# Retrofitting Legacy Unit Tests for Parameterized Unit Testing

Madhuri R. Marri[1], Suresh Thummalapenta[1], Tao Xie[1], Nikolai Tillmann[2], Jonathan de Halleux[2]
[1]Department of Computer Science, North Carolina State University, Raleigh, NC
[2]Microsoft Research, One Microsoft Way, Redmond, WA
[1]{mrmarri, sthumma, txie}@ncsu.edu, [2]{nikolait, jhalleux}@microsoft.com

## 1. INTRODUCTION

In software development life cycle, unit testing is a key phase that helps detect defects at an early stage. In practice, it is essential to write high-covering tests during unit testing to ensure high quality of the software under development. Conventionally, a unit test do not accept any parameters and include three major parts: test scenario, test data, and test oracle. Test scenario refers to a sequence of method calls invoked by the unit test. Test data refers to concrete values passed as arguments for the method calls. Test oracle refers to an assertion statement that verifies whether the actual behavior is the same as the expected behavior.

Parameterized Unit Tests (PUTs) [18] are a new advancement in the field of software testing. Unlike conventional unit tests, PUTs accept parameters. The major advantage of PUTs compared to conventional unit tests is that developers do not need to provide test data in PUT, but only need to provide the variables that represent the test data as parameters. The concrete values for these parameters can be generated automatically using the Dynamic Symbolic Execution (DSE) [11, 9, 16] approach. Given code under test, DSE explores the code under test with random values and collects constraints along the execution path. DSE next systematically flips collected constraints and uses a constraint solver to generate concrete values that guide program execution through alternate paths. Microsoft Pex [17, 5] is an example state-of-the-art DSE based test generation tool that accepts PUTs and uses DSE to generate test data. Pex explores the provided PUTs, and generates a conventional unit test for each set of concrete values that explores a new path in the PUT or the code under test. PUTs thus provide a generic representation of conventional unit tests and are more powerful as PUTs provide a means for guided test data generation when used with an automatic testing tool such as Pex.

Although PUTs, in combination with DSE based tools such as Pex, are more beneficial than conventional unit tests, PUTs are not widely adopted by the software industry. One major reason for such a low application of PUTs in practice could be that writing PUTs is a challenging task. On the other hand, most of the software industry uses conventional unit tests as a primary means for ensuring the quality of their developed software. To leverage the benefits of PUTs and to alleviate the complexity in writing PUTs, we propose to transform the legacy conventional unit tests into PUTs. This process of transforming conventional unit tests into PUTs, referred to as *test generalization*, has three major advantages. First, using PUTs, unit tests that can achieve a high coverage of the code under test can be automatically generated. Therefore, through test generalization, new conventional unit tests can be generated that cover additional paths (in the code under test), which are not covered by the legacy conventional unit tests. Second, test generalization increases the fault detection capabilities of the test suite by appending more unit tests to the legacy unit tests. Third, test generalization helps reduce the efforts required for test code maintenance since a single PUT can represent multiple conventional unit tests. Given these benefits of test generalization, we propose a systematic procedure for assisting developers in generalizing conventional unit tests for leveraging the benefits of PUTs. To the best of our knowledge, ours is the first empirical study that proposes a systematic procedure for generalizing conventional unit tests into PUTs and also shows the benefits of test generalization.

The main objective of our systematic procedure is to take existing conventional unit tests and transform into PUTs to test the same or generalized behaviors as the ones tested by conventional unit tests. To generalize a conventional unit test, we first identify elements in the conventional unit test (such as arguments or the receiver object of the method under test) and promote these elements as PUT parameters. We then generalize the existing test oracles to fit into the PUT. During test generalization, there are two major challenges with respect to handling parameters of PUTs: (1) generating legal input values for primitive argument types and (2) generating desirable object states for non-primitive argument types. These desirable object states are the states that help explore paths in the code under test by covering `true` or `false` branches. To assist Pex or any other DSE-based tool to generate legal input values, we need to provide a sufficient number of assumptions on the parameters of PUTs. For example, consider a method under test that accepts an integer argument and includes a precondition that the argument should be greater than 1000. We add assumptions on the parameters of the PUTs to ensure that legal input values for primitive types are generated by the test generation tool. To address the second issue of generating desirable object states for non-primitive argument types, we write method-call sequences that create and mutate instances of non-primitive object types. Furthermore, there is an intrinsic challenge in generalizing test oracle and it might not be possible to fully generalize test oracles. To deal with these challenges and to ease the process of test generalization, we propose several test patterns [8] and use various supporting techniques. Furthermore, when a PUT requires desired states of the environment, external to the application, such as a file needs to exist in the file system for each generated unit test, we use mock objects to isolate

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

the execution of the generated unit tests from the environment.

We show the benefits of generalizing conventional unit tests into PUTs using our systematic procedure with three real-world applications. As we are not developers of these real-world applications, we generalize conventional unit tests of these applications as third-party developers using our systematic procedure. Our results show that test generalization increases the coverage of the code under test and also detects new defects that are not found by existing legacy unit tests. Our results also show that test generalization results in better test-code maintenance by reducing number of tests to be maintained.

In summary, the paper makes the following major contributions:

- A systematic procedure for assisting developers in generalizing existing legacy unit tests into PUTs to leverage the benefits of PUTs.
- A first-of-a-kind empirical study that investigates the benefits of PUTs over conventional unit tests. In our empirical study, we show that testing with PUTs increased branch coverage by 0% over conventional unit tests and detects 0 new defects. We also show that test generalization reduces the amount of test code, thereby, helping in better test-code maintenance. The detailed results of our empirical study are available at `http://ase.csc.ncsu.edu/projects/putstudy/`.
- Empirical results that show the benefits of supporting techniques during test generalization.
- Empirical results that show the utility of test patterns in test generalization.

The rest of the paper is structured as follows. Section 3 presents our systematic procedure designed based on our experience with the NUnit framework. Section **??** presents the categories of test patterns used in writing the PUTs. Section 4 describes how the supporting techniques are used in our test generalization. Section 6 presents the benefits of test generalization found in our study. Section **??** describes the categories of conventional unit tests that are not amenable for test generalization. Section 7 discusses the limitations of PUTs. Section 8 discusses the related work. Section 9 discusses the threats to validity. Finally, Section 10 concludes.

## 2. BACKGROUND

TODO: To add information on Pex and its industrial influence. Assume that this section introduces Pex.

## 3. SYSTEMATIC PROCEDURE

We next present our systematic procedure for generalizing conventional unit tests into PUTs using illustrative examples from the NUnit framework [1]. Our systematic procedure includes four major steps. First, we promote concrete values and other local variables in the conventional unit test as parameters for PUTs. Second, we identify the test pattern for the PUT that best matches the existing conventional unit test to be generalized. Identification of test pattern helps in easing the process of test generalization. Third, we use supporting techniques such as factory methods and mock objects to assist Pex while exploring PUTs. Fourth, we add necessary assumptions to guide Pex in generating legal values for both primitive and non-primitive types.

We next explain our systematic procedure with the help of an example from the real-world application, NUnit [1]. Figure 1 shows a method under test `SaveSetting` from the `SettingsGroup` class

```
00:public class SettingsGroup {
01:  MemorySettingsStorage storage; ...
02:  public SettingsGroup(MemorySettingsStorage storage) {
03:    this.storage = storage;
04:  }
05:  public void SaveSetting(string sn, object sv) {
06:    object ov = storage.GetSetting( sn );
07:    //Avoid change if there is no real change
08:    if (ov != null ) {
09:      if (ov is string && sv is string &&
                 (string)ov == (string)sv ||
10:        ov is int && sv is int && (int)ov == (int)sv ||
11:        ov is bool && sv is bool && (bool)ov == (bool)sv ||
12:        ov is Enum && sv is Enum && ov.Equals(sv))
13:          return;
14:    }
15:    storage.SaveSetting(sn, sv);
16:    if (Changed != null)
17:      Changed(this, new SettingsEventArgs(sn));
18:    }
19:}
```

**Figure 1: The `SettingsGroup` class of the NUnit framework with the `SaveSetting` method.**

```
00://testGroup is of type SettingsGroup
01:[Test]
02:public void TestSettingsGroup() {
03:  testGroup.SaveSetting("X", 5);
04:  testGroup.SaveSetting("NAME", "Charlie");
05:  Assert.AreEqual(5, testGroup.GetSetting("X"));
06:  Assert.AreEqual("Charlie", testGroup.GetSetting("NAME"));
07:}
```

**Figure 2: A conventional unit test to test the `SaveSetting` method (shown in Figure 1)**

of the NUnit framework. The `SaveSetting` method accepts a setting name `sn` and a setting value `sv`, and stores the setting in a storage (represented by the member variable `storage`). The setting value can be of type `int`, `bool`, `string`, or `enum`. Before storing the value, `SaveSetting` checks whether the same value already exists for that setting in the storage. If the same value already exists for that setting, `SaveSetting` returns without making any changes to the storage.

Figure 2 shows a conventional unit test for testing the `SaveSetting` method. The conventional unit test saves two setting values (of types `int` and `string`) and verifies whether the values are set properly using the `GetSetting` method. The conventional unit test verifies the expected behavior of the `SaveSetting` method only for the setting values of types `int` and `string`. The conventional unit test does not verify the behavior for types `bool` and `enum`. Furthermore, the conventional unit test does not cover the `true` branch in Statement 8 of Figure 1. The reason is that the conventional unit test does not invoke the `SaveSetting` method more than once with the same setting name. This conventional unit test achieves 0% of branch coverage in the code under test. We next explain how we use our systematic procedure to transform the conventional unit test (shown in Figure 2) into a PUT shown in Figure 3, and show how our generalized PUT can address the preceding issues.

**Promoting concrete values and local variables.** To generalize this conventional unit test, we first identify the concrete values used in the test case. For example, the unit test includes a concrete `string` value "`Charlie`" in Statement 4. We replace these concrete values with symbolic values by promoting those values as parameters for the PUT. The advantage of replacing concrete values with symbolic values is that Pex can generate concrete values based on the constraints encountered in different paths of the code

```
//PAUT: PexAssumeUnderTest
00:[PexMethod]
01:public void TestSettingsGroupPUT([PAUT] SettingsGroup st,
02:  [PAUT] string sn, [PAUT] object sv) {
03:    st.SaveSetting(sn, sv);
04:    PexAssert.AreEqual(sv, st.GetSetting(sn));
05:}
```

**Figure 3: PUT for the conventional unit test shown in Figure 2.**

```
//MSS: MemorySettingsStorage (class)
//PAUT: PexAssumeUnderTest (Pex attribute)
00:[PexFactoryMethod(typeof(MSS))]
01:public static MSS Create([PAUT]string[]
02:     sn, [PAUT]object[] sv) {
03:    PexAssume.IsTrue(sn.Length == sv.Length);
04:    PexAssume.IsTrue(sn.Length > 0);
05:    MSS mss = new MSS();
06:    for (int count = 0; count < sn.Length; count++) {
07:      mss.SaveSetting(sn[count], sv[count]);
08:    }
09:    return mss;
10:}
```

**Figure 4: An example factory method for the type `MemorySettingsStorage`.**

under test. For example, we promote the $string$ value "`Charlie`" and $int$ value `5` as a single parameter for the PUT, and we describe the type of the parameter as `object`. Pex automatically identifies the possible types for the `object` type such as `int` or `bool` from the code under test and generates concrete values for those types. Therefore, a single PUT can achieve the same test effectiveness as multiple conventional unit tests with different concrete values. In addition to promoting concrete values as parameters of PUTs, we also promote other local variables such as the receiver object of the `SaveSetting` method as parameters for PUTs. In this example, the local variable `testGroup` is a non-primitive object. Promoting such receiver objects as parameters can help generate different states (for those receiver objects) that can help cover new paths in the code under test. For example, consider that there is a defect in the `SaveSetting` method that can be exposed when there are five elements in the storage. Promoting the receiver object of `SaveSetting` as a parameter can help generate such desirable object states and expose those related defects.

**Identification of test pattern.** We next analyze the conventional unit test to identify a test pattern [2] for the PUT that best matches the existing conventional unit test. Identifying the test pattern can help in generalization of the conventional unit test as these patterns can serve as guidelines during the test generalization. In our current conventional unit test, a setting is stored in the storage using the `SaveSetting` method and is verified with the `GetSetting` method. An analysis of the conventional unit test suggests that the PUT can apply the round-trip pattern, which applies to classes such as `SettingsGroup` that has a method (such as `SaveSetting`) and an inverse method (such as `GetSetting`). Based on the identified pattern, we can find that the test oracle can include the `GetSetting` method to assert the behavior of the method under test,`SaveSetting`. Section **??** presents more details on our test patterns that are used during our generalization phase. In our empirical study, we identify that these patterns cover a broad range of conventional unit tests and assist during test generalization.

**Adding assumptions.** During test generalization, we identify that Pex requires guidance in generating legal values for PUTs. We address this challenge in our systematic procedure by adding sufficient assumptions to PUTs. For example, without any assumptions provided, Pex by default generates `null` values for non-primitive parameters of PUTs. To address this issue, we annotate a PUT parameter with a tag `PexAssumeUnderTest`[1], which describes that the parameter value should not be `null` and the parameter type should be the same as specified type. We add further assumptions based on the behavior verified by the conventional unit test. For example, if the conventional unit test requires an assumption that the setting to be added should not already exist in the storage, we add additional assumptions to the PUT.

**Using supporting techniques.** During test generalization, we identified that another major challenge is to handle parameters of non-primitive types. Pex can effectively handle primitive-type parameters such as `string` or `int`. However, for non-primitive types, method-call sequences are required for generating desirable object states. These desirable object states are the states that help explore paths in the code under test. For example, a desirable object state to cover the `true` branch of Statement 8 in Figure 1 is that the storage should already include a value for the setting name.

The primary challenge in constructing desirable states for non-primitive arguments is to construct a sequence of method calls that create and mutate objects. However, similar to the other state-of-the-art method-call sequence generation approaches [], Pex also faces challenges in generating sequences for non-primitive types such as `st`. To address this issue, we use a feature, called factory methods, to assist Pex in generating effective method-call sequences that can help achieve desirable object states. Figure 4 shows an example factory method for the `MemorySettingsStorage` type written manually. We wrote the factory method for `MemorySettingsStorage` as the member variable `storage` is of type `MemorySettingsStorage`. Our factory method accepts two arrays of setting names (`sn`) and values (`sv`) and adds these entries to the storage. This factory method helps Pex to generate method-call sequences that can create desirable object states. For example, Pex can generate five names and five values as arguments to our factory method for creating a desirable object state with five elements in the storage[2]. The same factory can be reused for all other PUTs using the `MemorySettingsStorage` type.

Along with factory methods, we use another supporting technique called mock objects. These mock objects help test features in isolation especially when the unit test interacts with environments such as file system. Section **??** provides more details on how we use these both factory methods and mock objects.

Figure 3 shows the skeleton of the PUT after generalizing concrete values and the receiver object. Our PUT accepts three parameters: an instance of `SettingsGroup`, name of the setting, and its value. The `SaveSetting` method can be used to save either an `int` value or a `string` value (the method accepts both types for its arguments). Therefore, the conventional unit test requires two method calls shown in Statements 3 and 4 of Figure 2 to verify whether the method under test correctly handles these types. On the other hand, only one method call is sufficient in the generalized PUT as the argument type is promoted to `object`. Pex automatically explores the code under test and generates tests that cover both `int` and `string` types. Indeed, the `SaveSetting` method also accepts `bool` and `enum` types. Existing conventional unit test did not include tests for verifying these two types. Our generalized PUT automatically handles these additional types, serving as a pri-

[1] `PexAssumeUnderTest` is a custom attribute provided by Pex.

[2] Note that the factory methods only provide an assistance to Pex in achieving the desirable object states, and that the desirable object states are defined by assumptions or Pex generates these object states based on the branching conditions in the code under test.

mary advantage of PUT as it helps reduce the test code significantly without reducing the behavior tested by the conventional unit test.

We next apply Pex on the `TestSettingsGroupPUT` PUT. Pex generated 8 conventional unit tests from the `TestSettingsGroupPUT` PUT. These conventional unit tests verify the `SaveSetting` method with different setting values of types such as `int` or `string` or other non-primitive object types. As described, a single PUT can substitute multiple conventional unit tests, resulting in a reduced test code maintenance. Furthermore, the conventional unit test used for generalization achieved a coverage of 0%, whereas the conventional unit tests generated from the PUT has achieved a final coverage of 0%. Although PUT achieved a higher code coverage compared to the conventional unit test, the PUT still could not cover the `true` branch of Statement 16. Developers while doing generalization can verify these uncovered portions and can either enhance PUTs or write new PUTs for achieving additional coverage of those uncovered portions[3].

# 4. SUPPORTING TECHNIQUES USED IN WRITING PUTS

We next detail on the supporting techniques used in writing PUTs. In writing PUTs, we use the factory method and mock object supporting techniques. We used an additional technique of input-space partitioning to achieve higher code coverage in our Phase 2. We next describe these techniques used in our study.

## 4.1 Factory Methods

A commonly used technique in test generalization to assist effective test generation is writing factory methods. In our study of writing PUTs and executing them with Pex, we observed that one of the primary reasons for not achieving high block coverage is due to lack of method-call sequences for achieving desirable object states. Although Pex includes a heuristic demand-driven strategy for generating method-call sequences, we found that Pex's strategy can generate method-call sequences effectively in certain limited scenarios where the constructors either accept primitive arguments or explicitly state the actual type of the argument. To address this issue, we use the factory-method feature provided by Pex. These factory methods allow developers to write method-call sequences that can help Pex in achieving desirable object states. Figure 4 shows an example factory method that we used in our study. Our factory method accepts two arrays of setting names and values, and adds those entries to the storage. This factory method helps generate different object states for `MemorySettingStorage`. For example, using our factory method, Pex can generate an object of `MemorySettingStorage` with five elements in the storage.

In our test generalization, we constructed five factory methods for assisting Pex in generating desirable object states. We observed that these factory methods are quite helpful in achieving high block coverage.

## 4.2 Mock Objects

Mock objects help test features in isolation by replacing functionalities with mock objects. In Phase 2, we added 21 new PUTs and found two methods where we needed to use mock objects. Two existing test methods `SaveEmptyConfigs` and `SaveNormalProject` involved testing saving of an `NUnitProject`. When a new project is created (i.e., an instance of `NUnitProject` is created), an `xml` file is saved in the project directory for saving the project configurations. These test methods add configurations on this project file and assert if the file is saved in the right format and contains the added configuration information. `SaveEmptyConfigs` test method tests an empty project, with default `debug` and `release` configurations and the latter tests a project saved with multiple configurations. In Phase 1, we were not able to generalize these two tests as they needed interaction with an `xml` file from a *specific location* (the specific location refers to the directory location where the project is saved; when a project is saved, a new directory is created as the project directory). The `xml` file was expected to already exist physically for the test to execute so that the code under test can access the file and add configuration information to the file, and test if the project is correctly saved with the configurations. The existing tests save the projects and assert by reading the `xml` file using a stream reader and check against an expected string (which is constructed based on the configurations that are added). Generalization of these conventional unit tests is not straightforward as every conventional unit test generated by Pex requires a physical file in the expected file location. The code under test has high dependency on external factors, i.e., file reading and writing from a specific location.

We handled this situation of not being able to generalize due to a high dependency on external environments by mocking the required `xml` file writer in Phase 2. When the code under test saves a project, it opens the `xml` file using `XmlTextWriter`. In order to avoid the complexity of generating a "real" file at a "real" file location (on creating a project) and writing into the file, we mocked the expected behaviour of `XmlTextWriter` as `MockXmlFileWriter`. This mock object, unlike the real object, appends the text to a string, preserving the output of the actual behaviour of `XmlTextWriter`. The mock object behavior results in the form of a string while the actual object would result in a file. Nevertheless, as suggested by the Pex tutorial [2], we did not mock every method of the actual class, but mocked only the methods used by the code under test. Figure 5 shows a code snippet from the mocked object.

By using the mock object technique, we were able to achieve generalization of both the existing tests of `SaveEmptyConfigs` and `SaveNormalProject`, resulting in a block coverage of 61.64% and 67.92%, respectively. Recall that we were not able to generalize the existing conventional unit tests when we did not use the mock object.

## 4.3 Input-Space Partitioning

Input-space partitioning [6] helps partition the input space into disjoint blocks, where the union of all the blocks should result in the complete input space. We explain how we performed the input-space partitioning in our study to achieve a higher coverage of the method `SaveSetting` of the class `SettingsGroup`. The `SaveSetting` method accepts an argument of type `Object`. The method accepts several types such as `int`, `string`, `bool`, and `enum`, and a different path of the code is covered for each type. Therefore, in order to achieve high code coverage, a PUT to test this method should be designed to generate conventional unit tests that take different types of the argument. For simplicity, we explain how we dealt with integers and strings only. We defined two partitions where the first partition includes integers and the second partition includes strings. We wrote separate PUTs for covering these partitions. We repeated the same procedure for other input types. Consequently, Pex achieved high coverage as it was able to generate different input types and cover several program paths.

# 5. DESIGN OF EMPIRICAL STUDY

[3]Recall that in our study, we only generalize the existing conventional unit test to PUTs to compare the benefits of PUTs over existing conventional unit tests and therefore did not write additional PUTs to achieve better coverage.

```
.........
.........
01:  public MockXmlTextWriter(string filename,
       Encoding encoding)
02:  {
03:      this.fileName = filename;
04:  }

05:  public void WriteAttributeString
     (string attributeName, string value)
06:  {
07:      xmlString = xmlString + " " + attributeName
             + "=" + "¨" + value + "¨";
08:  }

09:  public void WriteEndElement()
10:  {
11:      xmlString = xmlString + " />";
12:  }

13:  public void Close()
14:  {
15:      xmlString = xmlString.Replace("/> />", "/>" +
             System.Environment.NewLine + "</"
                 + startString + ">");
16:      CreatedProjects.currentProject = xmlString;
17:  }
............
............
```

**Figure 5: Sample code from the `MockXmlTextWriter` mock object. *In the `WriteAttributeString` method, we append the argument string to a global string `xmlString` and this global string represents the content written to the xml file.***

We conduct an empirical study using three real world applications to study the benefits of Parameterized Unit Tests (PUT) over Conventional Unit Tests (CUT). In our empirical study, we show the benefits of PUTs over existing CUTs in terms of three factors: code coverage, defects, and complexity of test code. In particular, we address the following three research questions through our empirical study:

- **RQ1.** How much higher percentage of code coverage is achieved by PUTs compared to existing CUTs? Since PUTs are a generalized form of CUTs, this research question helps to address whether PUTs can achieve additional coverage compared to CUTs.
- **RQ2.** How many new defects (that are not detected by CUTs) are detected by PUTs and vice-versa? This research question helps to address whether PUTs have more fault-detection capabilities compared to CUTs.
- **RQ3.** How many number of tests are reduced by generalizing CUTs to PUTs? This research question helps to address whether PUTs require lesser efforts for maintaining test code compared to CUTs.

## 5.1 Coverage

We next describe our design of our experiment for addressing RQ1. We execute the existing test suite (CUTs) and measure the coverage achieved for each class under test. We then execute the PUTs, i.e., the generalized CUTs and measure the coverage achieved for each class under test by these PUTs. We then compare the code coverage achieved by both CUTs and PUTs for each class under test. We use the Pex coverage report to measure the dynamic coverage achieved by both CUTs and PUTs.

## 5.2 Defects

**Table 1: Mapping of number of CUTs and PUTs.**

| CUT | PUT | # of occurrences |
|---|---|---|
| 1 | 1 | 129 |
| 1 | 2 | 2 |
| 1 | 3 | 2 |
| 2 | 1 | 29 |
| 3 | 1 | 14 |
| 4 | 1 | 15 |
| 5 | 1 | 4 |
| 6 | 1 | 4 |
| 7 | 1 | 1 |
| 8 | 1 | 2 |
| 9 | 1 | 1 |
| 15 | 1 | 1 |

To address RQ2, we conduct two evaluations. In the first evaluation, we identify the number of real defects detected by PUTs. Since we are using the CUTs that are available with the sources, we did not find any failing CUTs, i.e., no defects are reported by the existing test suite. Therefore, any failing unit tests resulting when testing using PUTs are considered as defects that are not found by the CUTs. However, before confirming the failing test case implies a defect in the code under test, we manually verify that the generated test inputs are valid and that it is not a false positive due to an ineffective PUT.

We conduct second evaluation based on mutation testing to further show the effectiveness of PUTs in detecting defects compared to CUTs. The reason for the second evaluation is that the existing CUTs do not detect any defects in the code under test. In this evaluation, we seed defects in the code under test using a mutation testing tool and verify how many mutants are killed by existing CUTs and PUTs. We consider that a mutant is killed if any previously passing test fails after executed with the seeded fault.

We use the following five basic mutation operators, recommended by Offutt, for seeding defects in the code under test.

- ABS: Forces each arithmetic expression to take values zero, positive and negative values.
- AOR: Replaces each arithmetic operator with every syntactically legal operator.
- LCR: Replaces logical connector (AND and OR) with other kinds of logical operators.
- ROR: Replaces each relational operator with other relational operators.
- UOI: Inserts unary operators in front of expressions.

## 5.3 Maintenance of Test Code

We next address the third research question of whether test generalization can reduce the efforts in maintaining test code. We use two metrics to address this research question. First, we compare the number of CUTs and the number of PUTs. The higher the difference between the number of CUTs and the number of PUTs, the lesser are the efforts required in maintaining the test code. The reason is that whenever the code under test is modified, all failing tests need to be modified based on the new expected behavior of the code under test. Therefore, a low of number of PUTs can significantly reduce the efforts in maintaining the test code since only a few PUTs need to be modified. Second, we compare the Lines of Code (LOC) of CUTs and PUTs. The reason for the second metric is that a low number of PUTs with a high amount of LOC does not help in reducing the efforts required in maintaining the test code.

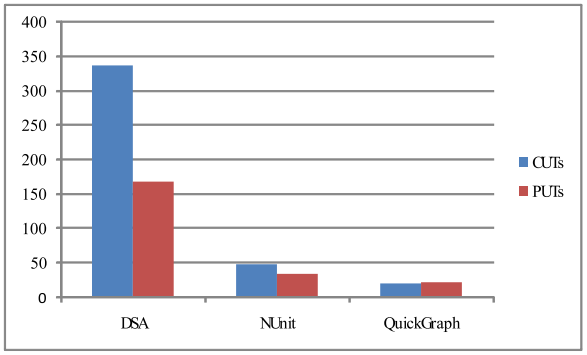


**Figure 6: Comparison of the number of CUTs and PUTs**

```
public void AddFirstTest([PAUT]SinglyLinkedList<int> sll,
          [PAUT]int[] ne) {
    PexAssume.IsTrue(ne.Length > 1);
    PexAssume.IsTrue(sll.Count == 0);
    for (int i = 0; i < ne.Length; i++)
        sll.AddFirst(ne[i]);
    PexAssert.AreEqual(ne[ne.Length - 1], sll.Head.Value);
    PexAssert.AreEqual(ne[0], sll.Tail.Value);
    PexAssert.AreEqual(ne.Length, sll.Count);
}
```

**Figure 7: The `SettingsGroup` class of the NUnit framework with the `SaveSetting` method.**

Figure 6 shows the comparison of the number of conventional unit tests and PUTs for all subject applications. The x-axis shows the subject application and y-axis shows the number of CUTs or PUTs. For all the three subject applications, 407 CUTs are transformed into 224 PUTs. The figure shows that, except for QuickGraph, there is a significant reduction in the number of tests for the subjects DSA and NUnit. The results show the test generalization can help in reducing the efforts of maintaining test code. Table 1 shows further relations between the number of CUTs and PUTs. For example, Row 1 of Table 1 shows that one CUT is transformed to one PUT in 129 occurrences. The last row show that in one occurrence, 15 CUTs are transformed into a single PUT. Rows 2 and 3 show exceptional cases where a CUT is transformed into multiple PUTs. Section **??** discusses more about these exceptional cases. Figure 7 shows an example PUT, which is a result of transforming four CUTs of the `SinglyLinkedList` class of DSA. The purpose of these four CUTs is to verify the behavior of the `AddFirst` method. The four CUTs verify different behaviors such as adding one element or two elements to the list and verifies whether the `head` and `tail` values include the correct values. The CUTs also verify whether the list contains the same number of added elements. We transformed all these CUTs into a single PUT shown in Figure 7. The previous CUTs verify by adding only a fixed number of elements with fixed values to the list, whereas our transformed PUT, along with combining all four CUTs, can verify the behavior of `AddFirst` for variable number of elements in the list.

We next show the results of our second metric related to the lines of code for CUTs and PUTs. Figure 8 shows the comparison of lines of code for CUTs and PUTs. For DSA, PUTs have lesser lines of code compared to CUTs, whereas for the other two subjects, the PUTs have a few additional lines of code compared to CUTs. Overall, PUTs have almost the similar lines of code as CUTs.

# 6. BENEFITS OF PUTS

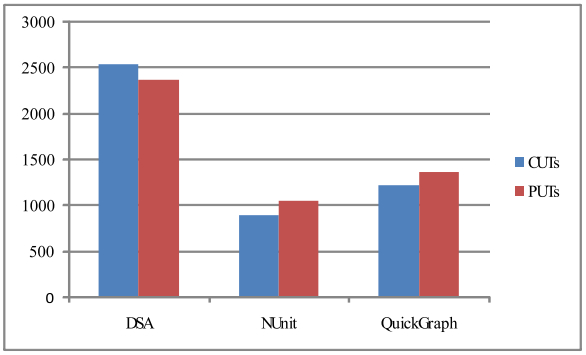


**Figure 8: Comparison of Lines of Code of CUTs and PUTs**

We next present the benefits of retrofitting conventional unit tests for PUTs. In test generalization, we transformed 57 conventional unit tests of 10 test classes resulting in 49 PUTs. In order to achieve higher block coverage, we wrote 21 new PUTs for 6 of the 10 classes under study. Table 2 shows the results of writing PUTs. Column "Test Class" shows the names of the test class and Column "Test Methods" shows the statistics of existing conventional test methods, the test methods that are amenable to test generalization, the percentage of amenable conventional unit tests, the number of transformed PUTs, and the number of new PUTs that were written. The five sub-columns represented by "Conventional", "#Amenable", "%", "#PUT", and "#New PUT" give cumulative figures for all test methods in the corresponding class. "#PUT" shows the number of PUTs written in Phase 1 and "#New PUT" shows the number of new PUTs written in Phase 2 to achieve higher coverage. Section **??** provides details of the test methods that are not amenable to test generalization with illustrative examples. Column "% Coverage" shows the block coverage reported by Pex on executing tests. These two sub-columns show the average of all the test methods in each class. Column "Avg. New Blocks" shows the average number of new blocks covered by PUTs. As PUTs can verify more general behavior, we found that PUTs achieve a high block coverage and cover new blocks that are not covered by conventional unit tests. Column "#Defects" shows the number of defects that were detected by the PUTs and were not detected by the existing conventional unit tests. This column again shows a cumulative value of all the test methods in a test class.

The benefits discussed here primarily reflect the results of the test generalization phase as we compare the benefits of PUTs over the existing conventional unit tests. Section 7 discusses the amount of effort we took in writing the new PUTs in comparison to generalizing the existing conventional unit tests. We found three major benefits of retrofitting conventional unit tests for PUTs: higher code coverage, detection of new defects, and reduced test code. We observed that generalization increases code coverage and detected new defects that were not detected by the existing conventional unit tests. We also identified that a single PUT often helps to replace multiple conventional unit tests, thereby reducing the amount of test code (as shown by Column "#Convention" and Column "#PUT" in Table 2). Sections 6.1, 6.2, and 6.3 explain these details.

## 6.1 Coverage

As generalized tests often help cover more scenarios, we found that test generalization helped to have an effective increase in the block coverage as shown in Table 2. For example, test generaliza-

**Table 2: Benefits of Retrofitting PUTs in Unit Testing**

| Test Class | Test Methods | | | | | % Coverage | | | Avg. New Blocks | #Defects |
|---|---|---|---|---|---|---|---|---|---|---|
| | #Conventional | #Amenable | % | #PUT | #New PUT | Conventional | PUT | With New PUT | | |
| NUnitProjectSave.cs | 3 | 1 | 33.33 | 1 | 2 | 35.71 | 40.98 | 57.91 | 10 | 0 |
| NUnitRegistryTests.cs | 5 | 5 | 100.00 | 4 | | 58.97 | 72.80 | 100.00 | 0 | 1 |
| TestAgentTests.cs | 2 | 2 | 100.00 | 2 | 1 | 100.00 | 100.00 | NA | 0 | 0 |
| RegistrySettings-StorageTests.cs | 6 | 5 | 83.33 | 6 | 4 | 45.34 | 90.60 | 100.00 | 0 | 1 |
| MemorySettings-StorageTests.cs | 6 | 4 | 66.67 | 4 | | 100.00 | 100.00 | NA | 2 | 0 |
| PathUtilTests.cs | 7 | 3 | 42.86 | 6 | | 85.00 | 85.00 | NA | 0 | 3 |
| RecentFilesTests.cs | 22 | 21 | 95.45 | 5 | 6 | 59.94 | 76.08 | 84.17 | 14 | 0 |
| ServerUtilityTests.cs | 2 | 2 | 100.00 | 3 | 1 | 90.32 | 90.32 | 95.16 | 0 | 2 |
| SettingsGroupTests.cs | 5 | 5 | 100.00 | 6 | 7 | 66.94 | 90.95 | 94.13 | 2 | 0 |
| ProcessRunnerTests.cs | 0 | NA | NA | NA | NA | NA | NA | NA | NA | NA |

tion of the `RegistrySettingsStorage` class shows an increase in the coverage of 45.24%. In addition, the tests generated for the PUTs in the test generalization achieved coverage of new blocks that are not covered by the existing conventional unit tests. In order to achieve more code coverage, we wrote 21 new PUTs for 6 classes and obtained an average increase of 35% code coverage (considering only those classes for which we wrote new PUTs).

## 6.2 Defects

After test generalization we found 7 new defects that were not detected by the existing conventional unit tests. We next explain a defect detected by our test generalization. The `NUnitRegistry` class stores the `RegistryKeys` in a tree-structured hierarchy. For building the key hierarchy, a default key is taken as a main key and the given keys are added as sub-keys to the main key or to the other sub-keys. During testing, adding a key hierarchy and on checking for the count or on clearing the keys, we found abnormal behavior for two tests. The PUT was written to take three test inputs. For one of the tests generated by Pex, the test inputs were `t`, `t`, and `t`, and the other test case took the test inputs as `\0`, `\0`, and `\0`. For the first test case, when the three inputs were added to a main key (two as sub-keys and the other as a sub-key to one of the added subkeys), the count check for the keys passed, i.e., `PexAssert(2, mainKey.SubKeyCount)` passed. The same assertion failed for the second case (with test inputs "\0") with an assertion failure message, "expected 2, got 1". This defect shows that the failure was possibly due to missing check on invalid characters.

## 6.3 Test Code

Test generalization also helped reduce the test code as shown in Column "#PUT" of Table 2. Often, test generalization either helps reduce the amount of code in a single test method or helps combine multiple test methods into a single PUT. Figure 9 shows an example PUT of the pattern type *Cases* that combined five conventional unit tests. Each conventional unit test verifies one case in the corresponding PUT. In addition, the PUT achieved higher coverage compared to the five conventional unit tests as the `MaxValue` is now accepted as an argument and the concrete values are generated from the argument based on captured constraints.

# 7. DISCUSSION

```
[PexMethod]
public void CountOverOrAtMaxPUT1(int MaxValue) {
  recentFiles.MaxFiles = MaxValue;
  PexAssert
    .Case(MaxValue < MIN)
      .Implies(() => MIN == recentFiles.MaxFiles)
    .Case(MaxValue == MIN)
      .Implies(() => MaxValue == recentFiles.MaxFiles)
    .Case(MaxValue > MIN && MaxValue < MAX)
      .Implies(() => MaxValue == recentFiles.MaxFiles)
    .Case(MaxValue == MAX)
      .Implies(() => MaxValue == recentFiles.MaxFiles)
    .Case(MaxValue > MAX)
      .Implies(() => MAX == recentFiles.MaxFiles);
}
```

**Figure 9: Single PUT constructed from five conventional tests.**

One of the major limitations of PUTs is that PUTs requires more effort from developers than writing conventional unit tests requires. Although PUTs reduce the complexity of writing multiple conventional unit tests with various concrete test inputs, developers need additional expertise in writing such PUTs as PUTs are more generic compared to conventional unit tests. For example, writing PUTs requires developers to prescribe a test oracle that can deal with the generality of test inputs. We show that to reduce the complexity of writing PUTs, we adopted a methodology of writing PUTs in two phases. In Phase 1, we generalized the existing conventional unit tests to PUTs using suggested test patterns. In Phase 2, we used supporting techniques to write more PUTs to assist Pex in generating high-covering tests.

In our study, we observed that we took longer time to write PUTs in Phase 2 compared to the time we took in Phase 1. In Phase 1, we transformed the existing conventional unit tests to PUTs and the existing unit tests assist in writing PUTs as shown in Section **??** with an example. In Phase 2, we discovered the un-covered code portions and wrote more PUTs to assist Pex to generate tests to cover the un-covered code portions. Based on our experience, we believe that to enjoy the test effectiveness achieved by writing PUTs and to reduce the cost involved in writing PUTs, a practical solution could be retrofitting PUTs by transforming these conventional unit tests to PUTs. We expect that writing a single conventional unit test to test a method under test and then transforming it to a PUT can help developers in writing the PUTs. Such a single conventional unit test

can act as an *example* unit test representing the intention of "what" needs to be tested. We expect that this methodology can both ease the process of writing PUT and still achieve high test effectiveness in unit testing.

As shown in our study, although the usage of the suggested test patterns and the supporting techniques can reduce effort of developers in writing PUTs and allow the test generation tool to generate high-covering tests, the complexity lies in being able to use them. In general, developers might consider it a tougher job to learn the supporting techniques and use them to write PUTs than writing multiple possible conventional unit tests. Nevertheless, our study shows that PUTs are more effective than conventional unit tests in detecting defects and also in achieving high code coverage. Therefore, we believe that the choice of writing PUTs is a trade-off between cost and benefit.

# 8. RELATED WORK

Pex [17, 18, 19] accepts PUTs and uses symbolic execution to generate test inputs. Similarly, other existing tools such as Parasoft Jtest [3] and CodeProAnalytiX [4] adopt the design-by-contract approach [13] and allow developers to specify method preconditions, postconditions, and class invariants for the unit under test and carry out symbolic execution or random testing to generate test inputs. More recently, Saff et al. [15] propose theory-based testing and generalize six Java applications to show that the proposed theory-based testing is more effective compared to traditional example-based testing. A theory is a partial specification of a program behavior and is a generic form of test methods where assertions should hold for all inputs that satisfy the assumptions specified in the test methods. A theory is similar to a PUT and Saff et al.'s approach uses these defined theories and applies the constraint solving mechanism based on path coverage to generate test inputs similar to Pex. The results of their study show a $25\%$ of test methods amenable to test generalization compared to the $80.18\%$ of test methods amenable to test generalization achieved in Phase 1 of our empirical study (calculated by the average of the Column "% amenable" in Table 2). Furthermore, their study does not provide the methodology of test generalization or show empirical evidence of benefits of test generalization shown in our study.

There are existing approaches [14, 7, 10] that can automatically generate required method-call sequences that achieve different object states. However, in practice, each approach has its own limitations. For example, Pacheco et al.'s approach [14] generates method-call sequences randomly by incorporating feedback from already generated method-call sequences. However, such a random approach can still face challenges in generating desirable method-call sequences, as often there is little chance of generating required sequences at random. In our test generalization, we manually write factory methods to assist Pex in generating desirable object states for non-primitive data types.

In our previous work [12], we presented an empirical study to analyze the use of parameterized model in unit testing with PUTs. We showed that using a mock object can ease the process of unit testing and identified challenges faced in testing code when there are multiple APIs that need to be mocked. In our current study, we also use mock objects in our testing with PUTs. However, our previous study showed the benefits of mock objects in unit testing, while our current study shows the use of mock objects to help achieve test generalization. In another of our previous work around PUTs [20], we propose mutation analysis to help developers in identifying likely locations in PUTs that can be improved to make more general PUTs. In contrast, our current study suggests a methodology of retrofitting conventional unit tests for parameterized unit testing.

# 9. THREATS TO VALIDITY

The threats to external validity primarily include the degree to which the subject programs, faults, and conventional unit tests are representative of true practice. The number of classes used in our empirical study is relatively small, although the defects detected during our study are real defects. These threats could be reduced by conducting more studies with wider types of subjects in our future work. The threats to internal validity are due to manual process involved in writing PUTs. Our study results can be biased based on our experience and knowledge of the subject programs. These threats can be reduced by conducting more case studies with more subject programs and other human subjects. The results in our study can also vary based on other factors such as the effectiveness of our written PUTs and test generation capability of Pex. One threat to construct validity is that our study uses the block coverage reports generated by Pex to reflect the quality of tests.

# 10. CONCLUSION

We conducted an empirical study to investigate the utility of PUTs in unit testing. We first generalize the existing conventional tests by transforming conventional unit tests into PUTs, and then write new PUTs to increase code coverage. In Phase 1 of our study, we generalized $57$ conventional unit tests in the NUnit framework to write $49$ PUTs. We identified benefits of test generalization such as increase in the block coverage by $9.68\%$ (on average) with a maximum increase of $45.26\%$ for one class under test and detection of 7 new defects. In Phase 2 of our study, we wrote $21$ new PUTs and achieved an increase in the block coverage of $17.41\%$ over the conventional unit tests. We also identified types of conventional tests that are not amenable for test generalization and proposed new PUT patterns. We present details on the utility of suggested test patterns and supporting techniques that help in writing PUTs. We further discuss the limitations of retrofitting unit tests for PUTs in terms of effort required in writing these PUTs.

# 11. REFERENCES


[1] NUnit, 2002. `http://nunit.com/index.php`.

[2] Pex Documentation, 2006. `http://research.microsoft.com/Pex/documentation.aspx`.

[3] Parasoft Jtest, 2008. `http://www.parasoft.com/jsp/products/home.jsp?product=Jtest`.

[4] CodePro AnalytiX, 2009. `http://www.eclipse-plugins.info/eclipse/plugin_details.jsp?id=943`.

[5] Pex - Automated White box Testing for .NET, 2009. `http://research.microsoft.com/Pex/`.

[6] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[7] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software Practice and Experience*, 34(11), 2004.

[8] J. de Halleux and N. Tillmann. Parameterized Test Patterns For Effective Testing with Pex, 2008. `http://research.microsoft.com/en-us/projects/pex/pexpatterns.pdf`.

[9] 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 (PLDI)*, pages 213–223, 2005.

[10] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 553–568, 2003.

[11] J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.

[12] M. R. Marri, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. An empirical study of testing file-system-dependent software with mock objects. In *Proceedings of the 2009 International Workshop on Automation of Software Test (AST), Business and Industry Case Studies (to appear)*, 2009.

[13] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, 2000.

[14] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proceedings of the 2007 International Conference on Software Engineering (ICSE)*, pages 75–84, 2007.

[15] D. Saff, M. Boshernitsan, and M. D. Ernst. Theories in Practice: Easy-to-write Specifications that Catch Bugs,. Technical report.

[16] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic Unit Testing Engine for C. In *Proceedings of the 2005 joint meetings of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 263–272, 2005.

[17] N. Tillmann and J. de Halleux. Pex - White Box Test Generation for .NET. In *Proceedings of the 2008 International Conference on Tests and Proofs (TAP)*, pages 134–153, 2008.

[18] N. Tillmann and W. Schulte. Parameterized Unit Tests. In *Proceedings of the 2005 joint meetings of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 253–262, 2005.

[19] N. Tillmann and W. Schulte. Unit Tests Reloaded: Parameterized Unit Testing with Symbolic Execution. *IEEE Software*, 23(4):38–47, 2006.

[20] T. Xie, N. Tillmann, P. de Halleux, and W. Schulte. Mutation analysis of parameterized unit tests. In *Proceedings of the 2009 International Workshop on Mutation Analysis (Mutation) (to appear)*, 2009.