Crash Reproduction via Test Case Mutation

Let Existing Test Cases Help

Jifeng Xuan[†], Xiaoyuan Xie[†], Martin Monperrus[*]
[†]State Key Lab of Software Engineering, School of Computer, Wuhan University, China
[*]University of Lille & INRIA, France
{jxuan, xxie}@whu.edu.cn, martin.monperrus@univ-lille1.fr

ABSTRACT

Developers reproduce crashes to understand root causes during software debugging. To reduce the manual effort by developers, automatic methods of crash reproduction generate new test cases for triggering crashes. However, due to the complex program structures, it is challenging to generate a test case to cover a specific program path. In this paper, we propose an approach to automatic crash reproduction via test case mutation, which updates existing test cases to trigger crashes rather than creating new test cases from scratch. This approach leverages major structures and objects in existing test cases and increases the chance of executing the specific path. Our preliminary result on 12 crashes in Apache Commons Collections shows that 7 crashes are reproduced by our approach of test case mutation.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*

Keywords

Crash reproduction, test case mutation, stack trace

1. INTRODUCTION

Reproducing a crash is inevitable to software debugging. Once software goes wrong, crash information (usually stack traces) is recorded to assist future reproducing the crash scenario. For instance, in Java programs, a crash mainly contains a stack trace of runtime exceptions and crashed positions in source files. In practice, crash reproduction is manually conducted via designing a test case to trigger the crash by developers. Based on such reproduction, developers can further understand the root cause of crashes and fix the bug behind crashes.

Automatic methods of crash reproduction are proposed to reduce the manual effort by developers, such as ReCore by Rößler et al. [5], BugRedux by Jin & Orso [3], and Star by Chen & Kim [1]. These methods generate new test cases to execute the program paths, which can trigger the target crash. The criterion of a successfully reproduced crash is that a test case can produce the same stack trace as in the target crash.

In this paper, we propose an approach to automatic crash reproduction via test case mutation, called MuCrash, which updates existing test cases to trigger crashes rather than creating new test cases from scratch. Our approach leverages major structures and objects in existing test cases and increases the chance of executing a specific path that reproduces the crash. The idea of test case mutation is motivated by program mutation, which is used to identify the strength of test cases in mutation testing [2]. Note that *test case mutation* in our work aims to update an existing test case and to generate its mutants while *mutation testing* aims to update the program under test to examine whether test cases can catch the updates.

To reproduce a crash, our approach MuCrash, takes the stack trace in the crash, the source code, and existing test cases as input; the output is a set of test cases after mutation that can reproduce the crash. An *existing test case* denotes a unit test case, which is released together with the program under test and is not able to trigger the crash. The process of MuCrash consists of three major steps. First, given a stack trace, MuCrash executes all the existing test cases on the program and selects test cases that cover the classes in the stack trace. Second, MuCrash eliminates program assertions in these selected test cases and maintains the program behavior inside assertions. Third, given a set of pre-defined mutation operators (e.g., an operator of setting one variable in a method call to `null` or adding a target method call), each selected test case produces a set of test case mutants. These resulting test cases are executed on the program and the ones that can reproduce crashes are extracted and sent to developers for manually verifying.

Our preliminary result on 12 crashes in Apache Commons Collections shows that 7 crashes are reproduced by our approach of test case mutation. One reproduced crash of ACC-331 by MuCrash is the first time of successful reproduction by automatic methods. This crash was not reproduced by the state-of-the-art method, Star [1].

This paper makes the following contributions.

1. **Test case mutation**. This method generates new test cases by updating existing test cases and keeps object-oriented features in test cases.

2. MuCrash**, an approach to crash reproduction with existing test cases**. This approach employs test case

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3675-8/15/08...$15.00
http://dx.doi.org/10.1145/2786805.2803206

mutations as a lightweight technique to reproduce crashes by exploring potential execution paths, compared with symbolic execution techniques in BugRedux and Star.

2. MOTIVATION AND NEW IDEA

We use a real-world bug in Apache Commons Collections to illustrate the motivation and the new idea of our work. *Apache Commons Collections* is a widely-used Java library of enhanced usage of collections, such as lists, maps, or sets.

Bug report 331 of Collections (ACC-331 for short) describes a null pointer exception in a class `CollatingIterator` in Version 3.2 or before. According to the document, a feature of `CollatingIterator` is to accept a null comparator as input and to load a default comparator to perform the comparison between two ordered list (note that this feature is discarded in its subsequent version since 4.0, due to complexity). Figure 1 shows code snippets and related test cases, which are written by developers and generated by our technique of test case mutation.[1]

As shown in Figure 1a, the buggy source code did not call a default comparator when a user inputs a null comparator. The bug will trigger a crash shown in Figure 1b. This crash records the stack trace of calling methods in Class `CollatingIterator`. However, it is hard to reproduce the scenario where the program crashes. During the process of debugging, a developer wrote a test case `testNullComparator()` to trigger the crash, in Figure 1c (triggered at Line 7). This test case will help developers to understand the root cause behind the crash.

Writing a test case in Figure 1c is not easy. On one hand, a human developer needs to fully understand the functionality of the constructor of `CollatingIterator` (Line 5) and to initialize two lists (Lines 2-3) for calling the method `next()` (Line 7). On the other hand, an automatic method of test case generation must handle a constructor with three variables (Line 5), especially with two variables of complex iterators; meanwhile, automatically creating a method call sequence like `hasNext()` and `next()` (Lines 6-7) is also challenging. In the state-of-the-art method, Star, the authors claim that they fail to reproduce the crash of ACC-331 [1].

Our new idea of crash reproduction is to leverage existing test cases, instead of creating new test cases from scratch. As shown in Figure 1d, in the test suite (which is released together with the buggy source code), a unit test case `testIterateEvenOdd()` is provided to test the general functionality of `CollatingIterator`. In this test case, two lists, `evens` and `odds`, are already provided. Once we change the `comparator` at Line 8 into `null`, a resulting test case will trigger the target crash. This resulting test case is shown in Figure 1e. Without changing the lists of `evens` and `odds`, code at Line 6 in Figure 1e will lead to the target stack trace in Figure 1b. Note that due to the change at Lines 2-3, original assertions may be incorrect and are replaced by Lines 5, 6, and 8.

The change from Line 8 in Figure 1d to Line 3 in Figure 1e is to set one variable in the method call to `null`. In this paper, such a change in a test case is called *test case mutation*, which is motivated by the concept of "mutation operators for programs" in mutation testing. Section 3.2.3 will present more useful mutation operators for test cases.

[1]Bug report of ACC-331, `http://issues.apache.org/jira/browse/COLLECTIONS-331`; its manually-written test case by developers for reproducing the crash, `http://issues.apache.org/jira/secure/attachment/12411653/CollatingIteratorTest.java`.

```
private int least() {
  Object curObject = values.get(i);
- if (comparator.compare(curObject,leastObject) < 0) {
+ Comparator comp = comparator == null ?
        ComparableComparator.getInstance() : comparator;
+ if (comp.compare(curObject,leastObject) < 0) {
      leastObject = curObject;
...
```

(a) Buggy code snippet (-) with its patch (+) in the program

```
java.lang.NullPointerException:
[...].iterators.CollatingIterator.least (CollatingIterator.java:333)
[...].iterators.CollatingIterator.next (CollatingIterator.java:229)
...
```

(b) Stack trace during software crashing

```
1 public void testNullComparator() {
2   List<Integer> l1 = Arrays.asList(1, 3, 5);
3   List<Integer> l2 = Arrays.asList(2, 4, 6);
4   CollatingIterator collatingIterator = new
5       CollatingIterator(null, l1.iterator(), l2.iterator());
6   for (int i = 0; collatingIterator.hasNext(); i++ )
7     Integer n = (Integer) collatingIterator.next();
8   ...
9 }
```

(c) Manually-written test case by developers for reproduction

```
1 private ArrayList evens = null;
2 private ArrayList odds = null;
3 public void setUp() throws Exception {
4 //Initialize evens and odds with 20 elements (before each test case)
5 }
6 public void testIterateEvenOdd() { // Existing unit test case
7   CollatingIterator iter = new
8       CollatingIterator(comparator, evens.iterator(), odds.iterator());
9   for(int i=0; i<20; i++) {
10    assertTrue(iter.hasNext());
11    assertEquals(new Integer(i), iter.next());
12  }
13  assertTrue(! iter.hasNext());
14 }
```

(d) Existing unit test case that is released with the buggy program

```
1 public void testIterateEvenOdd_MUTATION() {
2   CollatingIterator iter = new
3       CollatingIterator(null, evens.iterator(), odds.iterator());
4   for(int i=0; i<20; i++) {
5     Object obj1 = iter.hasNext();
6     Object obj2 = iter.next();
7   }
8   Object obj3 = ! iter.hasNext();
9 }
```

(e) Test case that is generated via test case mutation: by setting one variable to null

Figure 1: Real-world example of Bug ACC-331 in Class `CollatingIterator`. To reproduce the crash in Figure 1b, developers write a test case in Figure 1c; our method of test case mutation generates a test case in Figure 1e based on an existing test case in Figure 1d. Differences between the existing test case (in green) and the mutated test case (in red) are marked with boxes.

3. APPROACH: TEST CASE MUTATION

We present our approach to automatic crash reproduction via test case mutation, MuCrash. The goal of our approach is to trigger a specific execution path by updating existing test cases. The technique of test case mutation bridges existing test cases and target test cases, which are expected to reproduce crashes.

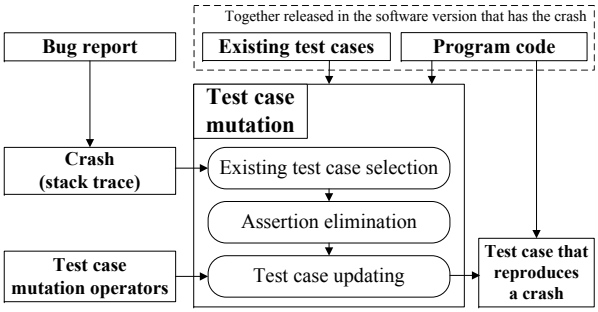

Figure 2: Overview of crash reproduction via test case mutation, MuCrash.

3.1 Overview

Figure 2 illustrates the overview of our proposed approach to crash reproduction via test case mutation. As an input, crash information in a bug report is usually expressed as stack traces, which record runtime exceptions and crashed positions in source code files; the other inputs are program source code as well as its related unit test cases that are not able to trigger the target crash. In modern software development, unit test cases are released together with source code to validate the software configuration.

Our proposed approach consists of three major steps: existing test case selection, assertion elimination, and test case updating. After applying a set of pre-defined mutation operators, an existing test case will be updated to a set of new test cases, each of which may provide different stack traces from the trace by the existing test case. These new test cases are executed on the buggy program; test cases that are successful in crash reproduction are extracted as output.

3.2 Test Case Mutation

3.2.1 Existing Test Case Selection

Our approach updates existing test cases to reproduce crashes. To reduce the number of considered existing test cases, we select a subset of existing test cases for subsequent steps. Given a stack trace, we extract classes in the trace, e.g., Class `CollatingIterator` in Figure 1b. Then, all existing test cases are executed; test cases that cover the classes in the trace are selected. Such selection discards irrelevant test cases and narrows down the range of potential existing test cases. We conduct this selection under an assumption that the selected test cases may relate to *methods* in the crash since these test cases have contributed to the testing of *classes* in the stack trace.

3.2.2 Assertion Elimination

In a test case, developers write assertions to ensure that software meets the expected behavior. For instance, code at Line 11 in Figure 1d ensures `iter.next()` equals to `i`. In our approach, the goal of test case mutation is to modify existing test cases to trigger new traces. Once a statement in a test case is changed, assertions are no longer applicable.

To remove the judgment of assertions, we conduct "assertion elimination" for all assertions in the selected test cases. *Assertion elimination* keeps the behavior of variables inside an assertion and removes the statement of assertions. For instance, `assertEquals(new Integer(i), iter.next())` at Line 11 in Figure 1d is changed into `Object obj2 = iter.next()`, as shown at Line 6 in Figure 1e. Such changes in assertion elimination will not add or reduce the program behavior to test cases, except the judgment by assertions. The kept program behavior can facilitate the crash reproduction. For instance, the crash in Figure 1e is triggered at Line 6.

Table 1: Test case mutation operators in our approach

Index	Mutation operator	Operator description
1	Variable null	Set a variable in a method call to `null`
2	Variable renewing	Set a variable in a method call to a newly-created object
3	Numeric variable replacement	Replace a numeric variable with default values (e.g., 0 or 1) and with existing values in the same test case
4	Method call addition	Add a call of a method that appears in the stack trace
5	Overloading method call addition	Add a call of an overloading method that appears in the stack trace

3.2.3 Test Case Updating

In *test case updating*, we modify selected test cases (after assertion elimination) to produce new traces. We leverage the concept of "mutation operators for programs" from mutation testing. A *mutation operator* is a pre-defined transformation rule that generates new program from an original program [2].

In this paper, we use *test case mutation operators* to update test cases. To facilitate crash reproduction, we focus on mutation operators with object-oriented features. Table 1 lists five types of test case mutation operators in our work. The first three mutation operators are related to update a variable in a method call while the other two are related to adding a method call that exists in the stack trace.

In contrast to general operators in mutation testing, test case mutation operators in our work could be guided by the stack trace. For instance, adding one method call in the trace to an existing test case (i.e., operator 4 in Table 1) may increase the opportunity of successful reproduction. In this work, we exhaustively apply all the mutation operators without any selection.

Note that applying one mutation operator to one test case will lead to more than one new test cases. For example, applying the first operator (Variable null) to code at Line 8 in Figure 1d will lead to three test cases with a `null` variable in a method call.

3.3 Novelty of Proposed Approach

In test case generation, to our knowledge, this is the first work of mutating test cases (in a style of program mutation). Previous work has explored generating new test cases from existing ones. The most related work is test case regeneration by Yoo & Harman [9] and test suite augmentation by Xu et al. [7]. However, their work treats existing test cases as seeds of new test case generation while our work of test case mutation directly updates existing test cases with mutation operators. Another related work is data mutation in modeling languages by Shan & Zhu [6]. Their work updates model diagrams to test a modeling tool while our work directly manipulates test cases for crash reproduction. In summary, test case mutation reuses object-oriented features inside existing test cases to avoid the complexity of creating new ones from scratch.

Table 2: Bug IDs of reproduced crashes among 12 bugs

Method	Bug IDs of reproduced crashes	Useless reproduction	#Reproduced crash	#Useful reproduction
Star	4, 28, 35, 48, 104, 411, **53**, **77**	77	8	7
MuCrash	4, 28, 35, 48, 104, 411, **331**	-	7	7

The most related work by authors is test case purification for fault localization [8]. Both the technique and the goal are different from those in test case mutation: previous work [8] splits failing test cases into small parts to improve the effectiveness of fault localization.

4. PRELIMINARY RESULT

4.1 Dataset and Implementation

In the state-of-the-art method of crash reproduction, Star [1], bug reports from three open-source Java projects are used for evaluation.[2] In our preliminary experiment, we use all the 12 bugs from one of their projects, Apache Commons Collections. Collections contains 26 KLoC of source code and 29 KLoC of test code (the latest version under evaluation).

Our prototype of MuCrash is implemented on the top of Spoon. Spoon [4] is a static library for Java program analysis and transformation.

4.2 Result of Reproducing 12 Crashes

Table 2 shows a preliminary result of crashes of 12 bug reports in Collections. We compare the proposed MuCrash with the state-of-the-art method, Star [1]. As defined in [1], a *reproduced crash* denotes that the generated test case triggers the same stack trace while a *useful reproduction* means that the crash reproduction is helpful to fix the bug. We manually check whether a reproduced crash is useful or not.

As shown in Table 2, MuCrash reproduces 7 out of 12 crashes while Star reproduces 8 out of 12 crashes. MuCrash is able to reproduce ACC-331 while MuCrash fails to reproduce ACC-53 and ACC-77. The major reason for this failure is that test cases for both crashes require frequent method calls, which cannot be directly performed by mutation operators.

Based on our manual check, all 7 reproduced crashes by MuCrash are useful to fix the bug while as reported by Chen & Kim [1], the reproduction of ACC-77 by Star is useless. Hence, both MuCrash and Star can reproduce 7 useful ones among 12 crashes.

The reproduced crash, ACC-331 (see Figure 1 for details), by MuCrash, shows the strength of test case mutation in crash reproduction. Comparing with techniques of symbolic execution and precondition analysis in Star, test case mutation in MuCrash is lightweight: it will add no runtime overhead of computing resources, except executing test cases; meanwhile, test case mutation maintains object-oriented features in existing test cases, which are updated to trigger new crashes.

To further understand the role of test case mutation in crash reproduction, Table 3 shows which mutation operator works for the reproduced crashes. Among seven reproduced crashes by MuCrash, test cases in four crashes are mutated by updating variables while test cases in the other three crashes are mutated by adding method calls from stack traces. ACC-331, the newly reproduced crash by MuCrash, is based on a simple mutation operator, i.e., to set a variable to `null`.

[2]Star Project, `http://sites.google.com/site/starcrashstack/`.

Table 3: Mutation operators that reproduce crashes

Bug ID	Buggy version	Mutation operator (see Table 1)
ACC-4	Collections 2.0	1 - Variable null
ACC-28	Collections 2.0	1 - Variable null **or** 2 - Variable renewing
ACC-35	Collections 2.1	4 - Method call addition
ACC-48	Collections 3.1	4 - Method call addition
ACC-104	Collections 3.1	5 - Overloading method call addition
ACC-331	Collections 3.2	1 - Variable null
ACC-411	Commit, r-1351903	3 - Numeric variable replacement

4.3 Discussion

Crash reproduction via test case mutation relies on the quality of existing test cases. The ability of reproducing crashes will be limited if the existing test cases are not well-designed for the software under test. We intend to mainly apply our method to projects with high-quality test suites, such as Apache Commons in this paper.

In our method, we exhaustively update test cases with all mutation operators; no selection of mutation operators is employed. This may lead to a large number of unnecessary test cases, which are helpless to crash reproduction. A potential solution to reduce the helpless ones is to prioritize mutation operators for successful reproduction.

5. CONCLUSIONS

This paper proposes a new approach to crash reproduction via test case mutation. This approach updates existing test cases to form new test cases for triggering crashes. In our preliminary study, 7 out of 12 crashes are reproduced and useful to bug fixing; this result achieves the same number of useful crashes by the state-of-the-art method, Star. Moreover, one crash ACC-331 is newly reproduced, which was reported non-reproduced previously.

Remained research questions. We plan to further understand the ability of test case mutation via an empirical study of more crashes. This study will explore the different power of test case mutation and symbolic-execution based test case generation. We plan to investigate the prioritization of applying mutation operators for crash reproduction.

6. REFERENCES

[1] N. Chen and S. Kim. STAR: stack trace based automatic crash reproduction via symbolic execution. *IEEE Trans. Software Eng.*, 41(2):198–220, 2015.

[2] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Trans. Software Eng.*, 37(5):649–678, 2011.

[3] W. Jin and A. Orso. Bugredux: Reproducing field failures for in-house debugging. In *ICSE 2012*, pages 474–484, 2012.

[4] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier. Spoon: A Library for Implementing Analysis and Transformations of Java Source Code. *Software: Practice and Experience*, 2015.

[5] J. Rößler, A. Zeller, G. Fraser, C. Zamfir, and G. Candea. Reconstructing core dumps. In *ICST*, pages 114–123, 2013.

[6] L. Shan and H. Zhu. Testing software modelling tools using data mutation. In *AST 2006*, pages 43–49. ACM, 2006.

[7] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. B. Cohen. Directed test suite augmentation: techniques and tradeoffs. In *FSE 2010*, pages 257–266. ACM, 2010.

[8] J. Xuan and M. Monperrus. Test case purification for improving fault localization. In *FSE 2014*, pages 52–63. ACM, 2014.

[9] S. Yoo and M. Harman. Test data regeneration: generating new test data from existing test data. *Softw. Test., Verif. Reliab.*, 22(3):171–201, 2012.