World Academy of Science, Engineering and Technology
International Journal of Computer and Systems Engineering
Vol:18, No:3, 2024

Assertion-Driven Test Repair Based on Priority Criteria

Ruilian Zhao, Shukai Zhang, Yan Wang, Weiwei Wang

Abstract—Repairing broken test cases is an expensive and challenging task in evolving software systems. Although an automated repair technique with intent-preservation has been proposed, it does not take into account the association between test repairs and assertions, leading a large number of irrelevant candidates and decreasing the repair capability. This paper proposes a assertion-driven test repair approach. Furthermore, a intent-oriented priority criterion is raised to guide the repair candidate generation, making the repairs closer to the intent of the test. In more detail, repair targets are determined through post-dominance relations between assertions and the methods that directly cause compilation errors. Then, test repairs are generated from the target in a bottom-up way, guided by the the intent-oriented priority criteria. Finally, the generated repair candidates are prioritized to match the original test intent. The approach is implemented and evaluated on the benchmark of 4 open-source programs and 91 broken test cases. The result shows that the approach can fix 89% (81/91) broken test cases, which are more effective than the existing intent-preserved test repair approach, and our intent-oriented priority criteria work well.

Keywords—Test repair, test intent, software test, test case evolution.

I. Introduction

As we all know, the program under test (PUT) is frequently altered due to bug fixing, refactoring, and incremental development activities in software evolution. Regression testing (RT) is essential to ensure the quality of evolving software. In RT, test cases are re-executed on the latest version to check whether the changes of PUT bring in new bugs. However, changes to the PUT may result in some test cases no longer being available [1]. Such test cases are called *broken test cases*. It can be observed that even small modifications can lead to lots of broken test cases, in some cases up to 74% of test suite[2], which leads to significant challenges in the maintenance of the test suite [3]. Hence, how to automatically repair broken test cases has been a critical problem in RT.

Currently, test case repair techniques are mainly divided into two categories. One focuses on the broken test cases caused by run-time exceptions and assertion failures. For instance, ReAssert [1] fixes the failed assertion of broken test cases through executing the test and observing run-time behavior. Symbolic Test Repair [4] uses symbolic execution to compute the expected values in the assertions, then leverages the expected values to renovate assertions. The other category of test repair techniques aims at compilation errors of test cases. For example, TCA [5], [6] combines static and dynamic program analysis to correct compilation errors in broken test cases caused by method signature changes, such as adding parameters, removing parameters, and changing the type of parameters or return. TestFix [7] uses a genetic algorithm to find a sequence of method invocations to fix a broken test case, making the broken test case pass.

In fact, run-time exceptions or assertion failures may cause test cases to fail only after the test cases are successfully compiled. On the other hand, previous researches on test case evolution have shown that more than 50% of the repaired test cases are related to compilation errors while less than 10% are associated with assertion changes [8]. Therefore, this paper aims at the broken test case caused by compilation errors and studies how to repair the test cases.

As mentioned above, there are some test case repair techniques for compilation errors. However, TCA's repair capability is limited because it mainly concerns fixing the broken test cases caused by method signature modification. TestFix does not pay close attention to the test intent, so it does not guarantee that the repaired test cases still keep their original test intent. Li et al. [9] proposed an intent preserving test repair (TRIP) approach, which generates repair candidates by a top-down approach and leverages logic expressions of path conditions [10] to model test intent.

However, the existing test repair approaches face many challenges. Firstly, test intent can be derived from explicit intent and implicit intent, where implicit intent is reflected in the execution path of test case, while explicit intent is embodied in the test assertion. TRIP adopts a top-down repair generation method, without considering the association between test repairs and assertions, resulting in a lot of irrelevant test repairs. Secondly, the more elements reused in test case repair, and the fewer new elements added, the closer the test case repair candidate is to the original test case intent; namely, the smaller the difference between the repair candidate and the original test case, the better the repair. But the intent-oriented priority criteria of TRIP only consider the newly added elements, and do not consider the reused elements. In addition, the intent-oriented priority criteria of TRIP do not take into account repair costs, which will result in some ineffective repairs and increase repair costs.

Therefore, this paper proposes a Assertion-Driven Test Repair approach (TRAD) for unit test cases of object-oriented software. TRAD proceeds from assertions, and determines the repair target related to assertions through post-dominance relations. Furthermore, an intent-oriented priority criterion is raised from the perspectives of reusing elements and reducing

Ruilian Zhao, Shukai Zhang, Yan Wang, and Weiwei Wang* are with the College of Information Science and Technology, Beijing University of Chemical Technology, Beijing, 100029 China (*corresponding author, e-mail: wangww@mail.buct.edu.cn).

repair costs, which considers both newly added elements and reused elements, as well as the repair cost. On this foundation, TRAD generates repair candidates from the repair target in a bottom-up manner under the guidance of the intent-oriented priority criteria, preserving the test intent and avoiding irrelevant repairs. In order to verify the validity of our test repair approach, TRAD is compared with the TRIP on the same benchmark, which contains four open-source programs and 91 broken test cases. The experiment results show that TRAD repaired 89% (81) broken test cases, more than 79% (72) that TRIP repaired. Moreover, TRAD approach can not only repair all the broken test cases that TRIP can repair, but also repair some broken test cases that TRIP cannot repair.

The contributions of this work are summarized below:

- An assertion-driven test repair approach is proposed, which uses the post-dominance relation to determine the repair targets, and devises an intent-oriented priority criterion to guide the test repairs generation from the repair target in a bottom-up way.
- Considering the perspective of reused elements and the repair costs, three intent-oriented priority criteria are proposed to guide the repair candidates generation, improving the quality of the repair candidates.
- A series of empirical experiments is conducted to evaluate the effectiveness of our approach. And the experiment results show that TRAD repaired 89% (81) broken test cases, better than the existing intent-preserved test repair approach.

The rest of this paper is organized as follows. Section II describes relevant concepts. Section III presents an example to explain our approach. In Section IV, our approach is described in detail. The experiments and findings are reported and discussed in Section V. Section VI illustrates threats to validity. In Section VII, related works are depicted in section. Section VIII is the conclusion and future work.

II. Preliminaries

This section introduces basic concepts that are relevant to the work in this paper.

A. Broken Test Case and Test Intent

In object-oriented software, test code inspect program functionality by invoking methods and accessing field in the PUT. So, a test case includes two parts: method sequences and test oracles. The method sequences consist of the methods invoked and the fields accessed. Test oracles are often in the form of executable assertions such as in the JUnit testing frameworkm[11], checking the return value of method invocations or the state of receiver objects to check the correctness of the program.

In this paper, a broken test case refers to the test case which does not compile and is destroyed by compilation errors. It means that for programs P and its modified version P', a test case t that can be executed on P is not compilable for P' due to compilation errors.

In intent-preserving test case repair, it is necessary to find a way to model the original test intent. Intuitively, test intent is reflected in the program behavior of the test and the expected result. The former represents the implicit intent of the test, while the latter represents the explicit intent of the test.

Since path conditions abstract the behavior of the program exercised by a test case, it can express the implicit intent of a test case. On the other hand, test assertion is to verify whether the program meets the expected result under this executable path. Thus, the explicit intent is embodied in the test assertion.

B. Public Program Element

The public program elements (PPEs) are basic elements (i.e., method or field in Java) that a test case can access in object-oriented software. Test cases validate program functionality through invoking methods and access fields in the program under test. Compared a program P and its new version program P', basic elements can be classified as *removed PPEs* (in P but not in P'), *unchanged PPEs* (both in P and P') and *new PPEs* (only in P'). It is observed that *removed PPEs* is the main reason that test case t does not compile on P', and *new PPEs* and *unchanged PPEs* may be used to repair the broken test case t.

C. Static Data Flow Graph of Test case

As mentioned above, a test case is made up of methods invocations sequence, and *PPEs* are the basic elements it can access. A *PPE* may be method invocations, field accesses, and built-in language operators, which involves method names and associated inputs and outputs. In order to describe the methods names, associated inputs and outputs, and the data dependencies among methods, and help identify the entities affected by removed *PPEs* in broken test cases, a *static data flow graph (DFG)* is provided. To be specific, a DFG is a bipartite graph which is defined as a 3-tuple $\langle N_{def}, N_{op}, E\rangle$, where N_{def} is a set of nodes representing *definition*s, N_{op} is a set of nodes representing *operation*s, and $E \in (N_{def} \times N_{op}) \cup (N_{op} \times N_{def})$ is a set of directed edges representing the relations between *definition*s and *operation*s. A *definition* can be a literal constant or a definition generated by an *operation*, and *operation*s are the methods invoked by a test case. An edge $d_1 ß op_1$ from a datum $d_1 \in N_{def}$ to a *operation* $op_1 \in N_{op}$ denotes that d_1 is a parameter of *operation* op_1, namely *operation* op_1 data depend on d_1. Similarly, an edge $op_1 ß d_1$ from a *operation* $op_1 \in N_{op}$ to a datum $d_1 \in N_{def}$ denotes that d_1 is the return of *operation* op_1. In addition, DFG label *definition* nodes with their type and *operation* nodes with the associated *PPEs*.

The DFG of a broken test case is called *broken DFG*. In a broken DFG, the *operation*s that directly cause compilation errors are named as *invalid operations*, which must be removed and correspond to removed PPEs in the broken test case. The *definition*s that are affected by *invalid operation*s directly or indirectly are called as *affected definitions*. And the *definition*s that do not affect by *invalid operation*s are available in repairing a broken test case.

For example, Fig. 1 shows the DFG of a test case, where rectangles express *operation* nodes, and rounded rectangles represent *definition* nodes. Nodes 0, 1, 2, 3, 4, 5 correspond to

well-formed nodes; nodes 6 and 12 correspond invalid nodes; nodes 7, 9, 11, 13 correspond affected definition nodes.

III. Case Study

In this section, we conduct a case study on a simple broken test case that involves compilation errors to illustrate our test repair method. We assume that for programs P and P', test case t is executable on program P but is broken on program P' because of compilation errors. The broken test case is shown in Fig. 2 (a), which has three non-compilable methods striking in red, that is, method *readlines* in line 6, method *check* in line 7. The corresponding DFG is shown in Fig. 1. The non-compilable methods are associated with the node 6 and node 12 in red by dashed border, which are *invalid operation*s in the DFG.

The affected definitions nodes of these two *invalid operation*s directly or indirectly are nodes 7, 9, 11 and 13, respectively. The reusable nodes in the broken test case, that is, nodes other than *invalid operation*s, are all operations in the DFG except nodes 6 and 12. The repair of this test case requires the use of the newly added methods in program P', including $getContent$ method and $checkFormat$ method. Their corresponding nodes are shown in blue boxes in Fig. 3.

In order to have a comparison, we first give the process by which TRIP fixes this test case.

TRIP first determines an initial partial DFG consisting of nodes 0, 1, 2, 3, 4, 5, and then expands the partial DFG by adding operation nodes with well-formed definitions as parameters to the partial DFG. Each time an expanded DFG is selected, it tries to replace the affected definition nodes in the broken DFG, i.e., nodes 7, 9, 11, 13, with well-formed definition nodes in the current DFG. When a partial DFG can make the broken DFG have no affected definition nodes after replacement, it is considered to have obtained a repair candidate. In fact, TRIP may generate repairs for nodes 7, 9, 11, 13, which means that node 8 preserved in the correct repair may be replaced by other operations that can generate defination node of the same type as node 9 ($boolean$ type). Furthermore, TRIP does not consider reuse in the priority criteria, which may lead to adding operations that are different from operation node 4, but can generate MyFile type definition nodes, and use this definition node to add node 17 in the correct repair or other wrong operations. In addition, TRIP does not consider the repair cost. These factors will eventually cause TRIP to generate invalid or irrelevant repairs.

In contrast, our proposed assertion-driven approach can start from the affected definitions in the assertions, and obtain the repair target, i.e., nodes 7 and 13, through the post-dominance relation. This can avoid the wrong repair of operation node 8 related to the assertion, and preserve some test intent. Furthermore, the reusable elements and repair cost are considered in the priority criteria, which can filter out some case where wrong parameters are added to the operation nodes, and reduce the invalid repairs. Moreover, the bottom-up test repair generation starting from the repair target guided by the priority criteria can avoid generating irrelevant repairs for node 11. And when filtering repair candidates, we also select the ones that are closest to the original broken test under the new priority criteria. Through the bottom-up test repair generation and the new priority criteria guidance, TRAD can generate better test repairs more efficiently.

IV. Approach

In this section, we describe our assertion-driven test repair approach, TRAD, in detail. TRAD takes program P, the updated version of program P', and a broken test case t as input, and outputs a series of repair candidates for test case t. Fig. 4 shows the skeleton of TRAD, which consists of three parts: (1) *Repair Target Determination*: The *repair target* is a set of affected definitions in the broken DFG, determined by the post-dominance relationship between the assertions and the invalid operations.

(2) *Intent-Oriented Priority Criteria*: TRAD raises three priority criteria to select the partial DFG closer to the test intent for expansion when generating repairs. Furthermore, after repaired DFG are generated, the priority criteria are used to select repair candidates closer to the test intent.

The three prioritization criteria are proposed considering i. the source of the added operation; ii. the cost of the repair; and iii. the similarity between the added operation and the invalid operation. (3) *Candidate Repaired DFGs Generation*: After deleting the nodes that need to be removed from the broken DFG, the remaining part is an initial partial DFG. In this initial partial DFG, TRAD uses a bottom-up approach to gradually extend the partial DFG from the repair target, guided by the priority criteria, until the partial DFG becomes a candidate repaired DFG.

A. Repair Target Determination

Intuitively, all repaired DFGs should reach test assertions. Thus, the test repair process can start from the test assertions in a bottom-up way. But as mentioned, the smaller the difference between the repair candidate and the original test case, the better the test repair. That is, the more reusable nodes linking the assertion is kept, the better the test repair. So, in order to better preserve the intent of test repairs, we use the post-dominance relationship to determine the *repair targets* considering the association between the invalid operations and the test assertions.

Before introducing the formal definition of post-dominance relationship, we illustrate three basic concepts that are *entry nodes*, *exit nodes* and *path* in the DFG. The *entry nodes* of DFG are definitions represented literal constants or operations without parameters, which in-degree is 0. The *exit nodes* are end points of DFG, which out-degree is 0. A *path* in a DFG is a finite sequence of nodes n_1, n_2, ...,n_k such that for i=1,2,...,k-1, there is an directed edge from n_i to n_{i+1}. The formal definition of post-dominance relationship is described as:

***Definition** 1: Post-dominance relationship:* A set of nodes S_1=$\{n_1, n_2, ..., n_i\}$ post-dominates another set of nodes S_2=$\{m_1, m_2, ..., m_j\}$ if and only if i. every directed path from any node $m \in S_2$ to any reachable exit node contains a node $n \in S_1$ and ii. any node $n \in S_1$is in a directed path from a

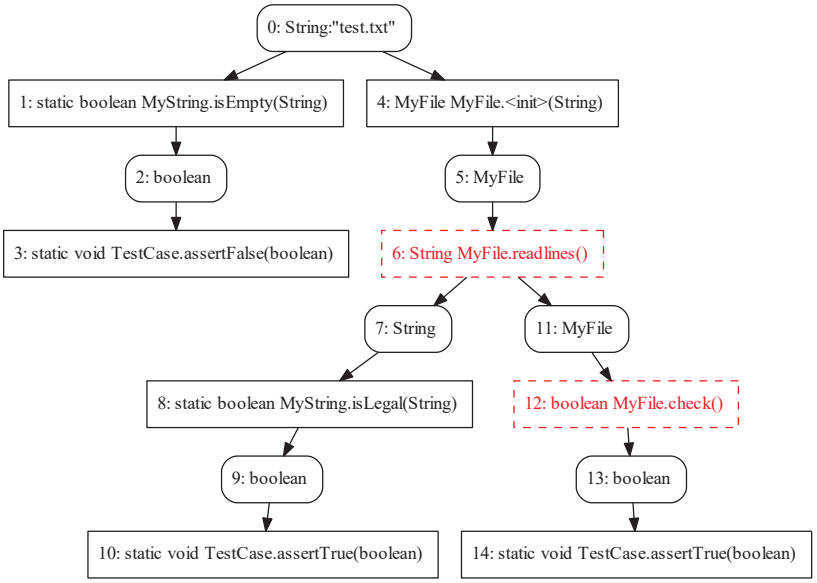

Fig. 1 DFG of a test case

```java
public void testMyFile() throws Exception{
    String filename = "test.txt";
    boolean empty = MyString.isEmpty(filename);
    assertFalse(flag);
    MyFile file = new MyFile(filename);
    String content = file.readlines();
    boolean isValid = file.check();
    assertTrue(isValid);
    boolean isLegal = MyString.isLegal(content);
    assertTrue(isLegal);
}
```

(a) A broken test case

```java
public void testMyFile() throws Exception{
    String filename = "test.txt";
    boolean empty = MyString.isEmpty(filename);
    assertFalse(flag);
    MyFile file = new MyFile(filename);
    String content = file.readContent();
    boolean isLegal = MyString.isLegal(content);
    assertTrue(isLegal);
    boolean isValid = MyFile.checkFormat(filename);
    assertTrue(isValid);
}
```

(b) The repaired test case

Fig. 2 An example of test case repair

node $m \in S_2$ to an exit node and $iii.$ for any node $n \in S_1$, there must exist a path to the exit node that does not contain any node in S_2.

According to the post-dominance definition, exit nodes post-dominate all nodes in the DFG. Assertions often are exit nodes of DFG, except in some cases that assertions have no data dependency with method sequences, such as using try-catch to check exceptions with fail statements. Hence, TRAD regards affected definitions used in exit nodes as the initial repair target. Then, TRAD uses the replacement relationship between the affected definitions to obtain the final repair target.

***Definition** 2: Replacement relationship:* For a set of nodes S that post-dominates all invalid operations, and a valid operation op, if a set of definition nodes n_1 satisfies $n_1 \in S$ and all nodes in n_1 are directly generated by op, and a set of definition nodes n_2 satisfies n_2 are all affected definitions and n_2 are parameters of op, then we say that n_2 can replace n_1 in S. It is easy to prove that the replaced S still post-dominates all invalid operation nodes.

According to the replacement relationship, TRAD starts from the initial repair target and continuously searches for replaceable nodes to obtain new repair target. After there are no replaceable nodes in the repair target, TRAD obtains the final repair target. The difference between the final repair target and the initial repair target is that the final repair target avoids the wrong repair of some affected definition nodes. Intuitively, the valid operations op used in the replacement process are often the operations that the test intent wants to preserve, while directly using the initial repair target to generate the repaired DFG may cause these operations not to be used.

In Fig. 1, nodes 6 and 12 are invalid operations that correspond to non-compilable methods. The initial repair target is the node set that includes nodes 9 and 13. There is a replacement relationship between node 9 and node 7. After replacement, we obtain a repair target consisting of nodes 7 and 13.

B. Intent-Oriented Priority Criteria

Ideally, the more elements reused in the test repair, and the fewer new elements added, the closer the test intent of repair candidate is to that of the original test case; namely, the smaller the difference between the repair candidate and the original test case, the better the repair. Furthermore, the higher the repair cost of that test repair is, and the less likely it is to be a repair that meets the test intent. Thus, in order to make the test intent of the repair candidate more similar to the original test cases, we design intent-oriented priority criteria from the perspectives of newly added elements and reused elements, as well as the repair cost.

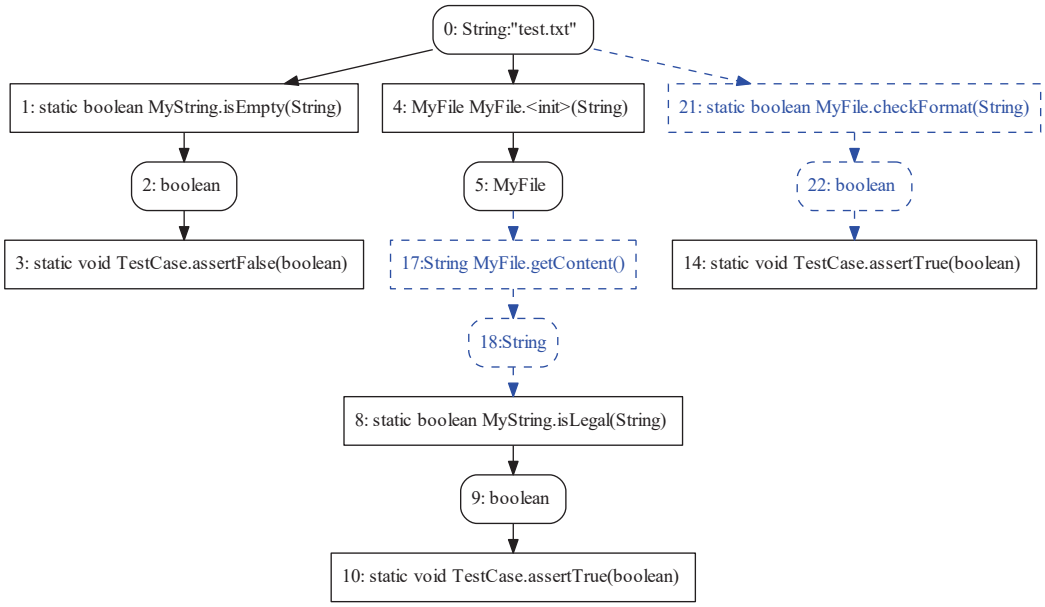

Fig. 3 The repaired DFG

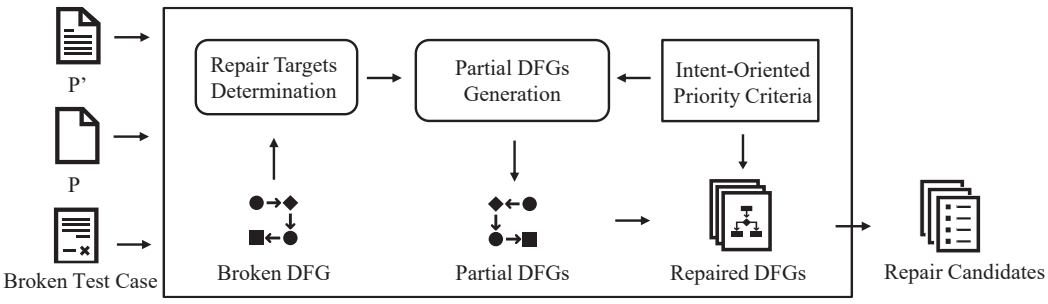

Fig. 4 The skeleton of TRAD

Detailed descriptions of our priority criteria are discussed as follows.

When generating repaired DFGs, in order to reduce repair costs, we think that the more unassigned definitions there are and the more of them that cannot find well-formed definitions of the same type in the partial DFG, the higher the repair cost of that DFG is, and the less likely it is to be a repair that meets the intent. Equation (1) is used to calculate the priority of the partial DFG:

$$f(dfg) = \frac{\sum_{udef \in dfg} weight_{udef}}{num_{udef}} \quad (1)$$

where $udef$ means the unassigned definitions, $weight_{udef}$ signifies the weight of unassigned definitions. The unassigned definitions that have well-formed definitions of the same type in partial DFG have a weight of 0.9, and those that do not have a weight of 0.1. num_{udef} means the number of unassigned definitions.

Furthermore, we think that *i.* adding operations that are new PPEs are more likely to be correct repairs that meet the test intent, adding operations that exist in the broken test case itself (i.e., except for invalid operations) are more likely to be correct repairs that meet the test intent, and other operations are less likely to be correct repairs that meet the test intent. *ii.* adding operations that are closer to the number of invalid operations are more likely to be correct repairs. *iii.* adding operations that have higher similarity with invalid operations are more likely to be a correct repair. Formula (2) is given to calculate the priority of DFG.

$$g(dfg) = \frac{\sum_{insop \in dfg} weight_{insop} * similarity_{insop}}{diffnum_{op} + 1} \quad (2)$$

where $insop$ refers to the added operations in the repaired DFG, $weight_{insop}$ refers to the weight of the added operations. When the added operation is $NPPE$, weight is 0.9, when the added operation is an operation that exists in the broken test

case itself, weight is 0.7, and when the added operation is other operations, weight is 0.5. When adding fields, weight is also assigned similarly; $diffnum$ refers to the absolute value of the difference between the number of added operations and invalid operations, and $similarity_{insop}$ refers to the similarity between the added operations and the invalid operations. The similarity formula is as follows:

$$similarity_{insop} = \max_{i=1}^{k}(s_1(insop, invop_i)+s_2(insop, invop_i)) \quad (3)$$

where $invop$ refers to the invalid operations. Moreover, the similarity between an added operation and each invalid operation consists of two parts. $s_1(insop, invop_i$ is *Levenshtein ratio* of fully-qualified declaration between added operation and invalid operations shown in (4). $s_2(insop, invop_i$ is *Levenshtein ratio* of method name between added operation and invalid operations shown in (5). The value of α in TRAD is 0.4.

$$s_1(insop, invop_i) = \alpha * Leven\_decl(insop, invop_i) \quad (4)$$

$$s_2(insop, invop_i) = (1-\alpha) * Leven\_name(insop, invop_i) \quad (5)$$

When generating repair candidates, we use $f(dfg)+g(dfg)$ as the final priority criteria, and when reporting possible repair candidates, we directly use $g(dfg)$ as the priority criteria.

C. Repaired DFG Generation

TRAD generates repaired DFGs from the repair target in a bottom-up way under the guidance of the intention-oriented priority criterion. TRAD starts from the repair target, making the final repair directly related to it, avoiding irrelevant test repairs. Meanwhile, the test intent related to the repair target determined by the assertion is maximally retained, thereby avoiding the repair of invalid operation outside the repair target.

Furthermore, the guidance of priority criteria can make the generated repaired DFGs more in line with test intent and effectively reduce repair costs. The detailed repaired DFG generation is shown in Algorithm 1.

Firstly, the repaired DFG generation process needs to obtain the operations that need to be used when extending the DFG. The function *getOperations* finds all operations involving $NPPE$ and $UPPE$ in the program P' (line 3). Sometimes it also needs to directly introduce field constants from P', so the field constants are also added to the $fields$ set (line 4). Then the function *initPartialDfg* deletes the nodes that need to be removed in the broken DFG, the removed nodes are invalid operation nodes and nodes that are not used in the replacement process and are reachable by invalid operation nodes (line 5). The function *getPartialDfg* takes out a partial DFG with the highest priority from the $partialDfgs$ set (line 7). The specific calculation of priority will be introduced in the next section. The function *isRepairedDfg* determines whether the current partial DFG is a repaired DFG. If yes, it is added to the repaired DFG set. If not, it selects an operation to extend partial DFG(line 10-14). The function traverses the previously

Algorithm 1: Repaired DFG Generation

Input: $NPPEs$; $UPPEs$; P'; DFG: the broken DFG
Output: $repairedDfgs$: a set of repaired DFGs

```
1  begin
2      repairedDfgs = {};
3      operations = getOperations(NPPEs,UPPEs,P');
4      fields = getFields(NPPEs,UPPEs,P');
5      partialDfgs = {initPartialDfg(DFG)};
6      while partialDfgs is not empty and time limit not reached do
7          g = getPartialDfg(partialDfgs);
8          foreach op ∈ operations do
9              extendDfg = extendPartialDfg(g, op);
10             if isRepairedDfg(extendDfg) then
11                 repairedDfg.add(extendDfg);
12             else
13                 partialDfgs.add(extendDfg);
14             end
15         end
16         foreach field ∈ fields do
17             extendDfg = extendPartialDfg(g, field);
18             if isRepairedDfg(extendDfg) then
19                 repairedDfg.add(extendDfg);
20             else
21                 partialDfgs.add(extendDfg);
22             end
23         end
24     end
25 end
```

obtained set of operations, and the function *extendPartialDfg* checks whether the return value type of the current traversed operation op matches the type of an unassigned definition in the current partial DFG. If it matches, the operation is added to the partial DFG and a new DFG is generated. The parameters used by the operation are set to unassigned definitions. When adding an operation, if there are already k unreachable operations with the same name in the partial DFG, then k+1 new partial DFGs are generated, where k are partial DFGs merged with k operations with the same name, and 1 is a partial DFG with a new operation added. After merging with an operation with the same name, the parameters of that operation are no longer set to unassigned definitions. For the $fields$ set, it looks for fields that match the type of unassigned definitions and directly assigns that field as a parameter value. If there is already a field definition node with the same type in that partial DFG, then that definition node is used as that unassigned definition (line 9,17). At the end of the loop, the DFGs in $repairedDfgs$ are converted to code and executed on P'. If they pass, they are reported as repair candidates.

In the example, the repair target is nodes 7 and 13, and the removed nodes are 6, 11, 12. The initial partial DFG is nodes 0, 1, 2, 3, 4, 5, 7, 8, 9, 10, 13, 14. After searching all the operations, the *getContent* method is selected as the

operation node to be added to the initial partial DFG, and its parameter MyFile is set to unassigned definition. For the definition *MyFile*, the operation *MyFile.¡init¿* is selected to be added to the DFG, which has the same name as the operation node 4 in the partial DFG. It is chosen to merge it, that is, directly use node 5 as the parameter of *getContent*. Then, the *checkFormat* method is selected, and its parameter String is set to unassigned definition. For the definition node String, we choose the String field "test.txt" as its assignment. This field node has the same name as node 0 in the partial DFG, so we choose to merge it. At this point, there are no unassigned parameters in the entire partial DFG, so it is a repaired DFG and added to *repairedDfgs*. After converting to code and executing successfully, it is reported as a possible repair candidate after guidance of priority criteria. The following describes the priority criteria when generating repaired DFGs and reporting possible repair candidates.

V. Evaluation

In this section, we describe our experimental setup and findings.

A. Implementation

We implemented our approach based on TRIP[9]. TRIP builds DFGs by Java Parser [12] extracts path conditions by a modified version of Java Pathfinder [13] as dynamic symbolic execution engine. The max time for each broken test case is 5 minutes. Our experiments were conducted on Windows 10 with 8GB RAM and Intel i5-9400 CPU (2.90 GHz).

TABLE I
Programs in Benchmark

Program	*Versions*	*Classes*	*LOC*	*Tests*
commons-lang	8	67–99	36K–52K	1193–2051
commons-math	7	614–990	12K–20K	2379–4587
gson	10	77–96	8K–12K	204–939
joda-time	13	142–157	47K–63K	2420–3838

B. Benchmark

We evaluated our approach on the same benchmark with TRIP. The benchmark contains four open-source programs. Some of them also are used in the previous research of test suite evolution [8]. Table I shows the details of the benchmark.

For each program in the benchmark, P and P' represent a pair of consecutive versions, T and T' represent their corresponding test suites. The benchmark contains 91 broken test cases which test case t' in T' has the same name, and each test case t in P' has compiled error. We judge a repair candidate as *a actual repair* for t or not, depends on whether it is semantically equivalent to t'.

C. Research Questions

Our study aims to generate repair candidates with more similar test intent for a better repair capability. Hence, we ask:

RQ1: *Does TRAD improve the repair capability compared to TRIP?*

RQ2: *How accurate is the selection of the repair targets?*

RQ3: *What is the influence of the priority criteria?*

To answer RQ1, we evaluated the repair capability by the number of actual repairs generated and compared TRAD with TRIP. To answer RQ2, We analyzed the accuracy of the selection of repair targets by comparing the actual repairs. To answer RQ3, we conducted a series of experiments with different combinations of priority criteria to investigate each priority criteria' role. Furthermore, we analyzed the feature of those broken test cases further.

D. Result and Analysis

TABLE II
Empirical Evaluation Results

Program	Broken Test Cases	TRIP	TRAD	Top		
				5	10	30
commons-lang	14	14	14	14	14	14
commons-math	3	1	2	1	1	2
gosn	69	54	61	40	49	57
joda-time	5	3	4	4	4	4
total	91	72	81	59	68	77

1) Answer for RQ1: Table II shows the results. The first column represents the four programs in the benchmark. The second column shows the number of broken test cases in those programs. In the third and fourth column, the numbers represent broken test cases TRIP and TRAD fixed, respectively. To select the actual repair, TRAD ranked repair candidates according to the similarity of path conditions. The last three columns show the number of actual repairs in Top-5, Top-10, and Top-30.

The result shows that TRAD repairs more broken test cases than TRIP. From the table, TRIP fixed 79% (72/91) broken test cases, and TRAD fixed 89% (81/91). TRAD fixed 9 broken test cases more than TRIP and improved almost 10% (9/91) repair rate. After manually checking, TRAD contains all of the broken test cases which TRIP fixed.

TRAD uses the same approach to extract and compare path conditions with TRIP. The actual repairs of TRAD in top-5, top-10, top-30 are 73% (59/81), 84% (68/81), 95% (77/81), compared to 94% (68/72) broken test cases in Top-3 of TRIP [9]. Repair candidates TRAD generated candidates had higher similarities in path conditions, which also proves our approach's effectiveness. The result shows that our approach has better repair capability.

2) Answer for RQ2: We select the repair targets from the affected definition nodes in the assertion through post-dominance and substitution relations.

Because we preserve some of the affected definitions when we obtain the final repair targets from the initial repair targets through replacement relations, the repair targets may be invalid that cause TRAD cannot generates the actual repair. For each broken test case, we check whether the selection of repair target is valid or not. The result in Table III shows that almost 97% (88/91) of the repair target selection is valid.

TABLE III
THE VALIDITY OF REPAIR TARGET D SELECTION

Repair Target	Valid	Invalid	Total
Numbers	88	3	91

TABLE IV
THE NUMBER OF OPERATIONS

Program	Operations	ALL	Available		Actual	
commons-lang	All	2411	2368	0.98	2098	0.87
	new	192	189	0.98	163	0.85
	unchanged	2219	2179	0.98	1935	0.87
joda-time	All	3554	3360	0.95	3056	0.86
	new	48	48	1.00	46	0.96
	unchanged	3506	3312	0.94	3010	0.86
commons-math	All	9028	8768	0.97	7505	0.83
	new	1141	1044	0.92	938	0.82
	unchanged	7888	7724	0.98	6567	0.83
gosn	All	838	710	0.85	590	0.70
	new	212	175	0.82	153	0.72
	unchanged	626	535	0.85	437	0.70

Furthermore, we analyzed the influence of repair targets on operations. An operation is available when all of its parameters can have a proper assignment. The actual operations are that operations may use in the partial DFG in our approach. The result in Table IV shows that the repair target decreases the number of operations. The actual operations are about 70–96% of all operations. The number of operations is the average of broken test cases in the versions of programs.

3) Answer for RQ3: TRAD has three priority criteria: (1) source of the added methods, (2) text-similarity, (3) repair cost. To answer Q3, we conducted three experiments with different priority criteria. The first is TRAD-MS, which represents TRAD without source of the added methods. The second is TRAD-TS, which represents TRAD without text-similarity. Moreover, the final is TRAD-RC which means TRAD without repair cost. Table V shows the results.

TABLE V
THE RESULT WITH DIFFERENT PRIORITY CRITERIA

Program	TRAD	TRAD-MS	TRRP-TS	TRRP-RC
commons-lang	14	14	14	14
commons-math	2	1	2	2
gosn	61	43	55	51
joda-time	4	4	4	4
total	81	62	75	71

Table V, the results with different priority criteria vary a lot. TRAD-MS decreased the most in the number of actual repairs, which only fixed 62 broken test cases. TRAD-RC decreased secondly, which fixed 71 broken test cases. Moreover, TRAD-TS decreased the least, which fixed 75 broken test cases. The result shows that the priority criteria have an important influence on the repair capability.

We conclude that all of the priority criteria have their own advantages and combine them to fix broken test cases.

Fig. 5 shows the composition of the broken DFG and the repaired DFG and relations between them. For a broken test case, n_1 represented the elements in the broken DFG and reused in the repaired DFG, n_2 represents the newly added elements in the repaired DFG, n_3 represents the elements that

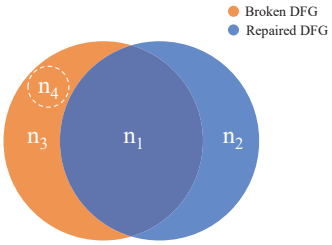

Fig. 5 The composition of DFGs

do not reuse in the broken DFG, n_4 represents the elements about invalid operations that cannot reuse. To represent the feature of broken test cases, we use r_1 represent *the nominal reuse rate* of the broken DFG shown in (6).

$$r_1 = \frac{n_1}{n_1 + n_3} \tag{6}$$

The actual reuse rate of the broken DFG is r_2.

$$r_2 = \frac{n_1}{n_1 + n_3 - n_4} \tag{7}$$

and r_3 represents the reuse rate in the repaired DFG.

$$r_3 = \frac{n_1}{n_1 + n_2} \tag{8}$$

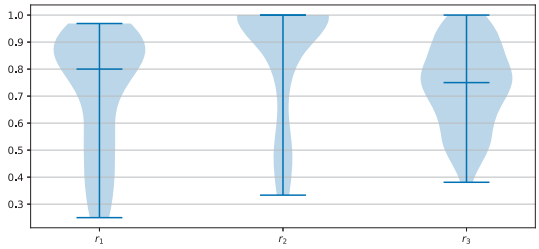

Fig. 6 The distribution of reuse rates

To investigate the feature of broken test cases, we analyzed those broken test cases in the benchmark further. Fig. 6 shows the distribution of reuse rates. As mentioned above, r_1 represents the nominal reuse rate, r_2 represents the actual reuse rate, and r_3 represents the reuse rate in the repaired DFG. The results show that the median of them is all higher than 0.7. The median of r_1 is 0.8, the median of r_2 is 1, which means most broken test cases have a high reuse rate, and a large part of the non-reusable methods are methods directly cause compilation errors. This also indirectly verifies that the method sequence retained by the repair target is basically reusable, that is, the selection of the repair target has a high accuracy rate. The median of r_3 is about 0.75, and the numbers are evenly distributed on both sides of the median.

Finally, we analyzed all 10 broken test cases that our approach cannot fix. Three of them are due to invalid repair targets. Two are due to the side effects of operations added. Two involve methods out of the program, such as $List()$. Two are due to long repair sequences. The last one is due to the parser error of complex type conversion.

VI. Threats to Validity

In this section, we discuss the internal and external threats to the validity of this paper.

Internal: There might be *mistakes in manually matching the repair candidates and the actual repairs* written by developers. To mitigate this threat, we examined results multiple times and compared them to the previous study results. There might be *faults in our implements*. To mitigate this threat, we check the interim results of each step manually.

External: The external threat is that *programs in the benchmark might not be representative*. Those program are used in previous study [8], [9]. They involve different domains—manipulation of the java core class, data and time computation, numeric computation, and serialization/deserialization. All of them are open-source in long-term maintenance. Therefore, we believe that this threat is limited.

VII. Related Work

Daniel et al. proposed a JUnit tests repair tool, ReAssert [1], which analyzes the exception information return by test cases and provides repair suggestions, including replacing literal values in test cases, changing assertion methods, or replacing one assertion with several. In subsequent work [14], they improved the result of ReAssert both quantitatively and qualitatively with Symbolic Test Repair that uses symbolic execution to compute the expected values in the assertions. Both of them focus on runtime exceptions and assertion failures rather than compilation errors.

Mirzaaghaei et al. presented TestCareAssistant (TCA) [5], [6] that fixes non-compilable test cases caused by changes of method signatures, i.e., addition, deletion, or modification of parameters. TCA uses static and dynamic program analysis to identify changes of method signatures and variables and possible initialization for variables. TCA restricts the repair situation to method signature modifications and cannot repair the test cases that involve new method sequences.

Xu et al. proposed TestFix [7], that synthesises method sequence by genetic algorithm to repair JUnit test cases. The fitness function of TestFix consists of two components, one aims to the more coverage, and the other is how to make the assertion true. However, TestFix can find a sequence of method invocations to make broken test case pass. However, it cannot guarantee that the repaired test cases still keep their original test intent.

Li et al. present TRIP [9], which is an intent-aware automated repair technique of unit test case. TRIP replaces the non-compilable statements with new statements to generate repair candidates. TRIP models test intent with path conditions extracted by dynamic symbol execution and select the repair candidates with more similar test intent as a result. However, TRIP only evaluates test intent on a whole repair candidate. A large number of irrelevant candidates decrease the repair capability.

ITRACK [15], presented by Nguyen et al., is a tool that matches program entities (i.e., classes, methods) based on the similarity of their interactions, such as inheritance, invocation, and collaboration. ITRACK repairs test cases by replacing entities. Those mapping relations between program entities help developers manually fix test cases.

API migration techniques replace obsolete code with new code sequences with new APIs while preserving original program semantics. Balaban et al. [16] proposed a technique that automatically migrates a program to the new APIs according to the mapping between legacy and new APIs. Some technologies [17], [18] perform API migration by mining usage patterns from a large number of client applications.

A mass of researches focuses on program repair. Yuan et al. proposed ARJA [19], a new genetic programming based repair approach for automated repair of Java programs. Hua et al. introduced a program repair technique, SketchFix, which generates candidate program fixes on demand during the test execution [20].

Several pieces of research proposed techniques to repair GUI and web test cases. GUIAnalyzer [21] provides a general solution for GUI test case maintenance by using the set of heuristics. Flowrepairer [22] uses dynamic profiling, static analysis, and random testing to suggest a replacement UI action that repairs a broken workflow. ATOM [23] uses an event sequence model to abstract possible event sequences on a GUI and a delta ESM to abstract the changes made to the GUI. WATER [24] is to test the behavior of the test case on two successive versions of the web application to repair the broken Selenium test scripts for evolving web applications. WATERFALL [25] applies test repair techniques iteratively across a sequence of fine-grained versions of a web application. VISTA [26] collects various visual information for each test and analyzes this information through a fast image processing pipeline after the application is updated. By analyzing the information, it identifies the damaged test cases and attempts to find candidate repair methods that can fix them.

VIII. Conclusion

We presented a test repair approach, TRAD, that focuses on generating repair candidates with more similar test intent. TRAD relies on the post-dominance relationship in the broken test cases to determine the repair targets and generate repairs from the targets in a bottom-up way, so as to retain as much as possible the explicit test intent associated with the assertion. Furthermore, TRAD characterizes test intent with the source of the added methods, the similarity between them and the invalid operations and the repair cost, and raise intent-oriented priority criteria. With the priority criteria, TRAD gradually extends the partial DFGs and prioritizes the repair candidates. The empirical experiment was performed on 91 broken test cases in 4 open-source programs. The result shows that our approach generates 81 actual repairs, more than 72 of TRIP. Moreover, the two priority criteria proposed from reused elements can improve the repair capability.

In addition to performing more experiments, we plan to explore in the future works in several directions. First, because extracting and comparing many repair candidates is computationally expensive, we will attempt to select relative

methods for reducing the number of repair candidates and model test intent from the interactions between program entities. Second, we plan to generate repair candidates with meta-heuristic methods and guide the generation process by test intent similarity. Finally, classes and test cases in the same programs are not isolated. So we will explore the repair of broken test cases from the perspective of the whole test suite.

REFERENCES

[1] B. Daniel, V. Jagannath, D. Dig, and D. Marinov, "ReAssert: Suggesting repairs for broken unit tests," *ASE2009 - 24th IEEE/ACM International Conference on Automated Software Engineering*, pp. 433–444, 2009.

[2] A. M. Memon and M. L. Soffa, "Regression testing of GUIs," *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, vol. 28, no. 5, pp. 118–127, 2003.

[3] J. Imtiaz, S. Sherin, M. U. Khan, and M. Z. Iqbal, "A systematic literature review of test breakage prevention and repair techniques," *Information and Software Technology*, vol. 113, no. May, pp. 1–19, 2019. [Online]. Available: https://doi.org/10.1016/j.infsof.2019.05.001

[4] B. Daniel, D. Dig, T. Gvero, V. Jagannath, J. Jiaa, D. Mitchell, J. Nogiec, S. H. Tan, and D. Marinov, "ReAssert: A tool for repairing broken unit tests," in *Proceedings - International Conference on Software Engineering*, 2011, pp. 1010–1012.

[5] M. Mirzaaghaei, F. Pastore, and M. Pezzé, "Automatically repairing test cases for evolving method declarations," in *IEEE International Conference on Software Maintenance, ICSM*. IEEE, sep 2010, pp. 1–5. [Online]. Available: http://ieeexplore.ieee.org/document/5609549/

[6] M. Mirzaaghaei, F. Pastore, and M. Pezzè, "Supporting test suite evolution through test case adaptation," *Proceedings - IEEE 5th International Conference on Software Testing, Verification and Validation, ICST 2012*, vol. 2, pp. 231–240, 2012.

[7] Y. Xu, B. Huang, G. Wu, and M. Yuan, "Using genetic algorithms to repair JUnit test cases," *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, vol. 1, pp. 287–294, 2014.

[8] L. S. Pinto, S. Sinha, and A. Orso, "Understanding myths and realities of test-suite evolution," *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE 2012*, vol. 1, pp. 1–11, 2012.

[9] X. Li, M. D'Amorim, and A. Orso, "Intent-preserving test repair," *Proceedings - 2019 IEEE 12th International Conference on Software Testing, Verification and Validation, ICST 2019*, pp. 217–227, 2019.

[10] R. A. Assi, W. Masri, and F. Zaraket, "UCov: a user-defined coverage criterion for test case intent verification," *Software Testing Verification and Reliability*, vol. 26, no. 6, pp. 460–491, 2016.

[11] T. Xie, D. Marinov, and D. Notkin, "Rostra: A framework for detecting redundant object-oriented unit tests," *Proceedings - 19th International Conference on Automated Software Engineering, ASE 2004*, pp. 196–205, 2004.

[12] D. van Bruggen. JavaParser : Analyse, transform and generate your Java codebase. [Online]. Available: http://javaparser.org/

[13] W. Visser, C. S. Păsăreanu, and S. Khurshid, "Test input generation with Java PathFinder," in *ISSTA 2004 - Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2004, pp. 97–107.

[14] B. Daniel, T. Gvero, and D. Marinov, "On test repair using symbolic execution," *ISSTA'10 - Proceedings of the 2010 International Symposium on Software Testing and Analysis*, pp. 207–217, 2010.

[15] H. A. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. V. Nguyen, "Interaction-based tracking of program entities for test case evolution," *Proceedings - 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017*, pp. 433–443, 2017.

[16] I. Balaban, F. Tip, and R. Fuhrer, "Refactoring support for class library migration," *ACM SIGPLAN Notices*, vol. 40, no. 10, pp. 265–279, 2005.

[17] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang, "Mining API mapping for language migration," *Proceedings - International Conference on Software Engineering*, vol. 1, pp. 195–204, 2010.

[18] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Statistical learning approach for mining API usage mappings for code migration," in *ASE 2014 - Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, 2014, pp. 457–468.

[19] Y. Yuan and W. Banzhaf, "ARjA: Automated repair of Java programs via multi-objective genetic programming," *arXiv*, vol. 46, no. 10, pp. 1040–1067, 2017.

[20] J. Hua, M. Zhang, K. Wang, and S. Khurshid, "Towards practical program repair with on-demand candidate generation," pp. 12–23, 2018.

[21] S. McMaster and A. M. Memon, "An extensible heuristic-based framework for GUI test case maintenance," *IEEE International Conference on Software Testing, Verification, and Validation Workshops, ICSTW 2009*, pp. 251–254, 2009.

[22] S. Zhang, H. Lü, and M. D. Ernst, "Automatically repairing broken workflows for evolving GUI applications," in *2013 International Symposium on Software Testing and Analysis, ISSTA 2013 - Proceedings*, 2013, pp. 45–55.

[23] X. Li, N. Chang, Y. Wang, H. Huang, Y. Pei, L. Wang, and X. Li, "ATOM: Automatic Maintenance of GUI Test Scripts for Evolving Mobile Applications," *Proceedings - 10th IEEE International Conference on Software Testing, Verification and Validation, ICST 2017*, pp. 161–171, 2017.

[24] S. R. Choudhary, D. Zhao, H. Versee, and A. Orso, "WATER: Web application TEst repair," in *2011 International Workshop on End-to-End Test Script Engineering, ETSE 2011 - Proceedings*, 2011, pp. 24–29.

[25] M. Hammoudi, G. Rothermel, and A. Stocco, "WATERFALL: An incremental approach for repairing record-replay tests of web applications," in *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, vol. 13-18-Nove, 2016, pp. 751–762.

[26] A. Stocco, R. Yandrapally, and A. Mesbah, "Visual web test repair," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: ACM, oct 2018, pp. 503–514. [Online]. Available: https://dl.acm.org/doi/10.1145/3236024.3236063