Mutation Analysis of Parameterized Unit Tests

Tao Xie[1] Nikolai Tillmann[2] Jonathan de Halleux[2] Wolfram Schulte[2]
[1]Department of Computer Science, North Carolina State University, Raleigh, NC 27695, USA
[2]Microsoft Research, One Microsoft Way, Redmond, WA 98052, USA
xie@csc.ncsu.edu, {nikolait,jhalleux,schulte}@microsoft.com

Abstract

Recently parameterized unit testing has emerged as a promising and effective methodology to allow the separation of (1) specifying external, black-box behavior (e.g., assumptions and assertions) by developers and (2) generating and selecting internal, white-box test inputs (i.e., high-code-covering test inputs) by tools. A parameterized unit test (PUT) is simply a test method that takes parameters, specifies assumptions on the parameters, calls the code under test, and specifies assertions.

The test effectiveness of PUTs highly depends on the way that they are written by developers. For example, if stronger assumptions are specified, only a smaller scope of test inputs than intended are generated by tools, leading to false negatives in terms of fault detection. If weaker assertions are specified, erroneous states induced by the test execution do not necessarily cause assertion violations, leading to false negatives. Detecting these false negatives is challenging since the insufficiently written PUTs would just pass. In this paper, we propose a novel mutation analysis approach for analyzing PUTs written by developers and identifying likely locations in PUTs for improvement. The proposed approach is a first step towards helping developers write better PUTs in practice.

1 Introduction

Unit testing has been widely recognized as an important and valuable means of improving software reliability, partly due to its capabilities of exposing faults early in the development life cycle. Recently parameterized unit testing [9] has emerged as a promising and effective methodology of extending the current industry practice of closed unit tests (i.e., test methods without parameters). In parameterized unit testing, test methods are generalized by allowing parameters. This generalization serves two purposes. First, parameterized unit tests are *specifications* of the behavior of the methods under test: they do not provide only exemplary arguments to the methods under test, but provide ranges of such arguments. Second, parameterized unit tests describe a set of traditional unit tests that can be obtained by *instantiating* the parameterized unit tests with given argument-value sets (generated either automatically or manually). Instantiations should be chosen so that they exercise different code paths of the methods under test.

In particular, parameterized unit testing allows the separation of (1) specifying external, black-box behavior (e.g., assumptions and assertions) by developers and (2) generating and selecting internal, white-box test inputs (i.e., high-code-covering test inputs) by tools. A parameterized unit test (PUT) is simply a test method that takes parameters, specifies assumptions on the parameters, calls the code under test, and specifies assertions. Admittedly, writing open PUTs is more challenging than writing closed traditional unit tests.

The test effectiveness of PUTs highly depends on the way that they are written by developers. (1) If *stronger assumptions* (i.e., defining a smaller input domain than intended) are specified, only a smaller scope of test inputs than intended are generated by tools, leading to false negatives in terms of fault detection. (2) If *weaker assumptions* are written, a larger scope of test inputs than intended are generated by tools, leading to false positives (since correctly specified assertions are violated by some generated test inputs that should have been filtered by intended assumptions). (3) If *stronger assertions* (i.e., defining a smaller allowable state domain than intended) are specified, violating these assertions does not necessarily indicate real faults, leading to false positives. (4) If *weaker assertions* are specified, erroneous states induced by test execution do not necessarily cause assertion violations, leading to false negatives. Among the preceding four cases, false positives (and their corresponding insufficiently specified assumptions or assertions) could be easily detected based on reported assertion violations. However, detecting false negatives (or their corresponding insufficiently specified assumptions or assertions) is challenging since the insufficiently written PUTs would just pass.

To address this issue, we propose a novel mutation analysis approach for analyzing PUTs written by developers and identifying likely locations in PUTs for improvement. The proposed approach is a first step towards helping developers write better PUTs in practice.

In our mutation analysis approach, we propose a set of *mutation operators* for systematically mutating PUTs written by developers. Our mutation operators simulate the effect of making a (likely insufficiently written) PUT (1) more *general* in terms of allowing *weaker assumptions* to be used to generate more test inputs (while violating no assertions) or (2) more *specialized* in terms of allowing *stronger assertions* to be checked (while still being satisfied by the generated test inputs).

Applying any of these mutation operators on a PUT produces one mutant PUT. Then a mutant PUT is determined to be *killed* if a test input can be generated by tools to make the mutant PUT fail (e.g., an assertion in the mutant PUT is violated). In traditional mutation testing [4], the more mutants are killed, the higher quality the *generated test inputs* are of. Here such a notion still applies. However, the more important implication of mutation testing here is as follows: the more mutant PUTs are killed, the higher quality the *original PUT* is of, and the less improvement space the *original PUT* can have. In addition, any live mutant PUT (i.e., one not being killed) indicates a potential improvement (e.g., further generalization on assumptions or specialization on assertions) of the original PUT by replacing it with the live mutant PUT. Such live mutant PUTs are recommended to developers for their consideration of PUT improvement.

2 Example

Throughout the rest of this paper, we illustrate our approach via a running example. The example is an integer stack implementation in C# adapted from one used by Henkel and Diwan [5]. Figure 1 shows the relevant parts of the code. The array `Store` contains the elements of the stack, and `Size` is the number of the elements and the index of the first free location in the stack. The method `Push`/`Pop` appropriately increases/decreases the size after/before writing/reading the element. Additionally, `Push` allows only non-negative integer elements to be pushed to the stack, and `Push` grows the array when the `Size` is equal to the whole array's length. `Pop` returns `-1` if the stack is empty; otherwise, it returns the element being popped. The method `IsEmpty` is an observer that checks if the stack has any elements, and the method `Equals` compares two stacks for equality.

A *parameterized unit test* (PUT) [9] is simply a method that takes parameters, specifies assumptions on the parameters, calls the code under test, and specifies assertions. For example, the test methods in Figure 3 are example PUTs for

```
public class IntStack {
  private int[] Store;
  private int Size;
  private static int INITIAL_CAPACITY = 10;
  public IntStack() {
    this.Store = new int[INITIAL_CAPACITY];
    this.Size = 0;
  }
  public void Push(int value) {
    if (value < 0) return;
    if (this.Size == this.Store.Length) {
      int[] store = new int[this.Store.Length * 2];
      Array.Copy(this.Store, store, this.Size);
      this.Store = store;
    }
    this.Store[this.Size++] = value;
  }
  public int Pop() {
    if (this.Size == 0) return -1;
    return this.Store[--this.Size];
  }
  public bool IsEmpty() {
    return (this.Size == 0);
  }
  public bool Equals(Object other) {
    if (!(other is IntStack)) return false;
    IntStack s = (IntStack)other;
    if (this.Size != s.Size) return false;
    for (int i = 0; i < this.Size; i++)
      if (this.Store[i] != s.Store[i]) return false;
    return true;
  }
}
```

Figure 1. An integer stack implementation

the `IntStack` class in Figure 1, whereas Figure 2 shows a traditional unit test. These PUTs are in the format supported by Pex [8], an automatic unit testing tool from Microsoft Research. A PUT is annotated with "`[PexMethod]`". Note that some lines in the PUTs are commented out with a prefix of "`//`" while being kept there for later illustration purposes. Let us consider the last PUT `TestPushPopPUT4` in Figure 3. It has two parameters: an `IntStack` object and an integer. The `IsTrue` method of the `PexAssume` class specifies an assumption for the PUT: any test input (i.e., PUT argument values) that violates the assumption is discarded without further being checked against specified assertions. The `IsTrue` method of the `PexAssert` class specifies an assertion for the PUT: a problem is detected when the assertion is violated by a test input that satisfies the specified assumptions. The `TestPushPopPUT4` basically specifies that for any non-null `IntStack` object and any non-negative integer, after pushing this integer to the stack, popping the stack would return this integer.

The test effectiveness of PUTs highly depends on the way that they are written by developers. For example, if stronger assumptions (i.e., defining a smaller input domain than intended) are specified, only a smaller scope of test inputs than intended are generated by tools, leading to false negatives in terms of fault detection. For example, in contrast to `TestPushPopPUT4` (the most general PUT in Figure 3), `TestPushPopPUT3` allows the intended pushing and popping behavior to be checked on only an empty `IntStack`. In fact, such behavior is applicable and should be checked on an `IntStack` with any elements.

```
  [PexMethod]
  public void TestPushPop() {
1    IntStack s = new IntStack();
2    s.Push(3);
3    s.Push(5);
4    Assert.IsTrue(s.Pop() == 5);
  }
```

Figure 2. Traditional unit test

```
  [PexMethod]
  public void TestPushPopPUT1(int j) {
1    PexAssume.IsTrue(j >= 0);
2    IntStack s = new IntStack();
3    s.Push(j);
4    s.Push(5);
5    //PexAssert.IsTrue(s.Pop() > -1);
6    //PexAssert.IsTrue(s.Pop() > 0);
7    PexAssert.IsTrue(s.Pop() == 5);
  }

  [PexMethod]
  public void TestPushPopPUT2(int j, int i) {
8    //PexAssume.IsTrue(i > 0);
9    PexAssume.IsTrue(i >= 0);
10   IntStack s = new IntStack();
11   s.Push(j);
12   s.Push(i);
13   PexAssert.IsTrue(s.Pop() == i);
  }

  [PexMethod]
  public void TestPushPopPUT3(int i) {
14   PexAssume.IsTrue(i >= 0);
15   IntStack s = new IntStack();
16   s.Push(i);
17   PexAssert.IsTrue(s.Pop() == i);
  }

  [PexMethod]
  public void TestPushPopPUT4(IntStack s, int i) {
18   PexAssume.IsTrue(s != null);
19   PexAssume.IsTrue(i >= 0);
20   s.Push(i);
21   PexAssert.IsTrue(s.Pop() == i);
  }
```

Figure 3. Parameterized unit tests

If weaker assertions (i.e., defining a larger allowable state domain than intended) are specified, erroneous states induced by test execution do not necessarily cause assertion violations, leading to false negatives. For example, assume that instead of Line 7, Line 5 is used for the assertion in the first PUT `TestPushPopPUT1`. Comparing to the more desirable assertion in Line 7, the assertion in Line 5 provides less constrained checking: the execution of a faulty implementation of `IntStack` can pass the assertion in Line 5 but violate the assertion in Line 7.

Detecting false negatives (or their corresponding insufficiently specified assumptions or assertions) is challenging since the insufficiently written PUTs would just pass. The goal of our mutation analysis approach is to identify likely locations in PUTs for improvement. For example, our approach helps improve the PUTs closer to the top of the PUT list in Figure 3 to be ones closer to the bottom.

3 Approach

The input to our mutation analysis approach is the program under test and its passing PUT: a PUT where no test inputs can be generated (by a test generation tool such as Pex [8]) to violate specified assertions in the PUT while satisfying the specified assumptions. The output of our approach is a subset of mutant PUTs (mutated from the original passing PUT) being recommended to the developers. The developers can inspect this subset of mutant PUTs for likely locations of the original PUT for improvement. We next present the two key components of the approach: mutation killing (used to select which mutant PUTs to recommend to developers) and mutation operators (used to generate mutant PUTs).

3.1 Mutation Killing

Given a mutant PUT generated with the mutation operators described in the next subsection, our approach determines the mutant PUT to be *live* if no test inputs can be generated (by a test generation tool such as Pex) to violate specified assertions in the mutant PUT while satisfying the specified assumptions, and to be *killed* otherwise.

The mutation operators described in the next subsection are specifically designed to produce mutant PUTs that are more general on assumptions and more specialized on assertions than the original PUT. In traditional mutation testing [4], the more mutants are killed, the higher quality the *generated test inputs* are of. Here such a notion still applies. However, the more important implication of mutation testing here is as follows: the more mutant PUTs are killed, the higher quality the *original PUT* is of. In addition, any live mutant PUT indicates a potential improvement (e.g., further generalization on assumptions or specialization on assertions) of the original PUT by replacing it with the live mutant PUT. Such live mutant PUT can be recommended to developers for their consideration of PUT improvement.

3.2 Mutation Operators

Our mutation operators simulate the effect of making a PUT (1) more *general* in terms of allowing *weaker assumptions* to be used to generate more test inputs (while violating no assertions) or (2) more *specialized* in terms of allowing *stronger assertions* to be checked (while still being satisfied by the generated test inputs).

We devise four main types of mutation operators in our approach: (1) weakening constraints specified in assumptions of a PUT, (2) strengthening constraints specified in assertions of a PUT, (3) replacing a primitive value in a PUT with an additional parameter, and (4) deleting a method invocation from a PUT.

Table 1. Example mutation rules for clauses from PUT assumptions or assertions ($const$: positive constant) (extended from [6])

Clause	Mutated Clauses	
	Strengthening	Weakening
$P == Q$	—	$P >= Q, P <= Q$
$P! = Q$	$P > Q, P < Q,$ $P == (Q + const),$ $P == (Q - const)$	—
$P > Q$	$P > (Q + const),$ $P == (Q + const)$	$P >= Q, P! = Q$
$P < Q$	$P < (Q - const),$ $P == (Q - const)$	$P <= Q, P! = Q$
$P >= Q$	$P > Q, P == Q,$ $P == (Q + const)$	$P >= (Q - const)$
$P <= Q$	$P < Q, P == Q,$ $P == (Q - const)$	$P <= (Q + const)$

3.2.1 Assumption Weakening

Assumptions in a PUT are stronger than intended when the intended input constraints (under which the specified assertions in the PUT should be satisfied) allow more inputs than the specified assumptions allow. Thus, if assumptions of a PUT are stronger than intended, a test generation tool will not generate or keep test inputs whose execution should have been checked against the specified assertions. Such a case reduces the fault-detection capability of the PUT. For this type of insufficiency of PUTs, we provide two mutation operators: deleting an assumption from the PUT and weakening a clause in an assumption. Example mutation rules for weakening a clause are shown in Column 3 of Table 1.

For example, based on the mutation operator of "deleting an assumption from the PUT", deleting Line 1 in the first PUT `TestPushPopPUT1` leads to a live mutant PUT. This live mutant PUT indicates that the assumption in Line 1 is not necessary for preventing the specified assertion from being violated. As another example, let us assume that the second PUT `TestPushPopPUT2` is using the assumption in Line 8 instead of the one in Line 9. Based on the mutation operator of "weakening a clause in an assumption" (more precisely weakening $P > Q$ to $P >= Q$), our approach produces a live mutant PUT. This live mutant PUT indicates that the assumption in Line 8 can be weakened to be the one in Line 9, allowing more test inputs to be generated and checked against the specified assertion.

3.2.2 Assertion Strengthening

Assertions in a PUT are weaker than intended when the constraints in the specified assertions allow erroneous program states not to violate the assertions. Thus, if assertions of a PUT are weaker than intended, generated test inputs may not cause violations of the assertions even when these test inputs cause erroneous program states. Such a case reduces the fault-detection capability of the PUT. For this type of insufficiency of PUTs, we provide one mutation operator: strengthening a clause in an assertion. Example mutation rules for strengthening a clause are shown in Column 2 of Table 1.

For example, let us assume that the first PUT `TestPushPopPUT1` is using the assertion in Line 5 instead of the one in Line 6 or 7. Based on the mutation operator of "strengthening a clause" (more precisely weakening $P > Q$ to $P > (Q + const)$), our approach produces a live mutant PUT that uses the assertion in Line 6. As another example, based on the mutation operator of "strengthening a clause" (more precisely weakening $P > Q$ to $P == (Q + const)$), our approach produces a live mutant PUT that uses the assertion in Line 7. These live mutant PUTs indicate that the assertion in Line 5 or 6 can be strengthened to be the one in Line 7 to allow more error-exposing test inputs to violate the specified assertion.

3.2.3 Primitive-Value Generalization

Promoting primitive values or objects inside a test method as its parameters is a way to exploit more benefits of a PUT. For this purpose, we provide one mutation operator: replacing a constant primitive value with a parameter of the PUT. For the convenience of illustration, let us consider the traditional unit test shown in Figure 2 (on which our approach can also be applied by treating it as a special PUT). Based on the mutation operator, our approach generalizes `3` in Line 2 of `TestPushPop` (Figure 2) to be a parameter `j` in the generated live mutant PUT (shown as `TestPushPopPUT1` in Figure 3 except for being without its first line).

3.2.4 Method-Invocation Deletion

There can be extra method invocations (including constructor invocations) in a PUT that could reduce the generality of the PUT. First, the developers may hard-code the construction of a particular object (such as the receiver object) inside the PUT; the intended object (that should have been used to specify the behavior) could be just any object instead of the specific fixed object. Second, when setting up an object state (such as a receiver-object state) for invoking the method under test, the developers may add extra unnecessary method invocations that induce a smaller scope of object states than intended. To address these issues, we provide one mutation operator: deleting a method invocation in the PUT. When a constructor invocation is deleted, the mutant PUT includes an extra parameter of the constructor class type and an assumption that this parameter is not a null reference. After a method invocation is deleted, any unused parameters of the PUT are also deleted.

For example, based on the mutation operator, deleting the method invocation in Line 11 in the second PUT `TestPushPopPUT2` leads to a live mutant PUT (`TestPushPopPUT3`). This live mutant PUT indicates that the method invocation in Line 11 is not necessary for preventing the specified assertion from being violated. As another example, based on the mutation operator, deleting the constructor invocation in Line 15 in the third PUT `TestPushPopPUT3` leads to a live mutant PUT (`TestPushPopPUT4`). This live mutant PUT indicates that providing any `IntStack` object (beyond an empty `IntStack` object) before Line 16 can prevent the specified assertion from being violated.

4 Related Work

Previous work [1–3] mutates a model and then uses the model mutants to assess the quality of a test suite or generate new test inputs. In contrast, our approach focuses on assessing the quality of the assumptions or assertions (which can be seen as properties) being specified in a PUT.

Martin et al. [7] proposed a mutation verification approach to assess the quality of properties specified for an access control policy in a declarative specification language. Their approach mutates a policy and see whether the given properties can be violated by each mutant policy. Our approach also targets at identifying insufficiency of properties specified in a PUT. However, unlike their approach in mutating the software under verification, our approach mutates properties.

Hou et al. [6] mutate component-interface contracts to simulate possible faults. They use mutation killing score to help select and prioritize tests in regression testing. Our mutation operators on weakening or strengthening a clause are extended from theirs. However, our mutation analysis provides a broader scope of mutation operators and targets at a totally different problem.

5 Conclusion

Parameterized unit testing has emerged as a promising and effective methodology in developer testing. However, the test effectiveness of parameterized unit tests (PUTs) highly depends on the way that they are written by developers. Providing high generality on assumptions and high specialization on assertions in a PUT is important for providing high fault-detection capability. To help the developers improve their PUTs, we have proposed a mutation analysis approach for identifying likely locations in a PUT for developers to improve. In our approach, we propose the determination of mutation killing and a set of mutation operators for producing mutant PUTs. In future work, we plan to pursue future directions as listed below.

We plan to automate the generation of mutant PUTs with the proposed mutation operators, and empirically investigate the effectiveness of the proposed approach in improving the fault-detection capability of PUTs written by developers. We also plan to expand the current set of mutation operators proposed in our approach, and empirically investigate the effectiveness of different mutation operators.

To improve the effectiveness of our mutation analysis in terms of producing live mutant PUTs, we plan to use static and dynamic analysis to help identify the right `const` values listed in Table 1 given an assumption or assertion. In addition, sometimes when we apply the mutation operators of primitive-value generalization or method-invocation deletion to produce live mutant PUTs, additional appropriate assumptions may need to be added to the mutant PUTs, and some assertions may need to be deleted or weakened to make the mutant PUT live.

Acknowledgments

This material is based upon work supported in part by the U. S. Army Research Laboratory and the U. S. Army Research Office under contract/grant number W911NF-08-1-0443.

References

[1] P. Ammann and P. Black. A specification-based coverage metric to evaluate test sets. In *Proc. HASE*, pages 239–248, 1999.

[2] P. Ammann, P. Black, and W. Majurski. Using model checking to generate tests from specifications. In *Proc. ICFEM*, pages 46–54, 1998.

[3] P. Black, V. Okun, and Y. Yesha. Mutation operators for specifications. In *Proc. ASE*, pages 81–88, 2000.

[4] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.

[5] J. Henkel and A. Diwan. Discovering algebraic specifications from Java classes. In *Proc. ECOOP*, pages 431–456, 2003.

[6] S.-S. Hou, L. Zhang, T. Xie, H. Mei, and J.-S. Sun. Applying interface-contract mutation in regression testing of component-based software. In *Proc. ICSM*, pages 174–183, 2007.

[7] E. Martin, J. Hwang, T. Xie, and V. Hu. Assessing quality of policy properties in verification of access control policies. In *Proc. ACSAC*, pages 163–172, 2008.

[8] Microsoft Pex development team. Pex. `http://research.microsoft.com/Pex`, 2008.

[9] N. Tillmann and W. Schulte. Parameterized unit tests. In *Proc. ESEC/FSE*, pages 253–262, 2005.