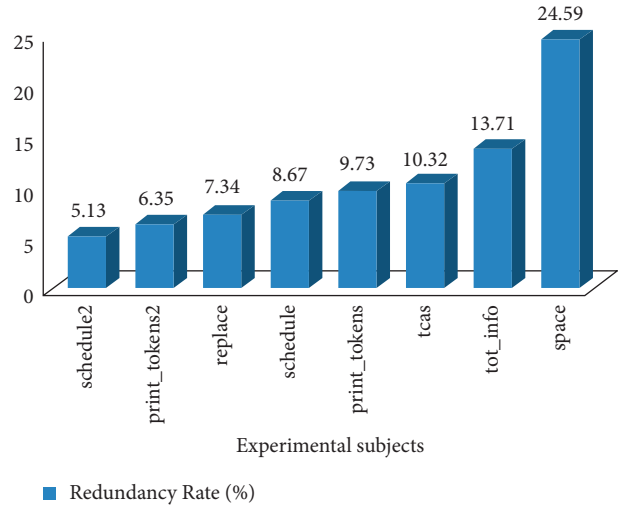

Figure 8: Redundant mutants of a variable in different subjects.

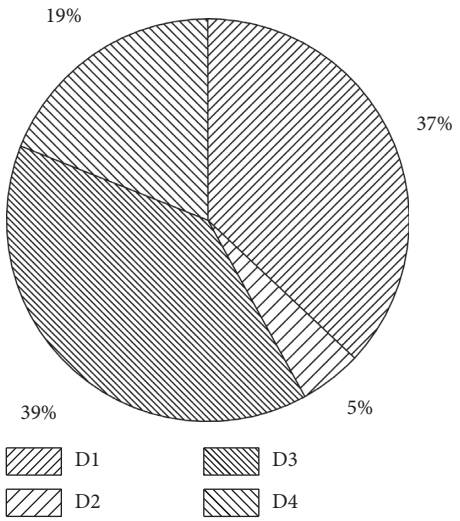

Figure 9: Percentage of redundant mutant identification rules.

from the identification rules, the program structure distribution of the program can be judged, and we can verify this from the source code of the program. If there are more sequential structures in a program, the rule $\mathbf{D_1}$ recognition will work in the recognition rule. There are more branch structures and loop structures in the Siemens program set. The variables defined in the branch structure and loop structure are used more in sequential blocks, which makes $\mathbf{D_3}$ identify more redundant mutants. The use of variables in the Siemens assembly is mostly through function calls, which makes $\mathbf{D_4}$ recognize more redundant mutants. Therefore, when there are more cyclic structures, the recognition rate of $\mathbf{D_3}$ will be increased. When there are more function calls, the recognition rule $\mathbf{D_4}$ will play an important role.

When the program size increases in the experimental program, the number of mutants increases accordingly when the test program passes through the mutation operator, and the number of test cases also increases, in which case the traditional mutation test execution time

Table 3: Redundant mutants of one mutation for different experimental subjects.

Experimental subjects	Number of redundant mutants				Total redundant mutants
	D_1	D_2	D_3	D_4	
print_tokens	59	10	37	123	249
print_tokens2	19	0	41	69	129
Schedule	31	0	12	20	63
schedule2	12	3	18	33	66
Replace	88	0	160	94	342
Tcas	0	0	73	23	96
tot_info	52	7	87	150	296
Space	872	73	521	0	1466

increases exponentially, significantly increasing the expensive resources for mutation testing. The redundant mutants are identified by statically scanning the definition-use chain of the program and the set of mutants, and it can be concluded that the identification of redundant mutants is linearly related to the size of the program and is not correlated with the test cases. Therefore, the expensive resources for mutation testing are reduced to some extent.

In order to better show the identification of redundant mutants based on two variables for data flow analysis, the identification of redundant variants based on one variable proposed by Sun et al. is given in Table 3 for Sun et al. [12].

Since the changed redundant mutant identification based on data flow analysis contains a definition-reference chain of variables as proposed by Sun et al., i.e., our proposed scheme has a higher identification rate in terms of implementation than the scheme proposed by Sun et al. As can be seen in Figure 10, the improved redundancy rate varies from one experimental subject to another, with the highest redundant mutant identification improvement of up to 498 in the space program; however, the schedule2 program has the lowest redundant mutant identification improvement of only 39 mutants.

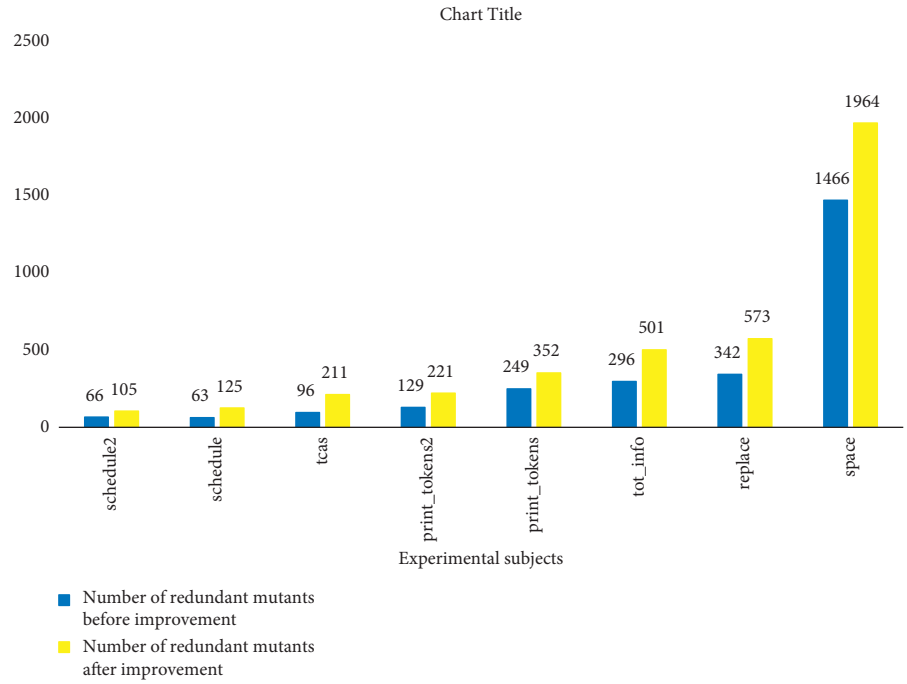

Figure 10: Comparison of redundant mutants before and after experimental improvement.

5. Conclusion

This paper improves the identification technique based on data flow redundant mutants from the point of view of reducing the execution time of mutant test, combines the program block structure and data flow analysis technique, and defines a set of redundant mutant identification rules based on the weak mutant test. A set of redundant mutant identification rules is defined based on weak mutation testing. The effectiveness of the proposed redundant mutant identification technique is evaluated using 8 C programs. The experiments show that a large number of redundant mutants can be identified using the method in this paper, which not only reduces the number of mutants, but also shortens the execution time of mutation testing; we have compared this with the previous technique and found that our improvement still greatly improves the identification of redundancy rate and further optimizes the identification of redundant mutants.

The main work in this paper is based on the improvement of the redundant mutant identification technique for data flow analysis. We introduce the inclusion relation of redundant mutants by adding one variable to achieve redundant mutant identification for two mutants, and the method can be better extended to multiple variables. Our next work is the redundant mutant identification technique for arbitrary multiple mutations, and better algorithms to further improve the testing efficiency.

Data Availability

The data are available at http://sir.csc.ncsu.edu/portal/index.php.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This research was supported by the Natural Science Foundation of Hebei Province, China, under Grant no. F2019201427 and Fund for Integration of Cloud Computing and Big Data, Innovation of Science and Education of China under Grant no. 2017A20004.

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, 1978.

[2] S. D. Wang, W. J. Yin, Y. K. Dong, L. Zhang, and H. Liu, "Data flow analysis for sequential storage structures," *Journal of Software*, vol. 31, no. 5, pp. 1276–1293, 2020.

[3] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2010.

[4] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proceedings of the 27th International Conference on Software Engineering*, pp. 402–411, Saint Louis, MO, USA, May 2005.

[5] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman, "Mutation testing advances: an analysis and survey," *Advances in Computers*, vol. 112, pp. 275–378, 2019.

[6] G. Rothermel, R. H. Untch, C. Y. Chengyun Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 10, pp. 929–948, 2001.

[7] R. A. Demillo, D. S. Guind, W. M. Mccracken, A. J. Offutt, and K. N. King, "An extended overview of the Mothra software

testing environment," in *Proceedings of the 2nd Workshop on Software Testing, Verification, and Analysis*, pp. 142–151, Banff, Canada, July 1988.

[8] R. A. DeMillo, "Test adequacy and program mutation," in *Proceedings of the 11th International Conference on Software Engineering*, pp. 355-356, Pittsburgh, PA, USA, May 1989.

[9] W. E. Wong, *On mutation and data flow*, Ph.D. Dissertation, Purdue University, West Lafayette, IN, USA, 1993.

[10] M. Sridharan and A. S. Namin, "Prioritizing mutation operators based on importance sampling," in *Proceedings of the IEEE 21st International Symposium on Software Reliability Engineering*, pp. 378–387, San Jose, CA, USA, November 2010.

[11] M. P. Usaola, P. R. Mateo, and B. P. Lamancha, "Reduction of test suites using mutation," in *Proceedings of the International Conference on Fundamental Approaches to Software Engineering*, March 2012.

[12] C. A. Sun, X. L. Guo, X. Y. Zhang, and T. Y. Chen, "A data flow based redundant mutants identification technique," *Chinese Journal of Computers*, vol. 42, no. 1, pp. 44–60, 2019.

[13] T. T. Chekam, M. Papadakis, M. Cordy, and Y. L. Traon, "Killing stubborn mutants with symbolic execution," *ACM Transactions on Software Engineering and Methodology*, vol. 30, no. 2, 2021.

[14] T. Titcheu Chekam, M. Papadakis, T. F. Bissyandé, Y. Le Traon, and K. Sen, "Selecting fault revealing mutants," *Empirical Software Engineering*, vol. 25, no. 1, pp. 434–487, 2020.

[15] N. Shomali and B. Arasteh, "Mutation reduction in software mutation testing using firefly optimization algorithm," *Data Technologies and Applications*, vol. 54, no. 4, pp. 461–480, 2020.

[16] S. M. J. Hosseini, B. Arasteh, A. Isazadeh, M. Mohsenzadeh, and M. Mirzarezaee, "An error-propagation aware method to reduce the software mutation cost using genetic algorithm," *Data Technologies and Applications*, vol. 55, no. 1, pp. 118–148, 2021.

[17] E. W. Krauser, A. P. Mathur, and V. J. Rego, "High performance software testing on SIMD machines," *IEEE Transactions on Software Engineering*, vol. 17, no. 5, pp. 403–423, 1991.

[18] K. N. King and A. J. Offutt, "A fortran language system for mutation-based software testing," *Software: Practice and Experience*, vol. 21, no. 7, pp. 685–718, 1991.

[19] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 2, pp. 99–118, 1996.

[20] K. Adamopoulos, M. Harman, and R. M. Hierons, "How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution," *Genetic and Evolutionary Computation - GECCO 2004*, vol. 3103, pp. 1338–1349, 2004.

[21] Q. Chen, K. Cheng, X. W. Zheng, H. S. Zhu, and L. M. Sun, "Function-level data dependence graph and its application in static vulnerability analysis," *Journal of Software*, vol. 31, no. 11, pp. 3421–3435, 2020.

[22] X. Yu, J. Liu, Z. J. Yang, X. Liu, X. Yin, and S. Yi, "Bayesian network based program dependence graph for fault localization," in *Proceedings of the 2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp. 181–188, Ottawa, OA, Canada, October 2016.

[23] C.-a. Sun, F. Xue, H. Liu, and X. Zhang, "A path-aware approach to mutant reduction in mutation testing," *Information and Software Technology*, vol. 81, pp. 65–81, 2017.

[24] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-C: a software analysis perspective," *Formal Aspects of Computing*, vol. 27, no. 3, pp. 573–609, 2015.

[25] M. E. Delamaro, J. C. Maldonado, and A. M. R. Vincenzi, "Proteum/IM 2.0: an integrated mutation testing environment," in *Mutation Testing for the New Century*, W. E. Wong, Ed., Springer, Boston, MA, USA, pp. 91–101, 2001.