Intent-Preserving Test Repair

Xiangyu Li
Georgia Institute of Technology
Atlanta, USA
xiangyu.li@cc.gatech.edu

Marcelo d'Amorim
Federal Univesity of Pernambuco
Recife, Brazil
damorim@cin.ufpe.br

Alessandro Orso
Georgia Institute of Technology
Atlanta, USA
orso@cc.gatech.edu

***Abstract*—Repairing broken tests in evolving software systems is an expensive and challenging task. One of the main challenges for test repair, in particular, is preserving the intent of the original tests in the repaired ones. To address this challenge, we propose a technique for test repair that models and considers the intent of a test when repairing it. Our technique first uses a search-based approach to generate repair candidates for the broken test. It then computes, for each candidate, its likelihood of preserving the original test intent. To do so, the technique characterizes such intent using the path conditions generated during a dynamic symbolic execution of the tests. Finally, the technique reports the best candidates to the developer as repair recommendations. We implemented and evaluated our technique on a benchmark of 91 broken tests in 4 open-source programs. Our results are promising, in that the technique was able to generate intent-preserving repair candidates for over 79% of those broken tests and rank the intent-preserving candidates as the first choice of repair recommendations for almost 70% of the broken tests.**

***Index Terms*—Software testing, test case repair, test intent characterization, search-based software engineering**

I. Introduction

As software evolves, regression testing is an essential, yet expensive activity. Much of the cost of regression testing comes from test-suite maintenance and, in particular, from repairing broken test cases (or test-case repair) [1]. Test cases can (and often do) break when the specification or the interface of a program change. When that happpens, broken test cases that are still relevant should ideally be repaired. Moreover, they should be repaired so that they exercise the same behavior as the original test, which can require considerable effort in terms of understanding the test cases and the functionality of the program being tested. In fact, anecdotal evidence suggests that developers often do not take the time to manually update the broken tests (e.g., if they believe these tests do not reveal faults in the program) [2].

To reduce the cost of test repair, researchers proposed several test repair techniques (e.g., [2]–[5]). One limitation of some of these existing techniques is that they focus on broken tests that result in runtime/assertion failures, where the fix consists of making the test expect an uncaught exception or modifying the failing assertion; they do not consider repairing broken tests caused by compilation errors, where the fix may involve complex changes in the interaction between the tests and the code. Another limitation of most existing test-repair techniques is that they follow (either explicitly or implicitly) the principle that the repair process should make minimal changes to a test, so as to limit the risk of changing the intent of the test. Although this minimal-change principle makes sense intuitively, it severely restricts the way in which tests can be repaired and may result in ineffective repairs. For example, a test case may be repaired by simply changing its assertion, which may work in some cases but could easily result in overfitting. Without other mechanisms to provide confidence that the intent of the original test is preserved, the repaired test may be useless (i.e., miss faults) or, even worse, wrong (i.e., fail when it should not). A study on test suite evolution of real-world software by Pinto and colleagues supports this argument [6]; they show that, for the subjects they used, nearly half of the test repairs were modifications that involved non-trivial changes to the test code, such as changes to method invocations and additions or deletions of statements. These earlier results indicate that handling non-trivial changes to the test code while preserving the intent of the tests is important for wider applicability of automated test repair.

In this spirit, and to address some of the limitations of existing test-repair techniques, we propose TRIP (Test Repair with Intent Preservation), an intent-aware automated unit test repair technique that can handle broken tests caused by compilation errors. TRIP takes as input a program P, its modified version P', and a test t for P that is broken by P'. Given these inputs, TRIP generates possible repair candidates for t, measures which candidates are most likely to be intent-preserving, and reports to the developer the top candidates as repair recommendations for t.

To generate test repair candidates, TRIP uses a search-based approach that removes the non-compilable statements in t and reuses the remaining test code. It then generates code to replace the removed statements incrementally, using the modified methods in P'. By doing so, TRIP can handle more general program changes, rather than being limited to method-signature changes, like some existing techniques (e.g., [3], [5]).

To model the intent of a test, TRIP uses an approach based on the path condition generated during dynamic symbolic execution of the test [7], [8]. Consider a possible repaired version of t, t', and let us indicate with $P(t)$ the execution of test t on P. TRIP determines whether t' is likely to exercise in P' the same behavior that t exercised in P by measuring the similarity between the path conditions generated when symbolically evaluating $P(t)$ and $P'(t')$. Intuitively, path conditions abstract the behavior of the program exercised by the test, and can thus be used as a proxy for test intent. This intuition is supported by the results of Banerjee and colleagues

[9]; they show that the path conditions generated for the a given test input on two programs with analogous functionality tend to be extremely similar, even when the two programs have substantially different implementations.

We evaluated TRIP on a benchmark consisting of 4 open-source programs and 91 broken test cases. Specifically, we studied whether TRIP can (1) generate intent-preserving repair candidates for broken tests and (2) rank the actual repairs high in the list of repair recommendations. Our results are promising, in that TRIP generated intent-preserving repair candidates for 72 of the 91 broken tests (79%), including in cases where the repair involved dealing with significant program changes. As for the second part of the study, TRIP ranked the intent-preserving candidates as the first repair recommendation in 63 of the 72 cases considered (87.5%), and among the top-3 recommendations in 68 of the 72 cases (94.4%). Moreover, in a comparison involving over 1,000 tests, our intent-similarity metric correctly identified corresponding tests in different versions of the program in over 95% of the cases considered. Overall, these results provides initial, yet strong evidence that our approach is effective and that path conditions can indeed be used to characterize and compare test intent.

The main contributions of this paper are:

- A new approach for characterizing/comparing test intent.
- A technique, TRIP, that generates and ranks intent-preserving repair candidates based on our approach for intent characterization.
- A tool that implements TRIP and that is publicly available, together with our experiment data and infrastructure (https://sites.google.com/view/trip-test-repair).
- An evaluation of TRIP's effectiveness and usefulness.

II. APPROACH

Figure 1 provides an overview of TRIP and shows its three main components: the Repair Candidate Generator, the Test Intent Extractor, and the Test Intent Comparator. Edge numbers represent the order in which the information flows through the system. To illustrate the approach, and throughout the paper, we use P and P' to refer to the old and new versions of a program. For each broken test t, TRIP takes as input (1) t, (2) P (on which t compiles and passes), and (3) P' (on which t fails to compile); given these inputs, TRIP produces as output a list of repaired test candidates that compile and pass on P', ordered by their likelihood of preserving the intent of t.

We now illustrate TRIP using the example of broken test t in Figure 2a, which is adapted from a real case we observed in our evaluation. In P', field `Entities.map` was made private and thus became inaccessible to (and broke) the test. To repair t, the *Repair Candidate Generator* produces a set of repair (test) candidates that compile and pass on P'; To do so, it first compares P and P' in terms of their public methods and fields. It then (1) generates variants of the broken test code, by removing method calls and field accesses that are no longer valid and replacing them with public elements in P', (2) runs the generated variants against P', and (3) outputs the passing

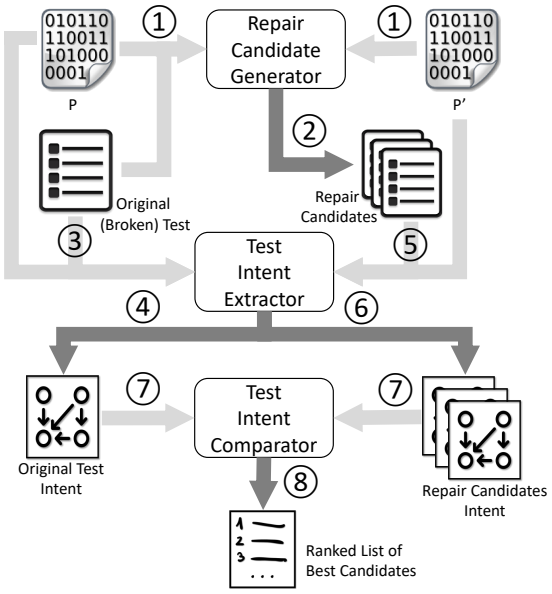

Figure 1: Overview of the approach.

ones as repair candidates. Due to space limitation, we only show two of the generated candidates, in Figures 2b and 2c. The *Test Intent Extractor* executes t on P using dynamic symbolic execution [7], [8] and uses the resulting path condition as the intent of the original test. It then computes the intents of the repair candidates in the same way, by executing them on P'. Finally, the *Test Intent Comparator* computes the similarity between the test intents of each repair candidate and t, ranks the candidates according to the computed similarity score, and reports the ranked list of candidates to the developer. Although the path conditions generated for the example are too complex to be shown, the relevant bit of information is that both t and the repair candidate in Figure 2c contain clauses involving `Entities.PrimitiveEntityMap` in their path conditions, while the candidate in Figure 2b does not. Therefore,TRIP ranks the repair candidate in Figure 2c above the other one and reports it as the first repair recommendation.

We now discuss the different parts of TRIP in detail. Please note that we describe our technique assuming that the program under test (and the tests) are written using an object oriented language. However, we believe that our technique can be generalized to other types of languages.

A. Repair Candidate Generator

This module searches for repair candidates of a broken test in several steps, which we describe individually.

1) Comparing P and P': Test code exercises program functionality by invoking methods and accessing fields in the program. Therefore, TRIP models P and P' in terms of their *public program elements* (*PPEs*). A PPE is basically an element (i.e., method or field in Java) that a test case can access in the program. TRIP identifies the PPEs in P and P', compares them, and categorizes them into three partitions: removed PPEs (PPEs in P but not in P'), new PPEs (PPEs in P' but not in P) and unchanged PPEs (PPEs in both P

```
1 public void testHtml40Nbsp() {
2   Entities e = new Entities();
3   e.map = new Entities.PrimitiveEntityMap();
4   Entities.fillWithHtml40Entities(e);
5   assertEquals(" ", e.escape("\u00A0")); }
```

(a) Broken test eample.

```
1 public void testHtml40Nbsp() {
2   Entities e = new Entities());
3   Entities.fillWithHtml40Entities(e);
4   assertEquals(" ", e.escape("\u00A0")); }
```

(b) Non-intent-preserving repair candidate.

```
1 public void testHtml40Nbsp() {
2   Entities e = new Entities(
3     new Entities.PrimitiveEntityMap());
4   Entities.fillWithHtml40Entities(e);
5   assertEquals(" ", e.escape("\u00A0")); }
```

(c) Intent-preserving repair candidate.

Figure 2: Test repair example.

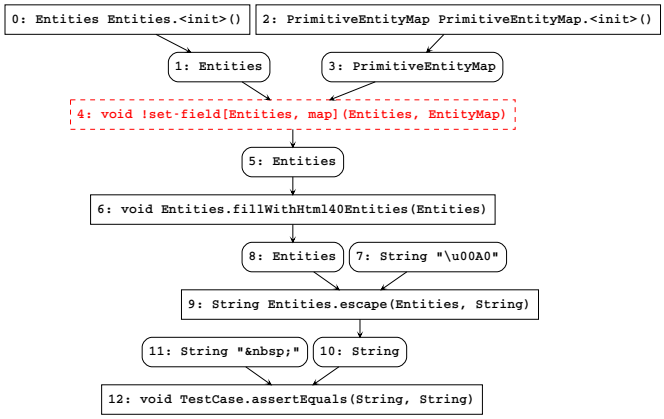

Figure 3: Data-flow graph for the broken test example.

and P'). Intuitively, (1) one or more of the removed PPEs is the reason why t cannot compile in P', and (2) new and unchanged PPEs are the elements that can be used to fix t.

2) Analyzing the Broken Test: This component analyzes the code of t and constructs an intermediate representation of t suitable for searching repair candidates.

To represent method invocations, field accesses, and built-in language operators in a uniform way, TRIP suitably transforms read and write field accesses into getters and setters, respectively, and then represents all three entities as *operations*. An *operation* is defined as a 4-tuple $\langle d, n, T, r\rangle$, where d is the declaring type, n is the operation name, T is an ordered list of parameter types, and r is the return type. This definition has a straightforward mapping to static method declarations. For an instance method, T also contains the declaring type as the first element. A constructor is handled as a static method with a special n and where r corresponds to the declaring type. For the getter of a static field, T is empty, and r corresponds to the type of the field. For the setter of a static field, conversely, T consists of the type of the field, and r is null. Similar to instance methods, T for the getter (or setter) of an instance field contains the declaring type as the first element. Built-in operators can be converted in a manner similar to static methods, with the difference that their declaring types are null.

TRIP constructs from the code of the broken test a *static data flow graph* (DFG)—a bipartite graph represented as a 3-tuple $\langle D, O, E\rangle$, where D is a set of nodes representing definitions, O is a set of nodes representing operations, and $E \in (D \times O) \cup (O \times D)$ is a set of directed edges representing the input-output relation between definitions and operations. A definition node is labeled with its type and represents either a literal constant or a definition generated by an operation.

Consider an operation with parameter types $T = (pt_1, pt_2, \ldots, pt_n)$ and return type $r = rt$. In addition to creating a definition node of type rt, the operation also generates a definition node of type pt_i for each pt_i that is not a primitive type; this accounts for the fact that the input objects could be modified as a side effect of the operation. TRIP performs an intra-procedural data flow analysis to identify the input definitions for each operation, treating objects as whole entities (i.e., without analyzing their internal data flow).

Figure 3 shows the DFG for the example of broken test in Figure 2a. In the figure, rectangles represent operation nodes, whereas rounded rectangles represent definition nodes. For ease of readability, in the figure we label the operation nodes using a Java-like syntax, instead of an actual 4-tuple. The mapping between the two should be straightforward. Also, for ease of reference, we prepend an ID to the node labels. Node 0 corresponds to the call to the constructor of class `Entities` on line 2 of the code. The operation takes no parameter and returns an object of type `Entities`, thus generating the output definition represented by node 1. Similarly, node 2 represents the call to the constructor of class `PrimitiveEntityMap` on line 3, and node 3 represents its resulting output definition. Node 4 represents the field assignment on line 3. The special name `!set-field[Entities, map]` indicates that this is a setter generated for field `map` in class `Entities`. This operation node has definition nodes 1 and 3 as inputs, and returns no result. It nevertheless generates definition node 5 because the input definition node 1 is an object that might have been modified by the operation as a side effect. (In fact, the operation also produces a side-effect output definition node of type `PrimitiveEntityMap`, which is never referenced and was removed to simplify the graph.) Nodes 6 through 12 encode the data flow involving lines 4 and 5 in a similar way.

TRIP searches for compilable variants of the broken test by modifying the DFG. Compared to directly changing the source code, using the DFG representation offers several benefits. In particular, the availability of data-flow dependency information allows TRIP to identify relevant parts of the code that are affected by the non-compilable statements. Also, by simplifying the handling of variables and references, the higher level representation of data in the DFG allows TRIP to avoid generating many semantically equivalent variants that are different only in variable assignments. Finally, the DFG models the side effects of operations explicitly, enabling TRIP to generate variants that cover different combinations of data flow caused by side effects. This is essential for test repair, where the basic building blocks are operations and their side effects cannot be ignored.

```
1: procedure SEARCHVARIANTS(dfg, nPPEs, uPPEs)
2:     variants ← {}
3:     partialDfgs ← {onlyWellFormedDefs(dfg)}
4:     operations ← genOperations(nPPEs, uPPEs)
5:     while time limit not reached do
6:         g ← nextPartialDfg(partialDfgs)
7:         variants.addAll(fixDfg(dfg, g))
8:         for o ∈ operations do
9:             extended ← extendPartialDfg(g, o)
10:            partialDfgs.addAll(extended)
11:        end for
12:    end while
13: end procedure
```

Figure 4: Algorithm for computing variants of a broken test.

TRIP analyzes the DFG to identify the set of invalid operation nodes that directly cause compilation errors and thus must be removed, the set of definition nodes that are affected by the invalid operations, and the set of definition nodes that can be reused in the repair candidates. An operation node is marked as *invalid* if it involves a removed PPE. The set of *affected definitions* contains the definition nodes that are forward reachable from an invalid operation node in the DFG—their data-flow predecessors could have been changed as a result of removing the invalid operation nodes, potentially altering the behavior of the test. The re-usable definition nodes are those that are still well-formed after the invalid operations are removed; a definition node d is *well-formed* if either it represents a literal constant or the operation node that generates d (1) is valid and (2) each of its inputs in the DFG has at least one well-formed reaching definition. It is worth noting that the sets of affected definitions and well-formed definitions are not mutually exclusive, as input arguments of operations can have multiple reaching definitions.

In our example, field `Entities.map` becomes inaccessible to the test code in P'. The operation corresponding to the setter for that field is thus in the set of removed PPEs. Node 4 would therefore be marked as invalid (indicated by its dashed border). The affected definition nodes include nodes 5, 8, and 10. Definition nodes 1, 3, 7, and 11 are well-formed and can be reused. We now discuss how TRIP uses this information to generate repair candidates for a broken test.

3) Generating the Repair Candidates: To generate repair candidates, TRIP first generates compilable variants of the broken test t using a search-based algorithm. It then runs the variants against P', reporting the passing ones as possible repair candidates.

The variant generation algorithm, SEARCHVARIANTS, is shown in Figure 4. The algorithm takes as input the DFG for t and the sets of new and unchanged PPEs (see Section II-A1), and produces as output a set of compilable variants.

We first introduce a set of functions that are essential for explaining the SEARCHVARIANTS algorithm. Function `onlyWellFormedDefs` (line 3) creates a *partial DFG*, which is a subgraph of the original DFG containing only well-formed definition nodes and corresponding operation nodes. TRIP generates variants of t based on this partial DFG, by re-using the well-formed definitions in the original test. Function `genOperations` (line 4) takes as input the sets of new and unchanged PPEs and generates a list of corresponding operations, annotated to distinguish between PPEs in the two sets. Function `extendPartialDfg` (line 9) takes as input a partial DFG and an available operation and extend the partial DFG using that operation. TRIP adds operation nodes by matching the types of their parameters with the types of the existing definition nodes in the partial DFG. For a given partial DFG g, and an operation that takes parameter types $(pt_1, pt_2, \ldots, pt_n)$, TRIP computes the set of lists of definition nodes $\{(d_1, \ldots, d_n) \mid d_i \in g \wedge \texttt{type}(d_i)\ \texttt{is assignable to}\ pt_i, 1 \leq i \leq n\}$, where each list represents a complete assignment of existing well-formed definitions to the parameters of the operation. For each assignment, the algorithm creates a new partial DFG that contains the elements of `g`, a new operation node created from the parameter assignment, and new output definition nodes for the added operation node. TRIP builds increasingly more complex partial DFGs by building on existing ones. Function `fixDfg` (line 7) attempts to restore the data flow of a broken DFG using elements of a partial DFG. Given a partial DFG g, it finds sets of definition node pairs $\{(d_1, d_2) \mid d_1 \in \texttt{affectedDefs(dfg)} \wedge d_2 \in g \wedge \texttt{type}(d_2)\ \texttt{is assignable to type}\ (d_1)\}$, where each pair (d_1, d_2) indicates the replacement of d_1 with d_2, and `affectedDefs(dfg)` represents the affected definitions in the broken DFG. (To replace a definition node d_1 with a definition node d_2, TRIP modifies the operation nodes that have d_1 as input arguments to have d_2 instead.) Each of these sets of definition pairs represents a way to replace a subset of the affected definitions with the well-formed definitions of compatible types in the given partial DFG, resulting in a new DFG g'. After removing the unreachable nodes from g', if all the remaining nodes are well-formed, TRIP generates code for the repaired DFG and retains the generated test code as a valid variant. Finally, the function returns to the main algorithm the set of valid variants generated from the broken DFG and the specified partial DFG.

Now that we have introduced the set of functions used by the SEARCHVARIANTS algorithm, we can describe how the high-level algorithm works. SEARCHVARIANTS first initializes set `variants`, which stores the set of compilable test variants, to an empty set. It then sets `partialDfgs`, which keeps track of the partial DFGs, to contain only the one constructed from the original DFG. On line 6, function `nextPartialDfg` selects a partial DFG `g` that has not been used before. Line 7 calls function `fixDfg` to restore the data flow of the broken DFG using the elements of the selected partial DFG g and stores the resulting variants. Since TRIP only modifies the data flow, the resulting code of the variants always has the same control flow as the original test. At lines 8–11, the algorithm extends g using the available operations and stores the extended partial DFGs, discarding possible duplicates. The algorithm then proceeds to the next iteration and repeats these steps until a time limit is reached.

It is important to note that function `nextPartialDfg` controls the order in which the variant generation algorithm

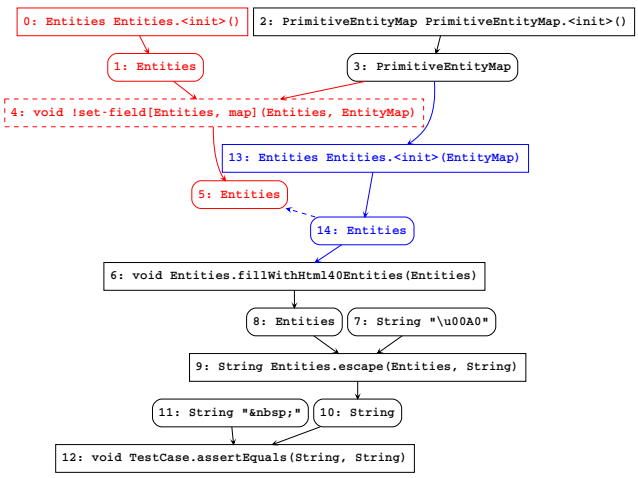

Figure 5: Repaired data-flow graph.

explores the search space. Specifically, it prioritizes the partial DFGs (1) that have fewer operation nodes and (2) whose operation nodes are more likely to be related to the semantics of the original test. To do so, TRIP assigns a *cost* to each available operation and computes the cost of a partial DFG as the total cost of its operation nodes, excluding those inherited from the original broken DFG. It then selects first partial DFGs with lower total cost. TRIP also uses several heuristics to reduce the cost of the operations that are more likely to be needed for the fix. In particular, because new PPEs are more likely to be useful than unchanged ones, TRIP initializes the cost of their corresponding operations to 1, while the cost of the other operations is set to 2. TRIP also (1) computes the textual similarity between each available operation and the removed operations in the broken DFG and (2) reduces the cost of the operations with higher similarity scores—operations more similar to the removed ones are, intuitively, more likely to correspond to replacements for such operations.

Figure 5 shows the repaired DFG for the example of broken test in Figure 2. The variant search algorithm created the new nodes 13 and 14 and the corresponding data-flow edges. Specifically, TRIP creates node 13 using an operation that corresponds to the new PPE `Entities.<init>(EntityMap)`. As its input, which must be of type `EntityMap`, the new node takes definition node 3, which is reused from the DFG of the broken test and has compatible type `PrimitiveEntityMap`. As output, node 13 generates the new definition node 14, of type `Entities`. The dotted arrow from node 14 to node 5 is not a DFG edge, and we use it to indicates that the former is used to replace the latter. As a result of the replacement, the DFG edge from node 5 to node 6 is removed and replaced by the edge from node 14 to node 6. Nodes 0, 1, 4, and 5 are no longer reachable and are therefore removed from the DFG. TRIP verifies that the resulting graph is a compilable variant by checking that all nodes are well-formed and, if so, generates code for it. This variant is among the first a few generated in the search process for two reasons. First, it requires only one new operation, and thus its total cost is low. Second, the operation used in this variant (1) is new and (2) has a relatively high textual similarity with the removed operation since they are both constructors and share the same declaring type.

In parallel with variants generation, TRIP executes the code corresponding to the generated variants (`variants`) against P' and reports the passing variants as repair candidates. Because algorithm SEARCHVARIANTS is intentionally designed to be flexible in the modifications it makes to the broken tests, the repair candidates it generates could potentially exercise behaviors completely unrelated to the original test. For this reason, TRIP further analyzes the repair candidates to identify the ones that are likely to preserve the original test intent.

B. Test Intent Extractor

For a given test, the Test Intent Extractor performs a form of dynamic symbolic execution [7], [8]: it marks some test input values as symbolic and runs the test symbolically, exploring only the path followed by the concrete execution. This produces a *path condition* (*PC*) expressed in terms of the initial symbolic values. TRIP uses this PC to characterize the intent of the test; intuitively, the PC encodes the runtime behavior of the execution expressed in terms of the test inputs, thus providing an abstraction of the test semantics.

TRIP marks as symbolic two types of test input values: (1) primitive values in the test code and (2) objects created in the test code. It is worth noting that by *input values* we mean, intuitively, values that flow into the code under test. Also intuitively, by *test code* we mean the code in a test case and in the test initialization code (e.g., a `@Before` method in a JUnit test case). Making primitive values symbolic is straightforward. For objects in general, TRIP marks as symbolic the reference to the object, so as to capture operations such as comparisons to "`null`" or checks on the type of the object. For string literal objects, which we consider a special kind of primitive values, TRIP additionally marks the length of the string and each individual character as symbolic, so as to be able to capture more detailed information in the PC.

A special case for TRIP is that of test cases that do not contain literal input values, as the PCs for these tests (which are expressed in terms of the symbolic test inputs) contain little information. In these cases, TRIP also marks the primitive values in the code under test as symbolic. Note that TRIP does not marks these values as symbolic by default, as we expect the information carried by the clauses that involve test inputs, when present, to overshadow the information in the clauses that involve only values in the program code.

While executing the test symbolically, TRIP maintains the association between the concrete values in the program state and their symbolic values, expressed in terms of the inputs. It also encodes the branches followed in the execution by suitably adding to the PC clauses expressed in terms of the symbolic values involved in the corresponding predicate (if any). At the end of the execution of the test, the Test Intent Extractor stores the resulting PC.

Figure 6 shows an example that illustrates how TRIP generates PCs for test cases. (We used this example as opposed to that from Figure 2a because it produces a much shorter and

```
1  public int sumAbs(int x, int y) {
2    if (x < 0)
3      x = -x;
4    if (y < 0)
5      y = -y;
6    return x + y;
7  }
8
9  @Test
10 public void testSumAbs() {
11   assertEquals(5, sumAbs(-2, 3));
12 }
```

Figure 6: Path condition example.

readable PC.) The function under test is `sumAbs`, which takes as parameters two integers, `x` and `y`, and returns the sum of their absolute values. The test method `testSumAbs` calls `sumAbs` and checks whether the return value matches the expected result.

In this example, TRIP would mark as symbolic the two primitive values -2 and 3 on line 11 and then execute the test. Let us refer to the symbolic values for -2 and 3 as sym_1 and sym_2. On line 2, $(-2 < 0)$ is true, so the clause $(sym_1 < 0)$ is added to the initially empty PC. On line 3, the value of x is negated, and therefore its value becomes 2 (concrete) and $-sym_1$ (symbolic). On line 4, $(3 < 0)$ is false, so the clause $\neg(sym_2 < 0)$ is conjoined to the current PC. On line 6, the sum of x and y is returned, which corresponds to 5 (concrete) and $-sym_1 + sym_2$ (symbolic). Because the expected value (5) is successfully compared to this return value, the clause $(-sym_1 + sym_2 = 5)$ is also conjoined to the current PC. The PC for the test at the end of its execution is therefore $(sym_1 < 0) \wedge \neg(sym_2 < 0) \wedge (-sym_1 + sym_2 = 5)$.

TRIP generates PCs in this way for the original test t by running it on P, and for the repair candidates by running them on P'. It then compares them as described in the next section.

C. Test Intent Comparator

The Test Intent Comparator computes the *intent similarity* of two tests by measuring the structural similarity between the PCs generated for the tests by the Test Intent Extractor. Although an approach based on logical reasoning for finding relationships between pairs of PCs (or parts thereof) may provide more accurate results, we believe that our approach provides a good cost-accuracy tradeoff.

TRIP represents each PC as a forest (*PC forest*). To do so, TRIP first generates a directed acyclic graph (DAG) for each clause in the PC, where the leaves in the graph represent symbolic inputs or constant values, and the internal nodes represent operators. It then transforms the resulting DAG into a tree (*clause tree*) by duplicating all the subgraphs that have multiple predecessors and assigning to each copy a distinct predecessor. The set of clause trees for all the clauses in a PC is the PC forest for that PC.

To compute the similarity between two clauses ct_1 and ct_2, TRIP runs a tree alignment algorithm [10] on the clause trees for ct_1 and ct_2, which identifies matching (i.e., identical) nodes between the two trees. It then calculates the *clause similarity score* for ct_1 and ct_2 as $\sqrt{m^2/(ct_1.size \times ct_2.size)}$ where m is the number of matched pairs of tree nodes in the maximum alignment, and $ct_i.size$ is the total number of nodes in ct_i. Intuitively, the clause similarity score represents the (geometric mean of the) proportion of matched nodes in the two trees.

To compute the similarity between two PCs pc_1 and pc_2 (computed for tests t_1 and t_2) TRIP finds a matching between the two sets of clauses in the PCs that maximizes the sum of the corresponding clause similarity scores. If we indicate with $maxsum$ this maximum sum, TRIP then computes the *intent similarity score* for t_1 and t_2 as $\sqrt{maxsum^2/(pc_1.size \times pc_2.size)}$, where $pc_i.size$ represents the number of clauses pc_i.

Because tree alignment and maximum matching of unordered sets are expensive to compute, the Test Intent Comparator uses two optimizations to improve its performance. *First*, it finds the pairs of identical clauses in the two PCs and excludes them from the tree alignment computation. We observed that, for test cases with similar intents, a significant proportion of clauses can be matched in this fast way. (This step can be completed in linear time by using hashing.) *Second*, instead of applying the tree alignment algorithm on every pair of remaining clauses, TRIP first computes a (much faster) approximation of clauses similarity. To do so, TRIP leverages an approach by Yang and colleagues [11] that calculates an upper bound of similarity by transforming the clause trees into fixed-dimensional vectors that partially encode the structural features of the trees; the pairs of clauses that are too dissimilar in the vector space are simply assigned a similarity score of 0 and excluded from further consideration.

Using this approach, the Test Intent Comparator computes, for each repair candidate, its intent similarity with respect to the original (broken) test t. It then sorts the repair candidates in descending order of intent similarity scores. In the case of ties, TRIP prefers candidates with a fewer number of new operations, based on the intuition that candidates more similar to t are also more likely to preserve its intent. Finally, TRIP presents the ranked list of repair candidates to the developer as repair recommendations.

III. EMPIRICAL EVALUATION

We evaluated TRIP on 91 broken tests in 4 open-source programs, by investigating the following research questions:

RQ1: How effective is TRIP at generating actual test repairs?

RQ2: Are path conditions a good abstraction of test intent?

We next describe our experiment setup and findings.

A. Experiment Setup

1) Implementation: We implemented TRIP for the Java programming language. To build DFGs, we leveraged JavaParser [12], which let us generate ASTs with visitor support. We used a modified version of Java Pathfinder [13] as a dynamic symbolic execution engine to compute path conditions. Although TRIP was implemented for Java code, the technique is general and could be implemented for other languages.

2) Benchmark: As a benchmark, we used four open-source programs used by some of the authors in previous research on test suite evolution [6]. Table I reports, for each program considered, the number of versions considered, the number of

Table I: Benchmark programs used in our evaluation.

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

Table II: Classification of failing test cases.

Program	*# RF*	*# RE*	*# TR*	*# EX*	*# BT*	**Total**
commons-lang	1	28	28	0	14	71
commons-math	44	25	0	0	3	72
gson	234	40	79	10	69	432
joda-time	0	17	0	3	5	25
Total	279	110	107	13	91	600

classes, the number of lines of code, and the number of tests. Because we considered multiple versions of each program, the values are reported as numeric ranges.

For each program in our benchmark, we considered each pair of consecutive versions, P and P', and their corresponding (JUnit) test suites, T and T'. We first identified the tests in T that pass on P but fail on P' and manually inspected them to determine the reasons why they fail. We identified five main reasons and classified the test cases accordingly, as shown in Table II. Some tests (*Column # RF*) fail because they test functionality that was removed in P'; these tests are obsolete, so we removed them from further consideration. Other tests (*Column # RE*) fail due to runtime errors, such as assertion failures and uncaught exceptions; these tests can be repaired using existing techniques (e.g., [2], [4]) that are complementary to our approach. The remaining tests have compilation errors on P', so they are the kinds of tests that our approach targets. Some of these tests, however, can be trivially repaired, so considering them would unfairly inflate the success rate of TRIP. In particular, some tests (*Column # TR*) could be fixed through trivial refactoring, such as method renaming. Similarly, tests that fail because of changes in exception declarations (*Column # EX*) could be fixed by changing the type of exceptions caught in the corresponding catch blocks. We expect that developers would evolve these tests together with their code (e.g., as part of a refactoring), so we focused on the remaining tests (*Column # BT*)—tests that are broken due to compilation errors and cannot be trivially fixed. As Table II shows, the number of tests in this category is comparable to that of tests that are broken due to runtime errors (91 vs 110) and corresponds to 28% (91 / (600-279)) of the non-obsolete broken tests.

3) Data Collection: To collect the data needed to answer our research questions, for each broken test t considered we performed the following steps. *First*, we searched for a test t' in T' with the same name. If present, we considered t' the repaired version of t (i.e., our ground truth). Note that we did not try to find repaired test cases that are also renamed in T', as manually determining semantic equivalence between arbitrary tests is expensive and may introduce bias. Moreover, we expect this occurrence to be rare, and having slightly fewer data points does not affect the validity of our results. *Second*, we determined whether TRIP was able to generate an actual repair for t by checking, for each repair candidate, whether it was semantically equivalent to t'. It was always straightforward to determine semantic equivalence between a repair candidate and t', as (for the successful checks) the candidate was either identical to t' or could be made so through trivial program transformations. The only exceptions were a few cases in which developers also extended the functionality of the test in addition to fixing it. *Finally*, for all the cases in which TRIP was able to generate an actual repair for t, we determined the rank of that repair in the list of intent-preserving candidates.

Table III: Empirical evaluation results.

Program	*New PPEs*	*BT*	*CG*	*CG-noTS*	*Rank1*	*Rank3*
commons-lang	302	14	14	14	14	14
commons-math	2367	3	1	1	1	1
gson	451	69	54	35	45	50
joda-time	66	5	3	3	3	3
Total	-	91	72	53	63	68

Table III shows a summary of our results. Column *New PPEs* reports the average number of new PPEs (public program elements) in each pair of versions (P, P') considered. Because our technique uses a search-based approach, repair candidate generation is more expensive if the number of PPEs in P' but not in P is higher. (Note that, since it does not make sense to compute the cumulative number of PPEs across programs, we omit the total for this column.) For convenience, Column *BT* repeats the total number of broken tests we already reported in the corresponding column of Table II. Columns *CG* and *CG-noTS* report, respectively, the number of broken tests for which TRIP successfully generated intent-preserving repair candidates with and without the use of textual similarity between PPEs to bias the search (see II-A3). Columns *Rank1* and *Rank3* report the number of broken tests for which TRIP ranked the correct repair candidate as the first repair recommendation and among the top three repair recommendations, respectively.

B. RQ1: How Effective Is TRIP At Generating Actual Repairs?

The results in Table III show that TRIP was able to generate an actual repair for a broken test in almost 80% (72 out of 91) of the cases we considered. The results also show that the use of textual similarity between PPEs to bias the search is effective, at least in the case of gson, for which the number of successful repairs generated goes from 54 to 35 without textual similarity. This is because TRIP generates more (invalid) repair candidates when the search is not biased, so it may timeout before generating the actual repair.

It is worth noting that we planned to compare the effectiveness of TRIP with that of TestCareAssistant (TCA) [5], which repairs non-compilable broken test cases caused by method signature changes. However, because TCA is an old project, the tool was not kept up to date, and the benchmarks used to evaluate it are no longer available. Therefore, despite the generous help offered by the authors of TCA, we could neither run their tool on our benchmark nor evaluate our technique

on their benchmark. We nevertheless provide a qualitative comparison of TRIP and TCA in Section IV.

C. RQ2: Are PCs a Good Abstraction of Test Intent?

As column *Rank1* in Table III shows, among the 72 broken tests for which TRIP generated an actual repair, such repair was ranked first in 63 cases, that is, 88% of the cases. This means that, in a fully automated test-repair scenario, TRIP could have repaired those broken tests without developer intervention. If we consider the cases for which TRIP ranked an actual repair among the top three repair candidates, the number goes from 63 to 68 (column *Rank3* in Table III), that is, over 94% of the cases. For the remaining four cases, the rank assigned by TRIP to the actual repair ranges between 13 and 17. A manual inspection of these cases revealed that, in all four cases, the original tests target methods that simply return a constant, which causes the PCs to contain no information.

We believe that these results strongly indicate that (1) PCs are a good way of abstracting test intent, and (2) our technique for comparing test intents expressed as PCs is effective.

To investigate RQ2 in more depth, we conducted an additional experiment, aimed to further assess whether tests that have the same intent would also generate similar PCs, even when they exercise different code. For the experiment, we considered all the test cases, rather than only the failing/broken ones, for one of our benchmark programs: commons-lang. (We considered only one program due to the computational cost of performing the experiment.) Specifically, for each pair of consecutive versions (P, P') of commons-lang and corresponding test suites T and T', we performed the following steps: *First*, we computed the set $SIT_{(P,P')}$ of pairs of tests (t, t') such that (1) $t \in T$, (2) $t' \in T'$, (3) t and t' have the same name, and (4) t and t' cover different statements in P and P', respectively. Again, we assumed that tests with the same name have the same intent, so we used $SIT_{(P,P')}$ as our ground truth. *Second*, for each test t such that (t, $*$) in $SIT_{(P,P')}$, we identified the set $MAXS_t$ of tests in T' that have the highest intent similarity with t (where $\|MAXS_t\| > 1$ when more than one test in T' have the same highest similarity score with t). *Finally*, we classified the resulting $MAXS_t$ into one of three categories:

Unique match: $\|MAXS_t\| = 1$, and the single t' in $MAXS_t$ is the test with the same intent as t (i.e., $(t, t') \in SIT_{(P,P')}$).

Non-unique match: $\|MAXS_t\| > 1$, and one of the t' in $MAXS_t$ is the test with the same intent as t (i.e., $\exists\ t' \in MAXS_t\ such\ that\ (t,t') \in SIT_{(P,P')}$).

No match: $MAXS_t$ does not contain the test with the same intent as t (i.e., $\nexists\ t' \in MAXS_t\ such\ that\ (t,t') \in SIT_{(P,P')}$).

Figure 7 shows the results of this experiment for all the tests t in all (t, $*$) pairs in all $SIT_{(P,P')}$ sets (1,070 overall). As the figure shows, in 1,024 (95.7%) of the 1,070 cases we found unique matches—the pairs of tests in T and T' with the same intents also had the highest intent similarity score (across over 1,000 other tests). In 33 cases (3.1%), we found multiple matches—the pairs of tests with the same intents also had the highest intent similarity score, but there were also other

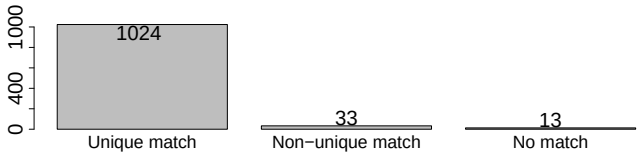

Figure 7: Number of test cases for which the test with the same intent in the next consecutive version is uniquely matched, non-uniquely matched, and not matched.

tests with the same intent similarity score. Finally, in 13 cases (1.2%), we found no matches—the pairs of tests with the same intents did not have the highest intent similarity score.

We believe that this second set of results provides additional, even stronger evidence that PCs and PC similarity are good abstractions for test intent and test intent similarity.

D. Example Repairs

To provide further information on our empirical evaluation, in this section we present the details of two test cases correctly repaired by TRIP (i.e., cases in which the repaired tests generated (and ranked first) by TRIP are semantically equivalent to the actual repairs performed by the developers).

Figure 8 shows the first example: a broken test t_1 in gson version 1.3 (a) and the repaired test t'_1 generated by TRIP (b). Test t_1 is broken because method `translateName(NamePolicy, Field)`, which t_1 calls on line 6, was changed in gson 1.5 to take a parameter of type `FieldAttributes` instead of `Field`. For readability, we slightly simplified the code in t_1 and t'_1. Similarly, to simplify the discussion, in the rest of this section we refer to objects using the names of the references pointing to them.

As shown in the figure, to repair t_1, TRIP removes one instruction (line 6 in t_1, marked with a dashed underline) and adds two instructions (lines 4–5 and 8 in t'_1, marked with a solid underline). This corresponds to removing one operation node from and adding two operation nodes to the DFG for t_1. The first new instruction (re)uses existing objects `SomeObject.class` and `f` in t_1 to invoke constructor `FieldAttributes(Class, Field)` and create a `FieldAttributes` object `var0`. The second new instruction uses existing object `policy` and the newly created object `var0` to call method `translateName(NamePolicy, FieldAttribute)`, which replaces the non-compilable method call in the broken test. TRIP recommends this intent-preserving repair candidate as the first choice because t'_1 uses object `Field` in the same way as t_1 does, resulting in a similar PC.

In this case, TRIP can find the correct repair candidate for t_1 even without using the textual similarity between PPEs to bias the search. This happens for two reasons: (1) the operations used for the repairs are new PPEs, which have lower costs; and (2) the objects in t_1 that can be reused in t'_1 provide sufficient type information to limit the number of compilable variants generated by TRIP (see Section II-A3).

Figure 9 shows the second example: a broken test t_2 in gson version 2.0 (a) and the semantically equivalent repaired test t'_2 (b). In this case, the test is broken because of a complex change

```
1  public void testFieldWithAnnotation() {
2   String fieldName = "fieldWithAnnotation";
3   Field f = SomeObject.class.getField(fieldName);
4   NamePolicy policy = new NamePolicy();
5   assertEquals("annotatedFieldName",
6     policy.translateName(f)); }
```

(a) Broken test t_1.

```
1  public void testFieldWithAnnotation() {
2   String fieldName = "fieldWithAnnotation";
3   Field f = SomeObject.class.getField(fieldName);
4   FieldAttributes var0 =
5     new FieldAttributes(SomeObject.class, f);
6   NamePolicy policy = new NamePolicy();
7   assertEquals("annotatedFieldName",
8     policy.translateName(var0)); }
```

(b) Intent-preserving repair candidate t'_1.

Figure 8: Example of test case repair #1.

in the software design. Class `InnerClassExclusionStrategy` (referenced on lines 3–4 in t_2) was removed in gson 2.1, and its functionality was moved to class `Excluder` and provided through a set of significantly different PPEs. In particular, method `shouldSkipField(FieldAttributes)` (invoked on line 7) is not present in the new class.

To repair t_2, TRIP removes three instructions (lines 3–4, 5–6, and 7 in t_2, marked with a dashed underline) and adds three instructions (lines 3–5 in t'_2, marked with a solid underline). TRIP ranks this repair candidate first despite the significant changes between t_2 and t'_2.

In this case, without considering textual similarity of PPEs, TRIP is able to generate a candidate that contains lines 3 and 5 of the valid repair. However, it fails to generate a candidate that also contains line 4 because class `Excluder` contains several other methods with the same parameter types as method `disableInnerClassSerialization`, which results in too many compilable variants being generated.

When TRIP leverages textual similarity of PPEs, conversely, the cost associated with the PPEs needed to generate the valid repair is reduced, as they contain terms that also appear in the removed PPEs in t_2 (e.g., "exclude", "field", "inner"). The search process therefore privileges these PPEs, enabling TRIP to find the valid repair within the given time limit.

E. Limitations of TRIP

This section discusses cases in which TRIP is unable to find an intent-preserving repair for a broken test. While searching for variants of broken tests, TRIP uses only type compatibility to identify possible operations to add. This makes the search algorithm more flexible, which allows TRIP to handle many kinds of program changes. However, this flexibility comes at a cost, so TRIP works best when repairs involve the generation of short sequences of method calls that operate on objects; in these cases, the type constraints involving operation parameters and available objects tend to limit the number of compilable variants in the search space. Conversely, when repairs involve long sequences, the search space could become too large for TRIP to be successful.

As an example of this situation, Figure 10 shows a broken test in commons-math 3.2, for which TRIP was unable to find a valid repair. Test t_3 is broken because the constructor of

```
1  public void testExcludeInnerClassField() {
2   Field f = getClass().getField("innerClass");
3   InnerClassExclusionStrategy strategy =
4     new InnerClassExclusionStrategy();
5   FieldAttributes fAttr =
6     new FieldAttributes(getClass(), f);
7   assertTrue(strategy.shouldSkipField(fAttr)); }
```

(a) Broken test t_2.

```
1  public void testExcludeInnerClassField() {
2   Field f = getClass().getField("innerClass");
3   Excluder excluder = Excluder.DEFAULT;
4   excluder = excluder.disableInnerClassSerialization();
5   assertTrue(excluder.excludeField(f, true)); }
```

(b) Intent-preserving repair candidate t'_2.

Figure 9: Example of test case repair #2.

`PolyhedronSet` was changed in commons-math 3.3 to take one more parameter of type double. There are 11 values of type double in scope in t_3 (i.e., definition nodes in the corresponding DFG): the 5 constants on lines 2 and 3 plus the 6 expressions in the original call to the constructor of `PolyhedronSet`). Therefore, the number of different assignments for the 7 parameters of the modified constructor is 11^7, which is close to 20 millions. In other words, in this case, the type constraints do little to limit the number of possible compilable variants, which caused TRIP to reach a timeout before finding an intent-preserving repair. In future work, we plan to explore mixed strategies that combine our general search algorithm with heuristics aimed to repair common types of program changes. Test t_3, for instance, would be easily repaired by a strategy that simply tries to add parameters to an existing method call, rather than replacing the call with all possible (type correct) alternatives.

Our current implementation of TRIP cannot handle broken tests whose repairs involve generics. Although the technique can reuse existing objects with generic types, it cannot currently generate operations using methods that have generic type parameters. This limitation is not a conceptual one and can be addressed with extra engineering effort.

F. Threats to Validity

Internal: There may be *faults in our implementation*. To mitigate this threat, we checked and tested our code throughout development. In particular, we performed an extensive manual inspection of the results, including spot-checking many repair candidates and intent similarity scores for all the broken tests considered. We might have made *mistakes in manually determining whether the repaired tests matched the actual repairs* produced by the developers. To mitigate this threat, we performed each check multiple times. Furthermore, almost all of these checks were straightforward (see III-A3).

External: Our *benchmark might not be representative*. To mitigate this threat, we chose programs used in previous studies and that span a variety of domains—numeric computation, manipulation of complex data structures, and XML parsing. We also made sure to include in our experiment a large number of broken tests whose repairs involve non-trivial changes.

```
1  public void testBuildBox() {
2   double x = 1.0, y = 2.0, z = 3.0;
3   double w = 0.1, l = 1.0;
4   PolyhedronsSet tree = new PolyhedronSet(
5     x - l, x + l, y - w, y + w, z - w, z + w);
6   // (Assertions omitted)
7  }
```

Figure 10: Example of broken test that TRIP could not repair.

IV. RELATED WORK

Daniel and colleagues propose ReAssert [4] and Symbolic Test Repair [2], two techniques for fixing tests that are broken due to runtime exceptions and assertion failures (usually caused by changes of software specifications). ReAssert executes the broken test on P' and modifies the failed assertion so that it matches the observed runtime behavior. Symbolic Test Repair improves on ReAssert by using symbolic execution to identify and change the literal values used to compute the expected values in the assertions. Both techniques require the broken test to be still compilable and are complementary to TRIP, which focuses on non-compilable broken tests.

TestCareAssistant (TCA) [5], by Mirzaaghaei and colleagues, repairs broken tests that do not compile due to method signature changes (e.g., parameter additions or removals). TCA uses a combination of static and dynamic program analysis to identify signature changes, variables involved in the changes, and possible initialization values for these variables. Unlike TCA, TRIP can also repair tests that are broken due to program changes that are not restricted to method signature modifications. Moreover, TRIP does not rely on a mapping between removed and new PPEs based on predefined rules, but rather uses a general search-based algorithms to find compilable variants of the broken tests. Finally, TRIP explicitly models test intent and uses it to rank intent-preserving repairs.

UCov [14], by Assi and colleagues, lets developers specify test intents by manually providing logic expressions about coverage of program elements and variable values. As software evolves, UCov checks these expressions to verify that the specified test intents are still satisfied. Unlike UCov, TRIP represents test intents using path conditions that are automatically generated and does not require any manual annotation.

Similar to TRIP, API migration techniques often use calls to API methods as building blocks to generate code sequences that can replace obsolete code while preserving the underlying program semantics. Balaban and colleagues propose a technique that takes the mapping between legacy and new APIs as input and automatically migrates a program to the new APIs [15]. TRIP, in contrast, does not require any predefined mapping between removed and new APIs. Techniques by Zhong and colleagues [16] and Nguyen and colleagues [17] perform API migration by mining usage examples from a large number of client applications. Unlike these techniques, TRIP does not need examples, which are typically unavailable, and uses a search-based approach instead.

There is a rich body of research on program repair techniques (e.g., [18]–[21]) that take a faulty program and a test suite that reveals the fault and try to modify the program so that all tests pass. These techniques cannot be directly used to repair broken test cases, as their goal is to alter the (incorrect) behavior of the program.

Because GUI and web-testing scripts are expensive to generate and maintain, several researchers have proposed techniques for repairing broken GUI and web tests (e.g., [22]–[27]. Unlike these techniques, which focus on repairing testing workflows that involve GUI elements, TRIP repairs broken code sequences composed of method calls and field accesses.

V. CONCLUSION AND FUTURE WORK

We presented TRIP, a test repair technique that focuses on non-compilable broken tests and addresses a key challenge for test repair techniques: preserving the intent of the original tests in the repaired ones. Given a broken test, TRIP combines a search-based algorithm for generating repair candidates with an approach for measuring intent similarity between broken and repaired tests. This latter is based on (1) computing path conditions for broken and repaired tests by executing them in a dynamic symbolic fashion and (2) measuring the similarity between the generated path conditions.

The results of our evaluation, performed on 91 real broken tests in 4 open-source programs, are promising. They show that not only TRIP is effective at generating actual repairs for broken tests, but also that the path conditions it generates are a good abstraction of test intent. In fact, using test-intent similarity, TRIP was able to rank in first position 88% of the correct test repairs it generated.

In addition to improving some practical aspects of TRIP and performing additional experimentation, there are several direction that we plan to explore in future work. *First*, although TRIP currently focuses on repairing broken tests that do not compile, other repair candidate generation techniques could be integrated into our approach and leverage its test intent extractor and comparator capabilities. We will pursue this line of research and investigate additional repair candidate generation techniques within TRIP. *Second*, TRIP does not currently consider the behavior of the original broken test while searching for candidates, which can lead the search to consider a large number of irrelevant variants. To address this limitation, we will investigate the use of our test-intent similarity measure to guide the search of the repair candidates. *Finally*, applying TRIP to non-unit test cases may result in path conditions that are too large to be efficiently collected and compared. We will explore the use of TRIP on integration and system tests and investigate ways to represent and process test intents more efficiently as needed.

ACKNOWLEDGMENTS

This work was partially supported by NSF, under grants CCF-1161821 and 1563991, DARPA, under contracts FA8650-15-C-7556 and FA8650-16-C-7620, ONR, under contract N00014-17-1-2895, and gifts from Google, IBM Research, and Microsoft Research.

REFERENCES

[1] L. Moonen, A. van Deursen, A. Zaidman, and M. Bruntink, *On the Interplay Between Software Testing and Evolution and its Effect on Program Comprehension*. Springer Berlin Heidelberg, 2008, pp. 173–202.

[2] B. Daniel, T. Gvero, and D. Marinov, "On test repair using symbolic execution," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ser. ISSTA 2010, pp. 207–218.

[3] M. Mirzaaghaei, F. Pastore, and M. Pezzè, "Automatically repairing test cases for evolving method declarations," in *Proceedings of the 26th IEEE International Conference on Software Maintenance*, ser. ICSM 2010, pp. 1–5.

[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 of the 33rd International Conference on Software Engineering*, ser. ICSE 2011, pp. 1010–1012.

[5] M. Mirzaaghaei, F. Pastore, and M. Pezzè, "Supporting test suite evolution through test case adaptation," in *Proceedings of the 5th International Conference on Software Testing, Verification and Validation*, ser. ICST 2012, pp. 231–240.

[6] L. S. Pinto, S. Sinha, and A. Orso, "Understanding myths and realities of test-suite evolution," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE 2012, pp. 33:1–33:11.

[7] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2005, pp. 213–223.

[8] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE 2005, pp. 263–272.

[9] A. Banerjee, A. Roychoudhury, J. A. Harlie, and Z. Liang, "Golden implementation driven software debugging," in *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2010, pp. 177–186.

[10] T. Jiang, L. Wang, and K. Zhang, "Alignment of trees an alternative to tree edit," *Theoretical Computer Science*, vol. 143, no. 1, pp. 137 – 148, 1995.

[11] R. Yang, P. Kalnis, and A. K. H. Tung, "Similarity evaluation on tree-structured data," in *Proceedings of the 31st ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD 2005, 2005, pp. 754–765.

[12] "Javaparser," http://javaparser.org/ (accessed on February 2019).

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

[14] 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.

[15] I. Balaban, F. Tip, and R. Fuhrer, "Refactoring support for class library migration," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2005, pp. 265–279.

[16] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang, "Mining api mapping for language migration," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, ser. ICSE 2010, pp. 195–204.

[17] 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 *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2014, pp. 457–468.

[18] F. Long, P. Amidon, and M. Rinard, "Automatic inference of code transforms for patch generation," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017, pp. 727–739.

[19] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE 2012, pp. 3–13.

[20] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015, pp. 166–178.

[21] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *Proceedings of the 35th International Conference on Software Engineering*, ser. ICSE 2013, pp. 772–781.

[22] A. M. Memon, "Automatically repairing event sequence-based gui test suites for regression testing," *ACM Transactions on Software Engineering and Methodology*, vol. 18, no. 2, pp. 4:1–4:36, Nov. 2008.

[23] M. Grechanik, Q. Xie, and C. Fu, "Maintaining and evolving gui-directed test scripts," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE 2009, pp. 408–418.

[24] B. Daniel, Q. Luo, M. Mirzaaghaei, D. Dig, D. Marinov, and M. Pezzè, "Automated gui refactoring and test script repair," in *Proceedings of the 1st International Workshop on End-to-End Test Script Engineering*, ser. ETSE 2011, pp. 38–41.

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

[26] S. R. Choudhary, D. Zhao, H. Versee, and A. Orso, "Water: Web application test repair," in *Proceedings of the 1st International Workshop on End-to-End Test Script Engineering*, ser. ETSE 2011, pp. 24–29.

[27] A. Stocco, R. Yandrapally, and A. Mesbah, "Vista: Web test repair using computer vision," in *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018, pp. 876–879.