ITC 1/52
Information Technology and Control
Vol. 52 / No. 1 / 2023
pp. 68-84
DOI 10.5755/j01.itc.52.1.30715

Automatic Repair of Java Programs with Mixed Granularity and Variable Mapping

Received 2022/02/14 | Accepted after revision 2022/11/07

https://doi.org/10.5755/j01.itc.52.1.30715

HOW TO CITE: Cao, H., Cui, Z., Deng, M., Chu, Y., Meng, Y. (2023). Automatic Repair of Java Programs with Mixed Granularity and Variable Mapping. *Information Technology and Control*, 52(1), 68-84. https://doi.org/10.5755/j01.itc.52.1.30715

Automatic Repair of Java Programs with Mixed Granularity and Variable Mapping

Heling Cao

Key Laboratory of Grain Information Processing and Control, Henan; Ministry of Education; Henan University of Technology; Zhengzhou 450001; China; e-mail: caohl@haut.edu.cn

Zhiying Cui, Miaolei Deng, Yonghe Chu, Yangxia Meng

College of Information Science and Engineering; Henan University of Technology; Zhengzhou 450001; China; e-mails: cuizhiying2021@163.com, dengmiaolei@haut.edu.cn, chuyonghe@haut.edu.cn, 1958502687@qq.com

Corresponding author: dengmiaolei@haut.edu.cn

During the process of software repair, since the granularity of repair is too coarse and the way of fixing ingredient is too simple, the repair efficiency needs to be further improved. To resolve the problems, we propose a Mixed Granularity and Variable Mapping based automatic software Repair (MGVMRepair). We adopt random search algorithm as the framework of program evolution, and utilize the mapping relationship between variables as an auxiliary specification. Firstly, fault localization is used to locate the suspicious statements and to form a list of modification points. Secondly, the ingredient of program repair at statement level is obtained, and the mapping relationship of variables is established. Then, the test case prioritization is improved from the perspective of the modification point. Finally, a program passes all test cases or the program iteration terminates. The experimental results show that MGVMRepair has a higher repair success rate than GenProg, CapGen, SimFix, jKali, jMutRepair and SketchFix on Defects4J.

KEYWORDS: Automatic software repair, random search, mixed repair granularity, test case prioritization.

1. Introduction

The application of the software has been integrated into all aspects of life, involving many fields such as national defense, aviation, economy, and medical care. The gradual generation of various complex requirements has also increased the complexity of the software, and faults are inevitable in software programs. A variety of software faults are generated almost every day, and the types of faults have become more complicated. In 2006, Mozilla's software maintenance staff observed that approximately 300 program bugs were found every day, a number far larger than Mozilla's capacity to handle [24]. The automatic repair is a promising approach to reduce the costs of manual debugging and increase software quality [18].

Although the researchers have proposed a variety of approaches and technologies to support the automatic repair of software faults, the existing research results show that the current automatic repair technology of software faults is still in its infancy, and the efficiency need to be improved. Fault repair is generally is generally time-consuming and expensive process for developers.

Existing fault repair techniques work at the statement level, which are too coarse in repair granularity. Le Goues et al. [14] implemented the fault repair prototype tool GenProg, which was the first fault repair tool based on genetic algorithm. Subsequently, Le Goues et al. [13] carried out a series of improvements and empirical studies on GenProg. Qi et al. [20] followed the mutation rules of GenProg and replaced the genetic algorithm in GenProg with random search algorithm to implement the prototype tool RSRepair for fault repair. Kim et al. [12] summarized different repair strategies from manually written patches to implement the fault repair prototype tool PAR. The above repair techniques [12-14, 20] work at the statement level, and the repair granularity is too coarse, which fails to use repair materials in a proper way. The repair material cannot be reasonably utilized, more fault programs cannot be repaired, therefore, these approaches have the lower efficiency of the repair.

Currently, there are fault repair techniques based on fine-grained levels such as SimFix [9], CapGen [23] and SketchFix [7]. Jiang et al. [9] proposed SimFix, which is an automatic repair method based on fine-grained code differences, while using similarity as a constraint for donor fragment selection and program evolution at the abstract syntax tree nodes. Wen et al. [23] proposed CapGen, which used the contextual information of the abstract syntax tree nodes for buggy program repair, and considered the abstract syntax tree node types and suspicious code elements of the desired components when selecting mutation operators. Hua et al. [7] proposed SketchFix, which worked in repairing bugs with expression manipulation at the AST node-level granularity, and utilized runtime information to substantially prune the space of candidate. In the above fine-grained program repair approaches, the search space of the candidate patches is so large that the computational cost is high, resulting in a lower repair efficiency of program repair.

To address the problems of too coarse in repair granularity, inaccurate material usage and low repair success rate, we propose a fault program evolution repair method MGVMRepair based on hybrid granularity and variable mapping. We mainly adopt a random search algorithm in program evolution, and use the mapping relationship between variables as an auxiliary specification. To summarize, this paper makes the following contributions:

- We propose a mixed granularity defect repair method, MGVMRepair to improve the success rate of buggy program repair.
- We propose a variable mapping approach that enables better use of defect repair ingredients at a fine-grained level.
- We propose a test case prioritization technique based on modification point execution information that can further improve the efficiency of program verification.
- We conduct the experimental study 224 real world faults on Defects4J defect dataset to show that MGVMRepair has higher repair success rate than the existing defect repair approaches.

The rest of this paper is organized as follows. Section2 provides an overview of related works. Section 3 outlines the background of automatic software repair. Section 4 elaborates the framework of our approach.

After that, we introduce the experimental study on the Defects4j in Section 5. Finally, we conclude the paper with potential future directions in Section 6.

2. Related Work

2.1. Coarse-grained Based Program Fixes

Coarse-grained based program fixes are at statement level. Le Goues et al. [14] implemented the fault repair prototype tool GenProg, which was the first fault repair tool based on genetic algorithm. Subsequently, a series of improvements and empirical studies on GenProg were conducted in Le Goues et al. [6, 13, 22]. Qi et al. [20] followed the mutation rules of GenProg and used a random search algorithm instead of the genetic algorithm in GenProg to implement the fault repair prototype tool RSRepair. The experimental results showed that GenProg did not take full advantage of the genetic algorithm. Kim et al. [12] summarized different repair strategies from manually written patches to implement the fault repair prototype tool PAR. Martinez et al. [16] present Astor, a publicly available program repair library that includes the implementation of three notable repair approaches (jGenProg2, jKali and jMutRepair) to explore the design space of automatic repair for Java. Le et al. [15] indicated that many real-world bugs cannot be repaired by existing techniques even after more than 12 hours of computation in a multi-core cloud environment. Afzal et al. [3] proposed SOSRepair, an automated program repair technique that used semantic code search to replace candidate buggy code regions with behaviorally-similar code written by humans. Chen et al. [5] proposed SequenceR, a sequence-to-sequence deep learning model that aimed at automatically fixing bugs by generating one-line patches.

The granularity of these approaches is too coarse to apply repair materials in a more appropriate way and to fix more buggy programs. Our approach has the higher efficiency than the approaches mentioned above, due to making use of a mixed granularity and variable mapping repair.

2.2. Fine-grained Based Program Fixes

Fine-grained based program fixes are at expression level. Jiang et al. [9] proposed SimFix to extract an abstract search space from existing patches and similar code from buggy source programs to form a concrete candidate patch search space, and obtain program patches from the intersection of these two search spaces. Subsequently, Jiang et al. [8] proposed another program transformation approach, GENPAT, which inferred program transformations based on code context and statistical information from a large code corpus. Wen et al. [23] proposed CapGen at a fine-grained level (e.g., expressions) to extract the contextual environment information of abstract syntax tree nodes and to select repair operators and ingredients under the contextual environment. Hua et al. [7] proposed an on-demand repair technique, SketchFix, which tightly integrated generation and validation phases of candidate programs. The technique reduced program repair to program synthesis by transforming faulty programs to sketches at the AST node-level granularity. Yuan and Banzhaf [25] proposed ARJA, a lower-granularity patch representation which properly decoupled the search subspaces of likely-buggy locations, operation types and ingredient statements. Thus, genetic programming can traverse the search space more effectively.

If the granularity of fault repair approaches is too fine, it is easy to cause explosion of search space of candidate patch. Our approach further improves the efficiency of the verification and reduces the sorting times of test cases. Therefore, our approach owns the higher efficiency than the compared SimFix, CapGen, and SketchFix approach.

3. Background

3.1. Repair Granularity

The repair granularity of fault repair approaches is a critical issue in the field of automatic fault repair. If the granularity is too coarse (e.g., at the statement level), the repair material cannot be reasonably used, if the granularity is too fine (e.g., at the expression level), search space of the candidate patch will explode easily. Existing fault repair techniques[6, 12-14, 20-21] work at the statement level, and the repair granularity of these methods is too coarse to apply the repair material in a more appropriate way to repair more defective programs.

As shown in Figure 1, an expression fault in the return statement at line 417 caused a fault in the Math

Figure 1
The faulty statement, correct statement and repair ingredient of Math 63 project on Defects4J

```
//Faulty statement
417: - return (Double.isNaN(x) && Double.isNaN(y)) || x == y;
//Correct statement
417: + return equals(x,y,1);
//Repair Ingredient
422: return (Double.isNaN(x) && Double.isNaN(y)) || equals(x,y,1);
442: return equals(x,y,1) || FastMath.abs(y - x) <= eps;
```

63 project on Defects4J. To fix the fault, the software maintainer changed the expression at the statement to equals(x,y,1). In this buggy program, the correct repair material is present in lines 422 and 442 of the program. If the repair is performed at statement-level granularity and the incorrect statement is directly replaced using another statement, the program fault cannot be eliminated by replacing the statement at lines 422 and 442. This motivates this paper to consider a more fine-grained repair approach to fix the buggy programs.

Martinez et al. [17] argue that code redundancy is more obvious at a finer granularity than the statement level, which indicates that better repair material can be found at finer granularity levels and that buggy programs are more likely to be repairable. Based on the current repair technology and research status, this paper proposes a fault program evolution repair method based on mixed granularity and variable mapping to solve the problems of too rough granularity at the statement level, too simple material usage and a low repair success rate.

3.2. Random Search Algorithm

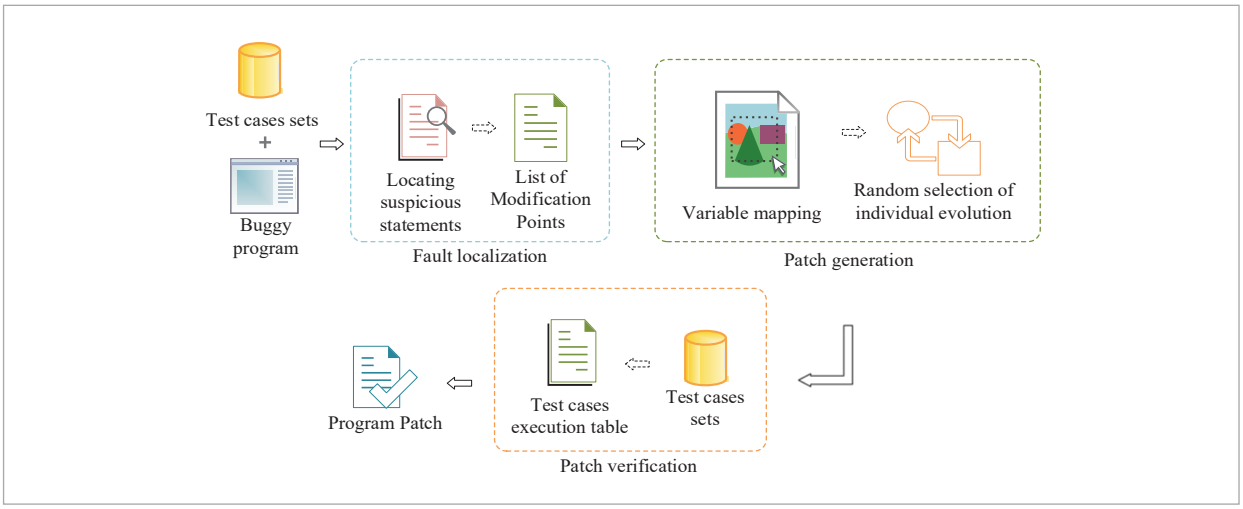
Compared with genetic programming, the random search algorithm is the simplest way of selection. When selecting individuals of the population for evolution, the genetic programming resorts to the fitness function as a constraint for individual selection, but the random search algorithm randomly selects individuals for the evolution of the buggy program.

In theory, genetic programming using fitness function evaluation can better guide the population evolution process. But, one of the challenges in the field of automatic repair of faults by intelligent evolutionary algorithms is how to set the appropriate fitness function to ensure the efficiency and accuracy of individual evaluation. If the fitness function is set too simply, it will not be able to distinguish the variability among individuals, which will reduce the repair effect or even lead to the failure of the repair. If the fitness function is set too complex, it will increase the time complexity of the repair algorithm and expand the time overhead of program evolution. In a study by Qi et al. [20], it was found that the existing fault repair method GenProg did not give full play the advantage of genetic programming, however a good repair effect could also be achieved by using a random search algorithm.

4. Our Approach

Different from the current coarse-grained fault program repair and fine-grained fault program repair, we propose a combination of coarse-grained and fine-grained fault program repair method MGVM-Repair. At the same time, in order to further improve the efficiency of patch verification, a test case prioritization technology based on the execution information of the modification point is proposed. This fault program repair method first uses fault localization technology to locate suspicious sentences and form a list of modified points. Secondly, the material space of the current sentence level is obtained. When selecting materials, the mapping relationship between the variables in the material sentence and the modified point sentence is established, and the variable or expression required for repair is selected according to the variable mapping relationship to replace the modified point sentence variable or expression statement. The entire program repair process is based on the idea of random search algorithm. When selecting population individuals for evolution, the selection is no longer based on the fitness value of individuals in genetic programming, but individuals are randomly selected for evolution. Then according to the test case execution status of the modified point, a test case priority execution information table is maintained for each modification point. The priority execution information table is dynamically adjusted according to the test case execution process. This program verification method can further improve the verification efficiency of candidate patches. The framework of our approach is shown in Figure 2.

Figure 2
The framework of our approach



1 Fault localization. In this stage, the fault source procedure is localized and analyzed by using the fault localization technique Ochiai, whose suspicious value is calculated as described in Equation (1).

$$Ochiai(s)=\frac{n_{ef}(s)}{\sqrt{n_f\times(n_{ef}(s)+n_{ep}(s))}}. \quad (1)$$

Among them, s represents the program entity, $n_{ep}(s)$ and $n_{ef}(s)$ denote the number of successful test cases and the number of failed test cases covering program entity s, respectively, and n_f denotes the number of all failed test cases of program entity s

2 Patch generation. The program undergoes evolution using a mixed-granularity evolution technique, where the search space of the material is formed at the statement level and the expressions and variables in the ingredients are applied at the expression level. For specific fixes, ingredient statements are obtained from the ingredient space and modification points are selected from the list of modification points. The types and names of expressions and variables in ingredient statements and modification point statements are extracted. Then the mapping relationship between modification points and variables in the ingredient is established, and is used as the basis for expression replacement. The constructed variable mapping relationship is a constraint condition for the application of material, and is also utilized to guide the evolution process. On this foundation, the fault procedure is evolved according to the idea of random search algorithm. This is, individuals are not selected according to the fitness values in genetic programming, but are randomly selected when selecting individuals of the population for evolution.

3 Patch verification. After the candidate patches are generated by program evolution, test cases need to be run repeatedly to verify the validity of the candidate patches. During this phase, a test case execution information table is maintained for each modification point based on the test case execution at the modification point, and the table is dynamically adjusted according to the test case execution process. This verification strategy can reduce the time complexity of the repair algorithm and decrease the program verification time without adjusting the test case execution order for each individual verification.

4.1. Fault Localization

Before fixing a buggy program, it is necessary to find out the location of likely-debugs in the program. Different fault localization techniques have different effects on the process of candidate patch generation. If the real bug is located in a program statement with a higher suspicious value, the correct patch can be found in a shorter time with fewer repair attempts.

If the real bug is located in a program statement with a lower suspicious value, it may take more computational resources to repair the buggy program or even cause the program to fail to repair. Therefore, the accuracy of fault localization technique plays a crucial role in the repair of buggy programs. In an empirical study by Abreu et al. [1, 2], it found that the fault localization effectiveness of Ochiai [2] is better than that of Tarantula [10] and Jaccard [4].

The modification points are selected according to the suspiciousness of the statements after fault localization, and a list of modification points is formed. In the actual repair process, the list of modification points is the suspicious space in the evolutionary repair of the buggy program, and the weight of modification points is calculated as shown in Equation (2).

$$W(i)=\frac{S_i}{\sum_{j=1}^{n} S_j}, \tag{2}$$

where i denotes the location information of the suspicious statement, S_i and S_j denote the suspicious value of the suspicious statement, and $W(i)$ denotes the weight value of the suspicious statement. When selecting the modification point, the weight value of the suspicious statement is used as a constraint for selecting the modification point.

4.2. Patch Generation

4.2.1 Mixed Repair Granularity

Mixed repair granularity refers to forming the ingredient search space at the statement level and applying the expressions and variables in the ingredients at the expression level. The repair at the statement level is a coarse-grained program repair, and the expression level is a fine-grained program repair. Both coarse-grained repair and fine-grained repair have extreme advantages and disadvantages, so we use a combination of coarse-grained repair and fine-grained repair for the buggy program evolution as a whole. In the process of program evolution, the ingredient search space is first formed at the coarse-grained level, i.e., the statement level, and in the ingredient search space at the statement level, the ingredient statements are first filtered according to the similarity values. After screening out the eligible ingredient statements, the mapping relationship is established between the expressions and variables between the selected ingredient statements and the modified point statements, and is used as a constraint for expression and variable modification in the program evolution.

When selecting an ingredient statement, the return value type and method name of the method, and the type and variable name of the variables contained in the statement are extracted. Then the similarity between the ingredient statement and the modification point is calculated using the Dice similarity coefficient, and the ingredient statement is selected based on the similarity. The similarity is calculated as shown in Equation (3).

$$Sim(f_i,ml)=\alpha\times Sim_1+\beta\times Sim_2, \tag{3}$$

where f_i represents the ith current repair material, ml represents the current modification point, $Sim(f_i,ml)$ represents the similarity between f_i and ml, Sim_1 represents the method similarity between the two, Sim_2 represents the variable similarity between the two, and α and β represent the method similarity coefficient and variable similarity coefficient, respectively.

4.2.2. Mixed Repair Granularity

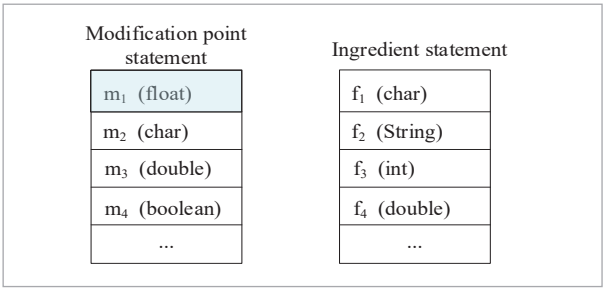
After selecting an ingredient statement based on similarity, a variable mapping relationship between the ingredient statement and the modification point statement needs to be established. And this mapping relationship is used as a constraint for expression or variable replacement. When the buggy program is modified, the expressions or variables in the ingredient statements need to be reused. Therefore, it is important to ensure that the mapped expressions or variables have type compatibility. An expression in a modification point statement can be modified only when the ingredient statement is of the same type as the expression contained in the modification point statement. When the modification point statement contains more than one expression, the first consideration is whether the types of the expressions are consistent; if the types are consistent, the expressions in the ingredient statement can be selected for modification operation; otherwise, the modification is impossible.

When the variable is used as a left value in a modification point statement, you need to make sure that the type of the variable in the selected ingredient statement is the parent type of the original variable; when

the variable is used as a right value in a modification point statement, you need to make sure that the type of the variable in the selected ingredient statement is a subtype of the original variable. There is no explicit subtype relationship for basic types in Java program. Therefore there are two basic types T and T', if any value in type T can be converted to a value in type T' without loss, T is considered to be a subtype of T' and there is type compatibility between T and T'. The variables between both the selected modification point statement and the ingredient statement are combined one by one when performing variable mapping, and if there is type compatibility between the two variables, the set of mapped variables is saved. This mapping relationship is used as the main constraint for variable selection. When the variables are subsequently selected to modify the variables in the modification point, it is possible to further determine whether the location can be changed based on its location.

Figure 3 shows an example of variable mapping, in which m_1~m_2 and f_1~f_4 represent the variables in the modification point statement and material statement, respectively, and the content in parentheses indicates the type of variables. Taking the variable m_1 in the modification point statement as an example, if the variables between the modification point statement and the material statement are combined one by one, there are four combinations of $<m_1, f_1>$, $<m_1, f_2>$, $<m_1, f_3>$, and $<m_1, f_4>$. Among these four combinations of variables, there is a type compatibility between the variable combinations $<m_1, f_3>$ and $<m_1, f_4>$, and then the two sets of variable mapping relationships are preserved. When m_1 is used as the left value in the modification point statement, we further judge these two sets of mapping relationships and find that the variable combination $<m_1, f_3>$ has a loss of precision

Figure 3
Variable mapping illustration



when replacing variable m_1 (float type) with variable f_3 (int type) in the modification point statement, but the variable combination $<m_1, f_4>$ does not have this situation, so the variable combination relationship in $<m_1, f_4>$ is as a constraint when the variables are modified. When m_1 is used as the right value in the material statement, it is found that the variable combination $<m_1, f_4>$ can expand the range of the right value expression type when using the variable f_4 (double type) to replace the variable m_1 (float type) in the modification point statement, which may conflict with the left value variable type in the modification point statement and easily introduce a new program bug, so the selection of this group of variable combination is dropped. The variable combination $<m_1, f_3>$ is selected as the constraint condition for variable modification.

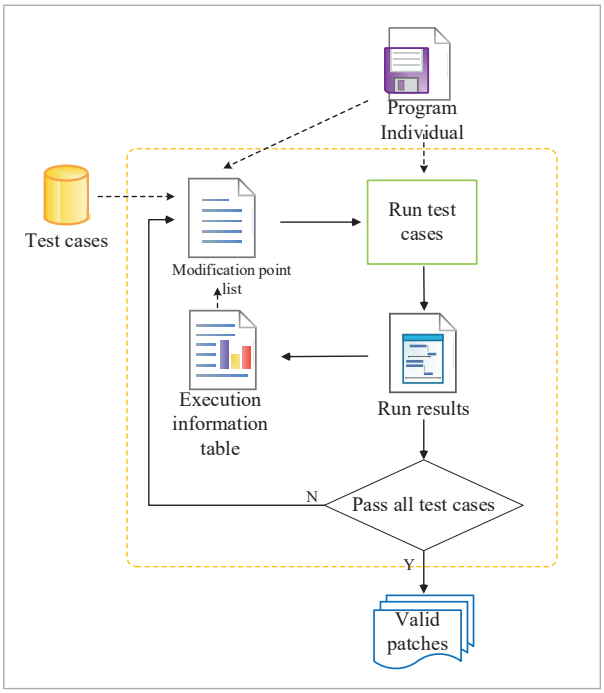
4.3. Patch Verification

The test case prioritization technique ranks test cases, according to their ability to identify the faults during the stage of the execution. And each individual verification requires a ranking of the test case set, which results in a large time overhead for program execution. Therefore, to further improve the efficiency of individual patch verification, we prioritize the test cases according to the test case execution information at the modification point.

Regression testing is often used during the process of verifying defect patches. Regression testing aims to make sure that the change did not introduce new faults or cause faults in other code when you modify old code and re-test it, so it plays an important role in software defect repair. The purpose of patch verification is to check whether the generated candidate patch can fix the faults in the original defective program and whether the repaired defective program introduces new faults. Finally, if a patch passes all test cases, the patch is considered valid, the repair process ends, and a valid patch is output. Otherwise, if any test case fails, the patch is invalid.

As shown in Figure 4, the test case prioritization technique based on modification point, each modification point has its corresponding test case priority execution information table, and the test cases are executed in different order among different modification points. A test case execution information table is maintained for each modification point, and dynamically adjusted according to the test case execution

Figure 4
Test case prioritization technique based on modification point



process. During the next-generation evolution, for the individual who selects the modification point for program evolution, the test cases in the information table are executed first, and then the other test cases in the test case set are executed after finishing the execution of the test cases in the information table. The test cases that have been executed at the modification point are marked, and are verified when all the program individuals evolved for the modification point. If any test case is not added to the test case execution priority information table, the test case is added to the test case execution information table and the table is dynamically adjusted. When an individual passes all test cases, the individual is output as a valid patch.

While the test case prioritization technique based on the ability to identify errors requires test case ordering once per individual verification, the test case prioritization technique based on modification point execution information only requires uniform adjustment of the modification point execution information table at the end of each generation of the verification process. Compared with the former, the latter only needs to adjust the information execution table once at the end of each generation of individual program verification. Since each generation of program evolution contains multiple program individuals, the test case prioritization technique based on the execution information of modification points can greatly reduce the number of test case prioritization adjustments and thus improve the program verification efficiency.

4.4. Bug Fixing Algorithm MGVMRepair

Algorithm 1 describes the specific process of the buggy program repair algorithm MGVMRepair in this paper. There are a buggy source program P and its corresponding test case set T. In lines 1~2 of this algorithm, a sequence of suspicious statements is located using the fault localization tool, and the suspicious space ModList is generated by selecting the suspicious statements according to the minimum suspicious value minSus and the maximum number of modification points maxMod by the program settings. Line 3 indicates the initialization operation of the population, and line 4 is the marker of whether the buggy program P is successfully repaired. Lines 7 to 22 describe the specific process of individual variation evolution. First, the parent variant parent is selected according to random selection (line 7), the modification point modPoint is selected from the list of modification points according to the size of the suspected value weight (line 8), and the operator Op is randomly selected from the operator space.

If the current operator can be applied to this modification point location at line 10 and the operator requires repair material at line 11. The repair material with high similarity to the modification point is obtained from the current material space as the ingredient statement at line 12. Line 13 indicates that the variables in the modification point statement and the material statement are extracted. Lines 14 to 16 indicate that a mapping relationship is established between the variables in the modification point statement and the variables in the material statement, and if a group of variables are type compatible (line 15), the mapping relationship will be stored in the matchMap as a constraint condition for variation evolution in the subsequent program (line 16). From the saved variable mapping relations, the repair material for the mutation evolution of the program individual is selected (line 18), and the mutation evolution of the current parent individual is performed according to the currently selected modification point, operator

and repair material (line 19). If the selected modification point does not require repair material, the selected parent individual is modified directly (line 21). Finally, the evolved generated program individual is saved to Offsprings (line 22).

Algorithm 1: MGVMRepair repair algorithm

```
Input P          // Faulty program
      T          // Test case sets
      OperatorSpace   // Operator space
      popSize         // Population size
      IngredientSpace // Ingredient Space
Output: cp       // Valid patches through all test cases
begin
1   SusList ← FaultLocalization(T,P)
2   ModList ← GetSusSpace(SusList,minSus,maxMod)
3   Pop ← InitPopulation(popSize)
4   fixSuccess = false
5   repeat
6   for i ← 1 to maxMut do
7     parent ← URSelect(Pop)
8     modPoint ← WRSelect(ModList)/
9     Op ← URSelect(OperatorSpace)
10    if canApplyOp(Op,modPoint) then
11      if opNeedIngredients(Op) then
12        fixIngredient ← getSimIng(IngredientSpace)
13          mvariables, fvariables ← get code variables of
              modPoint, fixIngredient
14            for all (mvariable, fvariable) ∈ (mvariables,
              fvariables) do
15          if mvariable and fvariable are compatible then
16            matchMap.add(mvariable, fvariable)
17        end for
18        fixingIng ←WRSelect(matchMap)
19      child ← generateNewVariant(parent,modPoint,Op,
          fixingIng)
20      else
21      child ← generateNewVariant(parent,modPoint,Op)
22    Offsprings add child
23    getmodPointTests(child)
24    if modPointValid(child) = true then
25      if validRestTests(T) = true then
26        cp ← child
27        fixSuccess = true
28        break
29      else markDifTests(T)
30  end for
31  adjustModPointTests(child)
32  Pop ← URSelect(PopSize,Offsprings,parents)
33  until fixSuccess = true
34  return cp
end
```

Lines 23 to 29 represent the specific process of program individual verification. First, the modification point selected by the current program individual is obtained (line 23); then the test cases of the modification point to execute the test cases in the information table is run, and if all the test cases in the information table can pass the execution (line 24), then the other test cases in the test case set are run. If all the remaining test cases in the test case set can pass the execution, the output of this program individual as a valid patch for this buggy program is saved (line 26), and the fix marker is set to true (line 27) to end the fixing process of this buggy program; otherwise, the test cases that have been executed at this modification point are marked (line 29). After this generation of population verification, for each modification point, if any test case is not added to the test case priority execution information table, this test case is added to this information table and the table is dynamically adjusted (line 31). After completing the variant evolution and verification process of the buggy program, PopSize individuals are randomly selected from the Offersprings as the parent variants for the next generation of program evolution (line 32) until a program individual passes all test cases or reaches the program iteration termination condition.

5. Experimental Study

We perform the empirical evaluation over a real bugs database, called Defects4J [11], which has been extensively used for evaluating Java repair systems. This experiment was conducted with Spoon [19], a code parsing tool, GZoltar [21], a fault location tool, and the operation system of Ubuntu 18.04 LTS with a 2.40 GHz Intel(R) CPU and 8G memory.

Referring to existing repair approaches [9, 20, 23], this paper verifies the effectiveness of the repair in three aspects: (i) the number of successful repairs of buggy programs, (ii) the generated NCP (Number of Candidate Patches) values when the buggy programs are successfully repaired, and (iii) the time spent when the buggy programs are successfully repaired.

5.1. Comparison Between MGVMRepair and GenProg

In order to demonstrate the repair effectiveness, the classical repair tool GenProg was selected. By ana-

lyzing the source code of the two methods, the time complexity of MGVMRepair is O(n^3), and the time complexity of GenProg is O(n^4). Table 1 shows the detailed comparison between MGVMRepair and GenProg for repairing the four major items of Chart, Math, Lang, and Time. The experimental results on the Defects4J dataset show that GenProg can successfully repair 28 buggy programs with a success rate of 12.50%, while MGVMRepair can successfully repair 40 buggy programs with a success rate of 17.86%. Compared with GenProg, the overall repair success rate of MGVMRepair is improved by 42.9%. In order to reduce the influence of chance factors, the data collected for each defect program is the average of 10 runs. Figure 5 shows the comparison of the number of fixable faults between MGVMRepair and GenProg on the four major buggy items (Chart, Math, Lang, and Time). And it can be seen that MGVMRepair outperforms GenProg in fixing all four major items. Figure 6 is a Venn diagram of the comparison of the fixes between MGVMRepair and GenProg. It can be visually seen that the specific differences between MGVMRepair and GenProg repair situations from this diagram.

From Table 1 and Figures 5 and 6, it can be seen that in terms of the number of buggy programs fixed, MGVMRepair fixes 1 more buggy program than GenProg on the Chart project; MGVMRepair fixes 6 more buggy programs than GenProg on the Math project. On the Lang project, GenProg has no buggy programs to repair successfully, while MGVMRepair

Table 1
Comparison of MGVMRepair and GenProg Repair Situation

Project	Number of Faults	Fixable Bug Procedures	
		MGVMRepair	*GenProg*
Chart	26	C1,C3,C6,C11, C12,C15,C24,C25	C1,C3,C5,C7, C13,C15,C25
		å=8	å=7
Math	106	M2,M5,M8,M20,M28, M30,M40,M44,M49, M50,M53,M57,M60, M63,M70,M71,M73, M78,M80,M81,M82, M84,M85,M95,M97	M2,M5,M8,M28,M40, M49,M50,M53,M60, M70,M71,M73,M78, M80,M81,M82,M84, M85,M95
		å=25	å=19
Lang	65	L7,L22,L24,L27	-
		å=4	å=0
Time	27	T4,T9,T11	T4,T11
		å=3	å=2
Total	224	40	28
Success rate	-	17.86%	12.50%

Figure 5
Comparison of repair quantity between MGVMRepair and GenProg

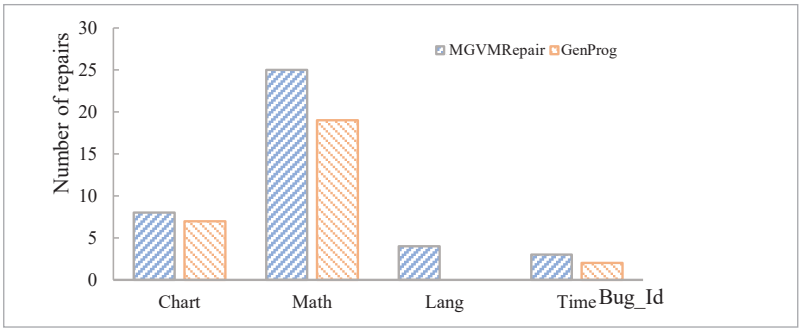
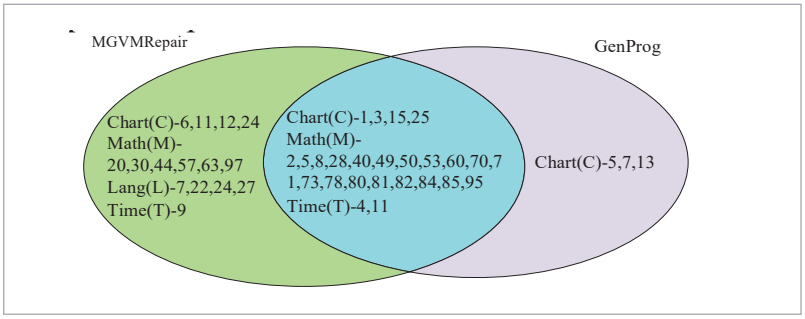


Figure 6
Venn diagram of MGVMRepair and GenProg repair comparison



can repair 4 buggy programs successfully. On the Time project, MGVMRepair repaired 1 more buggy program than GenProg. MGVMRepair's repair success rate is improved due to making use of a mixed granularity and variable mapping repair, which is able to find the appropriate repair ingredients and apply them correctly to the evolution of the buggy program. The mixed granularity repair can quickly find the material statements that meet the requirements of the protocol, and the variable mapping allows for the variant evolution of the buggy program at a fine-grained level. Therefore, MGVMRepair can overcome the drawbacks of the coarse-grained repair approach by using materials in an overly simplistic manner and increase the successful repair rate of the buggy program.

Table 2 provides detailed information on the NCP values, verification time, and total repair time for the

Table 2

Comparison of Repair Information between MGVMRepair and GenProg

Project	Bug_Id	Repair approaches	NCP	Verification time (min)	Total time (min)	Validation time improvement rate	Total time improvement rate
Chart	C1	GenProg	20	15.6	23.2	76.9%	64.2%
		MGVMRepair	6	3.6	8.3		
	C3	GenProg	58	17.3	32.3	41.6%	32.8%
		MGVMRepair	38	10.1	21.7		
	C15	GenProg	46	11.3	25.3	62.0%	46.6%
		MGVMRepair	26	4.3	13.5		
	C25	GenProg	35	18.5	36.7	84.3%	81.7%
		MGVMRepair	25	2.9	6.7		
Math	M2	GenProg	42	45.6	63.4	89.3%	78.2%
		MGVMRepair	9	4.9	13.8		
	M5	GenProg	63	58.2	68.3	90.5%	67.6%
		MGVMRepair	15	5.5	22.1		
	M8	GenProg	25	44.8	80.6	88.8%	72.8%
		MGVMRepair	11	5.0	21.9		
	M28	GenProg	53	66.2	89.1	90.6%	80.7%
		MGVMRepair	17	6.2	17.2		
	M40	GenProg	54	78.3	89.4	96.8%	81.3%
		MGVMRepair	6	2.5	16.7		
	M49	GenProg	43	90.2	100.4	91.7%	70.9%
		MGVMRepair	25	7.5	29.2		
	M50	GenProg	20	21.5	43.6	83.3%	78.2%
		MGVMRepair	9	3.6	9.5		
	M53	GenProg	64	77.2	117.3	92.1%	81.0%
		MGVMRepair	31	6.1	22.3		
	M60	GenProg	42	37.4	50.9	80.5%	75.2%
		MGVMRepair	38	7.3	12.6		
	M70	GenProg	38	18.3	23.8	80.3%	42.4%
		MGVMRepair	22	3.6	13.7		
	M71	GenProg	64	28.9	42.7	66.4%	45.9%
		MGVMRepair	14	9.7	23.1		

Project	Bug_Id	Repair approaches	NCP	Verification time (min)	Total time (min)	Validation time improvement rate	Total time improvement rate
Math	M73	GenProg	54	24.2	40.2	66.1%	38.1%
		MGVMRepair	16	8.2	24.9		
	M78	GenProg	120	82.7	118.4	85.2%	58.7%
		MGVMRepair	53	12.2	48.9		
	M80	GenProg	27	17.8	23.8	78.1%	52.9%
		MGVMRepair	14	3.9	11.2		
	M81	GenProg	19	20.2	34.1	65.8%	49.3%
		MGVMRepair	16	6.9	17.3		
	M82	GenProg	128	27.3	33.5	57.1%	35.5%
		MGVMRepair	48	11.7	21.6		
	M84	GenProg	125	90.1	106.5	82.4%	74.2%
		MGVMRepair	59	15.9	27.5		
	M85	GenProg	30	13.8	26.3	47.1%	13.7%
		MGVMRepair	18	7.3	22.7		
	M95	GenProg	87	21.8	27.1	89.9%	80.8%
		MGVMRepair	9	2.2	5.2		
Time	T4	GenProg	36	11.2	19.6	23.2%	8.7%
		MGVMRepair	27	8.6	17.9		
	T11	GenProg	24	7.3	10.7	34.2%	15.0%
		MGVMRepair	11	4.8	9.1		

25 buggy programs that can be repaired by MGVMRepair and GenProg. The data collected for each buggy program is the average of 10 runs in order to reduce the effect of accidental factors during the repair.

As observed in Table 2, in terms of the number of candidate patches generated, GenProg needs to generate 53 candidate patches per buggy program repaired on average, while MGVMRepair needs to generate only 23 candidate patches per buggy program repaired on average. Compared with GenProg, MGVMRepair generates 56.6% fewer NCPs per buggy program repaired, which indicates that MGVMRepair can find valid patches for buggy programs after fewer evolutions. In terms of total program repair time, the longest time taken by MGVMRepair to repair a buggy program was 48.9 minutes and the shortest time was 5.2 minutes, while the longest time taken by GenProg to repair a buggy program was 118.4 minutes and the shortest time was 10.7 minutes. MGVMRepair took only 458.6 minutes to fully repair the 25 buggy programs, with an average of one buggy program every 18.34 minutes. Compared with repair time consumption of GenProg, the total time efficiency of MGVMRepair improved by 8.7% to 81.7%, and the total repair time efficiency increased by 57.1% on average, which indicates that MGVMRepair has a better advantage in the time efficiency of repairing buggy programs.

In terms of candidate patch validation, the total time taken by GenProg to validate all individuals was 945.7 minutes, with an average of 37.83 minutes per buggy program validated, while the total time taken by MGVMRepair to validate all individuals was 164.5 minutes, with an average of 6.58 minutes per buggy program validated. Compared with the GenProg, the total time efficiency of MGVMRepair improved by 23.2% to 96.8%, and the time efficiency increased by 73.8% on average. The test case prioritization technique is used to improve the efficiency of individual verification, which speeds up the process of the individual verification.

Figure 7 illustrates the total time spent on fault repair for MGVMRepair and GenProg to visually compare the efficiency of the repair and further explore the differences between different fault program. In

this graph, the x-axis represents the fault programs that can be repaired and the y-axis represents the total time spent on the fault repair (Unit: minute). According to Figure 7, it can be seen intuitively that the total time taken by MGVMRepair to repair the buggy program is much smaller than the total time required by GenProg, which indicates that MGVMRepair has a great efficiency.

Figure 8 shows the NCP comparison between MGVMRepair and GenProg on repairable faults. In this graph, the x-axis represents the fault program and the y-axis represents the number of candidate patches generated to repair the fault program. According to Figure 8, it can be seen that the number of NCPs generated during the repair process is not the same either between different fault programs or between different fault items. The different number of NCPs indicates that the difficulty of their repair is not consistent. Since the dataset used in the experiment originated from actual development, the project development time is long and large, and the types and locations of errors vary contained in each defective program, so the data fluctuation in the repair process will occur to different degrees. The smaller the NCP value indicates that the repair method has fewer repair attempts to find a valid patch for the defective program, and it is clear that MGVMRepair has better repair results from Figure 8.

Figure 7
MGVMRepair and GenProg run time-consuming line graphs on repairable bugs

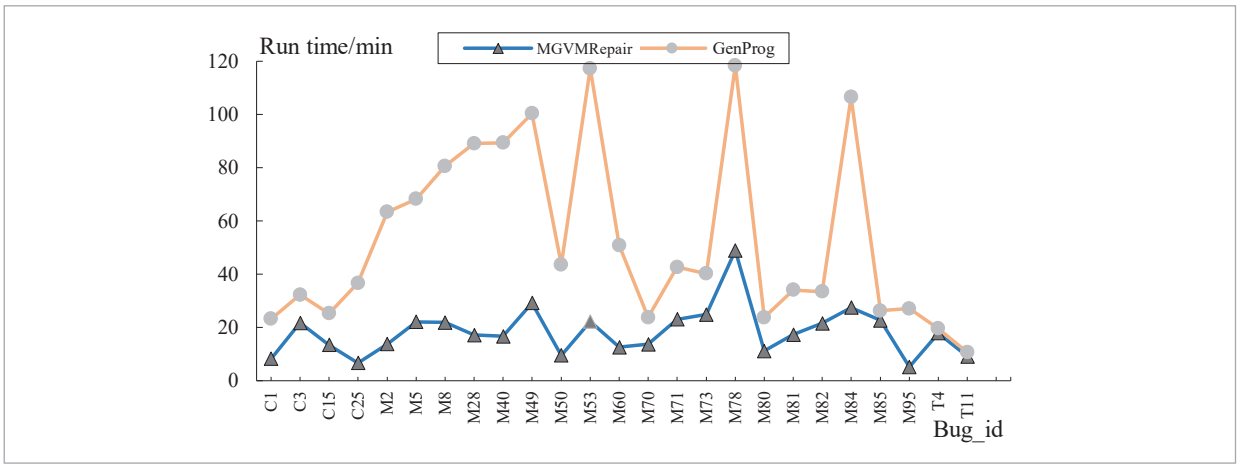


Figure 8
Comparsion of NCP values between MGVMRepair and GenProg on repairable faults

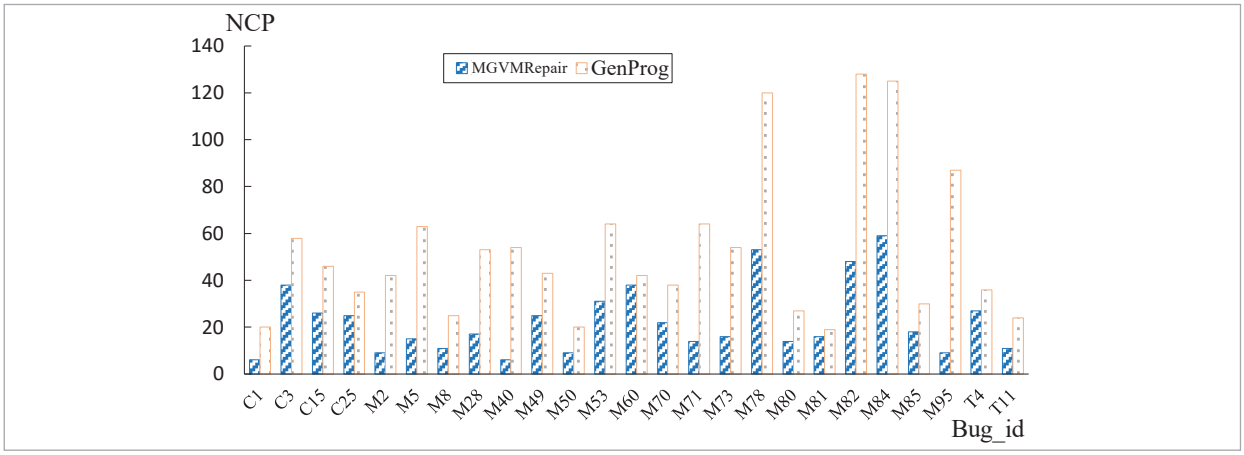


Table 3
Repair Quantity Comparison between MGVMRepair and CapGen, SimFix, jKali, jMutRepair, SketchFix

Project	Total number of faults	MGVMRepair	CapGen	SimFix	jKali	jMutRepair	SketchFix
Chart	26	8	4	4	6	4	8
Math	106	25	16	14	14	11	8
Lang	65	4	5	9	0	1	4
Time	27	3	0	1	2	1	1
Total	224	40	25	28	22	17	21
Success rate	-	17.86%	11.16%	12.50%	9.80%	7.59%	9.40%

Compared with the GenProg, MGVMRepair overcomes the drawbacks of the coarse-grained repair approach by using materials in an overly simplistic manner to increase the successful repair rate and the time efficiency of the buggy programs. And MGVMRepair uses the test case prioritization technique which improves the efficiency of individual verification to reduce the total time taken to repair the buggy program.

5.2. Comparison Between MGVMRepair and Other Existing Repair Approaches

To further validate the effectiveness of repair, we compared MGVMRepair with four fault repair approaches, SimFix [9], CapGen [23], jKali [16], jMutRepair [16] and SketchFix[7], as shown in Table 3. We choose these five repair approaches to compare with MGVMRepair because they are typical approach in repair field. By analyzing the source code of the five methods, the time complexity of MGVMRepair is $O(n^3)$, while that of SimFix, CapGen, jKali, jMutRepair and SketchFix are $O(2^n)$, $O(n^3)$, $O(n^2)$, $O(n^3)$ and $O(n^3)$, respectively. From the table, it can be seen that MGVMRepair successfully repaired 40 buggy programs with a success rate of 17.86%. But in other approaches, only up to 28 buggy programs were repaired with a maximum success rate of 12.50%. In order to reduce the influence of chance factors, the data collected for each defect program is the average of 10 runs.

In terms of the number of successful fixes for buggy programs, MGVMRepair fixes 15, 12, 18, 23 and 19 more buggy programs than CapGen, SimFix, jKali, jMutRepair and SketchFix, respectively, with 60%, 43%, 82%, 135% and 90% higher fix success rates, respectively. The reason for the better repair results of MGVMRepair is the ability to filter ingredient statements from a huge space using similarity as a constraint and the ability to use repair ingredients at a fine-grained level. The MGVMRepair repair method not only finds ingredient statements that meet the constraints quickly but also applies the ingredient in a more appropriate manner, increasing the success rate of the fault repair process.

Figure 9 shows the comparison of the number of fixable fauts between MGVMRepair and CapGen, SimFix, jKali, jMutRepair and SketchFix on the four items Chart, Math, Lang, and Time. The details of the repairable fault procedures of MGVMRepair and the other five repair approaches on Chart, Math, Lang, and Time were further counted, as shown in Table 4. It can be visually seen that MGVMRepair is able to fix more of buggy programs than the existing fault repair approaches from Figure 9 and Table 4.

MGVMRepair has the better repair results because it is able to filter ingredient statements from a huge space using similarity as a constraint and because it is able to use repair ingredients at a fine-grained level. The MGVMRepair not only finds ingredient statements that meet the constraints quickly, but also applies the ingredient in a more appropriate manner, lead to increasing the success rate of repair process. To sum up, MGVMRepair can obviously improve the repair success rate of buggy programs.

Figure 9
Comparison of the number of repairable faults between MGVMRepair and other repair approaches on Chart, Math, Lang, and Time

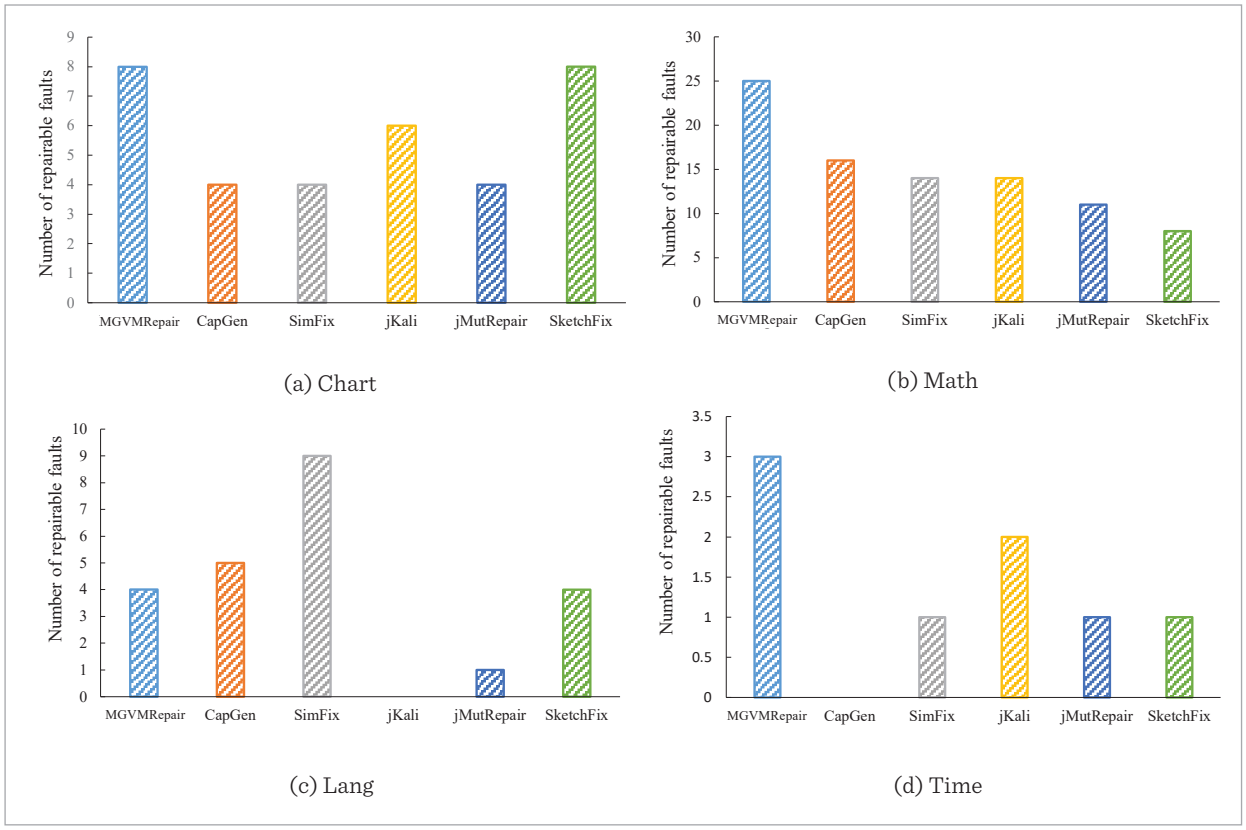


Table 4
Comparison of MGVMRepair and Other Five Repair Approaches Fault Program Repair Situation

Project	Number of faults	Repairable fault program					
		MGVMRepair	***CapGen***	***SimFix***	***jKali***	***jMutRepair***	***SketchFix***
Chart	26	C1,C3,C6,C1,C12, C15,C24,C25	C1,C8, C11,C24	C1,C3, C7,C20	C1,C5,C13, C15,C25, C26	C1,C7, C25, C26	C1,C8,C9,C11, C13,C20,C24,C26
Math	106	M2,M5,M8,M20, M28,M30,M40,M44, M49,M50,M53,M57, M60,M63,M70,M71, M73,M78,M80,M81, M82,M84,M85,M95, M97	M5,M30,M33, M53,M57,M58, M59,M63,M65, M70,M75,M79, M80,M81,M82, M85	M5,M33,M35, M41,M50,M53, M57,M59,M63, M70,M71,M75, M79,M98	M2,M8, M28,M32,M40, M49,M50,M78, M80,M81,M82, M84,M85,M95	M2,M28,M40, M50,M57,M58, M81,M82,M84, M85,M88	M5,M33,M50,M59, M70,M73,M82,M85
Lang	65	L7,L22,L24,L27	L6,L26,L43,L57, L59	L16,L27,L33, L39,L41,L43, L50,L58,L60	-	L27	L6,L51,L55,L59
Time	27	T4,T9,T11	-	T7	T4, T11	T11	T4
Total	224	40	25	28	22	17	21

5.3. Summary

In a word, our approach has the higher efficiency than the compared CapGen, SimFix, jKali, jMutRepair and SketchFix from the above investigation. Because GenProg, jKali and jMutRepair approach work at the statement level, and their repair granularity is too coarse, their repair is lower efficiency than that of ours. The granularity of repair of SimFix, CapGen and SketchFix is fine-grained, the search space of the candidate patches is so large, resulting in a lower repair efficiency of program repair than that of ours.

6. Conclusion and Future Work

In this paper, we propose MGVMRepair, an automatic repair method based on hybrid granularity and variable mapping for fault program. MGVMRepair follows the general framework of random search algorithm, which is able to apply ingredients at fine granularity level after filtering out the repair materials that meet the constraints. Also, a test case prioritization technique based on modification point execution information is proposed to further improve the efficiency of program verification. Experimental results on Defects4J show that the repair efficiency of MGVMRepair is higher than the existing program repair approaches, GenProg, CapGen, SimFix, jKali, jMutRepair, SketchFix.

Under the actual scenario of fixing faulty program, there is often more than one fault in the program. However, the current fault repair approaches have focused on exploring the automatic repair of single-fault programs, and the automatic repair for multi-fault programs is slightly under-researched. In the future, we want to consider the automatic repair of software faults under a multi-fault environment.

Acknowledgements

This work was partially supported by National Natural Science Foundation of China (Nos.62276091, 62206087, 61602154), Cultivation Programme for Young Backbone Teachers in Henan University of Technology, Key scientific research project of colleges and universities in Henan Province (No.22A520024), Major Public Welfare Project of Henan Province (No.201300311200) and Key Laboratory of Grain Information Processing and Control (Henan University of Technology), Ministry of Education (No. KFJJ2022006).

References

1. Abreu, R., Zoeteweij, P., Van Gemund, A. J. C. An Evaluation of Similarity Coefficients for Software Fault Localization. Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing (PRDC 2006), Riverside. USA, Dec, 2006, 39-46. https://doi.org/10.1109/PRDC.2006.18
2. Abreu, R., Zoeteweij, P., Van Gemund, A. J. C. On the Accuracy of Spectrum-Based Fault Localization. Proceedings of Academic and Industrial Conference Practice and Research Techniques-Mutation (TAICPART-MUTAYION 2007), Windsor, UK, Sept, 2007, 89-98.https://doi.org/10.1109/TAIC.PART.2007.13
3. Afzal, A., Motwani, M., Stolee, K. T., Brun, Y., Le Goues, C. SOSRepair: Expressive Semantic Search for Real-World Program Repair. IEEE Transactions on Software Engineering, 2021, 47(10), 2162-2181.https://doi.org/10.1109/TSE.2019.2944914
4. Chen, M. Y., Kiciman, E., Fratkin, E., Fox, A., Brewer, E. Pinpoint: Problem Determination in Large, Dynamic Internet Services. Proceedings of International Conference on Dependable Systems and Networks (DSN 2002), Bethesda, Maryland, USA, June, 2002, 595-604, http://dx.doi.org/ 10.1109/DSN.2002.1029005.
5. Chen, Z., Kommrusch, S., Tufano, M., Pouchet, L.-N., Poshyvanyk, D., Monperrus, M. SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair. IEEE Transactions on Software Engineering, 2021, 47(9), 1943-1959. https://doi.org/10.1109/TSE.2019.2940179
6. Forrest, S., Nguyen, T., Weimer, W., Le Goues, C. A Genetic Programming Approach to Automated Software Repair. Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation (GECCO 2009), Montreal, Québec, Canada, July, 2009, 947-954. https://doi.org/10.1145/1569901.1570031
7. Hua, J., Zhang, M., Wang, K., Khurshid, S. SketchFix: A Tool for Automated Program Repair Approach using Lazy Candidate Generation. Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018), Lake Buena Vista FL, USA, Oct, 2018, 888-891.https://doi.org/10.1145/3236024.3264600

8. Jiang, J., Ren, L., Xiong, Y., Zhang, L. Inferring Program Transformations from Singular Examples via Big Code. Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE 2019), San Diego, CA, USA, Nov, 2019, 255-266.https://doi.org/10.1109/ASE.2019.00033

9. Jiang, J., Xiong, Y., Zhang, H., Gao, Q., Chen X. Shaping Program Repair Space with Existing Patches and Similar Code. Proceedings of the 27th ACMSIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018), Amsterdam, Netherlands, Jul, 2018, 298-309. https://doi.org/10.1145/3213846.3213871

10. Jones, J. A., Harrold, M. J., Stasko, J. Visualization of Test Information to Assist Fault Localization. Proceedings of the 24th International Conference on Software Engineering (ICSE 2002), Orlando, Florida, USA , May, 2002, 467-477. https://doi.org/10.1145/581396.581397

11. Just, R., Jalali, D., Ernst, M.D. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. Proceedings of the 23th International Symposium on Software Testing and Analysis (ISSTA 2014), Seattle, WA, USA, July, 2014, 437-440. https://doi.org/10.1145/2610384.2628055

12. Kim, D., Nam, J., Song, J., Kim, S. Automatic Patch Generation Learned from Human-Written Patches. Proceedings of the 35th International Conferenceon Software Engineering (ICSE 2013), San Francisco, USA, May. 2013, 802-811. https://doi.org/10.1109/ICSE.2013.6606626

13. Le Goues, C., Dewey-Vogt, M., Forrest, S., Weimer, W. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for 8 Each. Proceedings of the 34th International Conference on Software Engineering (ICSE 2012), Zürich, Switzerland, Jun, 2012, 3-13. https://doi.org/10.1109/ICSE.2012.6227211

14. Le Goues, C., Nguyen, T., Forrest, S., Weimer, W. GenProg: A Generic Method for Automatic Software Repair. IEEE Transactions on Software Engineering, 2012, 38, 54-72. https://doi.org/10.1109/TSE.2011.104

15. Le, X. B. D., Lo, D., Le Goues, C. History Driven Program Repair. Proceedings of International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016), Osaka, Japan, Mar, 2016, 213-224. https://doi.org/10.1109/SANER.2016.76

16. Martinez, M., Monperrus, M. Astor: A Program Repair Library for Java. Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016), Saarbrücken, Germany, July, 2016, 441-444. https://doi.org/10.1145/2931037.2948705

17. Martinez, M., Weimer, W., Monperrus, M. Do the Fix Ingredients Already Exist? An Empirical Inquiry Into the Redundancy Assumptions of Program Repair Approaches. Proceedings of the 36th International Conference on Software Engineering (ICSE 2014), Hyderabad, India, May, 2014, 492-495. https://doi.org/10.1145/2591062.2591114

18. Motwani, M., Soto, M., Brun, Y., Just, R., Le Goues, C. Quality of Automated Program Repair on Real-World Defects. IEEE Transactions on Software Engineering, 2022, 48(2), 637-661. https://doi.org/10.1109/TSE.2020.2998785

19. Pawlak, R., Monperrus, M., Petitprez, N., Noguera, C., Seinturier, L. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. Software Practice and Experience, 2016, 46, 1155-1179. https://doi.org/10.1002/spe.2346

20. Qi, Y., Mao, X., Lei, Y., Dai, Z., Wang, C. The Strength of Random Search on Automated Program Repair. Proceedings of International Conference on Software Engineering (ICSE 2014), Hyderabad, India, May, 2014, 254-265, http://dx.doi.org/10.1145/256822 5.2568254

21. Riboira, A., Abreu, R. The GZoltar Project: A Graphical Debugger Interface. Proceedings of International Academic and Industrial Conference on Practice and Research Techniques(TAIC PART 2010), Windsor, UK, Sept, 2010, 215-218. https://doi.org/10.1007/978-3-642-15585-7_25

22. Weimer, W., Forrest, S., Le Goues, C., Nguyen, T. Automatic Program Repair with Evolutionary Computation. Communications of the ACM, 2010, 53, 109-116. https://doi.org/10.1145/1735223.1735249

23. Wen, M., Chen, J., Wu, R., Hao, D., Cheung, S.-C. Context-Aware Patch Generation for Better Automated Program Repair. Proceedings of the 40th International Conference on Software Engineering, Gothenburg, Sweden, May, 2018, 1-11. https://doi.org/10.1145/3180155.3180233

24. Xuan, J., Martinez, M., Demarco, F., Clement, M., Marcote, S. R. L., Durieux, T., Berre, D. L., Monperrus, M. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. IEEE Transactions on Software Engineering, 2017, 43, 34-55. https://doi.org/10.1109/TSE.2016.2560811

25. Yuan, Y., Banzhaf, W. ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming, IEEE Transactions on Software Engineering, 2020, 46(10), 1040-1067. https://doi.org/10.1109/TSE.2018.2874648