An Empirical Study of 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

Abstract

Owing to the significance of unit testing in software development life cycle, unit testing has been widely adopted by the software industry to ensure high quality software. It is a manually cumbersome job to write high-covering test cases to ensure the quality and therefore calls for automation. One such automatic testing technique to achieve high-covering tests is to write Parameterized Unit Tests (PUTs) and use them in combination with a test generation tool that accept these PUTs. PUTs are generalized form of conventional unit tests that accept parameters and where programmers can describe the expected behavior or specifications in a generic manner. Pex is an automatic test generation tool that accepts these PUTs and generates a set of conventional unit tests to achieve a high code coverage. We conduct an empirical study to show the utility of Parameterized Unit Tests (PUTs) to generate high-covering conventional unit tests. In our empirical study, we used an open source C# project called NUnit to study the benefits of PUTs over conventional unit tests. In this paper, we present our empirical study where we carried out the process of writing PUTs in two phases: test generalization (transforming existing conventional unit tests to PUTs) and writing new PUTs to cover the code under test that is not covered by the transformed PUTs. In our study, we found that test generalization increased the average code coverage by 9.68% *and writing new PUTs resulted in a total increase of* 17.41%. *We also found that testing with PUTs detectected* 7 *new defects that were nit detected by the existing conventional unit tests.*

1. Introduction

Unit testing is a key phase of the software development life cycle that helps detect defects at an early stage and ensures quality of the developed code. It is essential to write high-covering test cases during the unit testing phase to ensure high quality of the software being developed. The conventional unit tests do not accept any parameters and include two major parts: test inputs and test oracles. In general, test inputs include the method calls that modify the state of the object and arguments (concrete values) of these methods, whereas test oracles include method calls that verify the modified state. Often these conventional unit tests are written manually. However, writing all unit tests manually to achieve high coverage is a prohibitively expensive task.

Parameterized Unit Tests (PUTs) [14] are a new advancement in unit testing that accept parameters unlike conventional unit tests, which do not accept any parameters. In particular, PUTs represent expected behavior or specifications of the code under test with symbolic values. Pex [13] is a Microsoft test generation tool that accepts these PUTs and generates a set of conventional unit tests that try to achieve a high block coverage of the code under test. Pex uses an approach called dynamic symbolic execution [9, 8, 12] for generating conventional unit tests from PUTs. Initially, Pex explores the program with random values and collects constraints along the path executed in the code under test. Consequently, Pex explores alternate paths in the code under test by systematically flipping the captured constraints and generates concrete values (using a constraint solver) that can cover the alternate paths. For each set of concrete values that explore a new path, Pex generates a conventional unit test case. Therefore, each PUT is instantiated a number of times to generate a set of conventional unit test to achieve high coverage of the code under test [15].

Although PUTs are more effective and can often achieve higher block coverage compared to conventional unit tests, it is a non-trivial task to write PUTs. Writing PUTs requires more efforts compared to writing conventional unit tests. To address this challenge, we use the categorized PUT patterns [7] and use various helper techniques to ease the process of writing PUTs. To show the utility of PUTs over conventional unit tests in our empirical study and to ease the process of writing PUTs, we carry out a systematic procedure including two phases 1) *test generalization* and 2) writing additional PUTs. Test generalization[1] is defined as the process of transforming a conventional unit test into a PUT [15]. In the test generalization phase we take existing conventional unit tests and write PUTs to test the same unit (or method) as the conventional unit test. To transform the conventional unit tests to PUTs, we first

[1]We refer to unit test generalization as test generalization through out the paper.

identify the members of the conventional unit test such as arguments or the receiver object of the method under test and promote those members as method arguments. We next transform the assertions provided by the conventional unit tests into `PexAssert`[2]. We execute the PUTs and record the coverage information reported by Pex. In the writing new PUTs phase we identify the code portions that are not covered by the transformed PUTs and write new PUTs. The goal of the second phase is to allow Pex to generate more test cases to cover the un-covered code portions. We define new scenarios and use techniques such as input-space partitioning in order to write new PUTs for covering the code portions that are not covered previously.

We observe that there can be two major challenges in test generalization with respect to the test input values: 1) generating legal input values 2) generating non-primitive object states. To assist the test generation tool to generate legal input values, a sufficient number of preconditions need to be specified. For example, a method under test accepting an `integer` can have precondition that the argument should be greater than 1000. In our test generalization with PUTs, we add such preconditions (required by the method under test) using `PexAssume` supported by Pex[3] in a PUT. To deal with the issue of non-primitive object states, we use the technique of factory method when writing PUTs. For example, if the method under test requires a non-primitive argument, Pex cannot effectively generate a method-call sequence that can produce desirable object state for the non-primitive argument. These desirable object states are the states that help explore several paths in the method under test by covering `true` or `false` branches in branching points. To address this issue, we use the factory method feature provided by Pex to assist it in generating desirable object states for non-primitive PUT arguments. We also use several proposed test patterns [7] when writing the PUTs.

In our study, we used the NUnit framework [1], which is a widely used open source unit-testing framework for the C# language analogous to JUnit [4] for Java. We found that the block coverage achieved by transforming the conventional unit tests to PUTs has increased by 9.68% (on average) with a maximum increase of 45.26% for one class under test. We also observed a further increase of 7.74% block coverage with the new PUTs. Therefore, the total increase in the block coverage achieved by PUTs over conventional unit tests is 17.41%. We also found that test generalization helped cover 80 new blocks and detected 7 new defects.

This paper makes the following contributions:

- A first empirical study that shows the benefits of PUTs over conventional unit tests. In our empirical study, we show that test generalization increases block coverage by 9.68% over conventional unit tests and detects seven new defects. We also show evidence that test generalization reduces the amount of test code, thereby, helping in better maintenance.

- A systematic procedure of retrofitting conventional unit tests for parameterized unit testing to reduce the efforts in writing PUTs. Although writing PUTs requires more effort compared to writing conventional unit tests, our systematic procedure helps reduce these additional efforts.

[2]`PexAssert` is a static helper class provided by Pex to evaluate assertions [2]

[3]`PexAssume` is a static helper class provided by Pex to filter inputs [2]

Attribute	**Value**
#Total Files	560
#Files in NUnit.Util	72
Total LOC	53K
NUnit.Util LOC	7.2K
# Test Files of NUnit.Util	32

Table 1. Code metrics of project used in the study (NUnit)

- A study that shows the benefits of three helper techniques in writing effective PUTs.

- A study that shows the utility of test patterns during test generalization and writing new PUTs. We also propose two new test patterns that are useful to support to enable easy writing of the PUTs.

The rest of the paper is structured as follows. Section 2 describes the characteristics of the NUnit framework. Section 3 describes the methodology followed in our study. Section 4 presents an example from the NUnit framework and explains the procedure for test generalization. Section 5 presents the categories of test patterns used in writing the PUTs. Section 6 describes how the helper techniques are used in our test generalization. Section 7 presents the benefits of test generalization found in our study. Section 8 describes the categories of tests that are not amenable for test generalization. Section 9 discusses the limitations of PUTs. Section 10 discusses the related work. Section 11 discusses the threats to validity and finally, Section 12 concludes.

2. Open Source Project Under Test

NUnit is a widely used open source unit-testing framework for all .NET languages analogous to JUnit for Java [1]. NUnit is written in C# and uses attribute based programming model [3] through a variety of attributes such as `[TestFixture]` and `[Test]`. The rationale behind choosing NUnit for our test generalization is the large number of unit tests available with the project. These unit tests also provide information about the runtime behavior of the system. The source code of the entire project includes 560 files and about 53 KLOC. The test code includes 264 source files with 25 KLOC. This significant amount of test code makes this project a good subject for our empirical study. For the purpose of the study, we chose the Util package (nunit.util.dll), which is one of the core components of the framework.

The Util package includes 7.2 KLOC with 72 files and 326 methods. The test files, test methods, and test LOC account to 32, 335, and 3.4 KLOC, respectively. The reason for choosing the `Util` namespace[4] for the study is two fold 1) its significance in probably being one of the first modules to be developed 2) t chose an independent module to reduce the amount of work required to facilitate unit testing by avoiding to control the modules on which the module used in the study depends.

3. Methodology Used in the Study

[4]Namespace is C# is equivalent to package in Java.

```
//st is of type MemorySettingsStorage and
//instantiated in the init() method of the test class
01:public void SaveAndLoadSettings() {
02:  Assert.IsNull(st.GetSetting("X"));
03:  Assert.IsNull(st.GetSetting("NAME"));
04:  st.SaveSetting("X", 5);
05:  st.SaveSetting("NAME", "Charlie");
06:  Assert.AreEqual(5, st.GetSetting("X"));
07:  Assert.AreEqual("Charlie", st.GetSetting("NAME"));
08:}
```

Figure 1. A conventional unit test from the `Util` project of the NUnit framework.

```
00:[PexMethod]
01:public void SaveAndLoadSettingsTest1(
02:  MemorySettingsStorage st, String sn, Object sv) {
03:    //Define Pex Assumptions
04:    st.SaveSetting(sn, sv);
05:    PexAssert.AreEqual(sv, st.GetSetting(sn));
06:}
```

Figure 2. PUT skeleton for the conventional unit test.

Our methodology of writing PUTs in our study can be classified into two major phases: 1) Test generalization 2) Writing additional PUTs to achieve high coverage. The first phase involves transforming existing conventional unit tests into PUTs, where both conventional unit tests and PUTs attempt to verify the same behavior. The second phase involves discovering the code portions that were not covered with the PUTs in Phase 1 and write new PUTs to achieve high coverage. We next explain the systematic procedure used in Phase 1 and Phase 2.

Phase 1 - Test Generalization. In Phase 1, for each conventional unit test, we identify the possible local variables that can be considered as parameters and replace those local variables with parameters. We next identify the test pattern to which the conventional unit test belongs. Identification of the test pattern for the PUT in advance helps to assist in writing PUTs effectively. We also define new patterns, if the conventional unit test does not fall into any of the existing test patterns. We next add necessary assumptions to make sure that the PUT tests the same expected behavior of the conventional unit test. We consequently add assertions to validate the actual behaviour of the code under test against the expected behavior. In our test generalization phase, we use the factory method helper technique supported by Pex to achieve successful generalization. When there are any non-primitive parameters, Pex may not be able to generate method-call sequences to construct desirable object states for those non-primitive parameters. These desirable object states are the states that help explore several paths in the method under test by covering `true` or `false` branches in branching points. We assist Pex by providing factory methods to achieve desirable object states for non-primitive parameters.

Phase 2 - Writing new PUTs. In Phase 2, our objective is to modify existing PUTs or write new PUTs to achieve a higher code coverage of the classes under test. In particular, we observe the detailed coverage reports generated by Pex to identify un-covered code portions. We identify techniques to extend existing PUTs to cover those un-covered portions. In our study, we used three two techniques to write the additional PUTs: mock objects and input-space partitioning. We explain these helper techniques in detail in Section 6.

4. Example

We next explain the systematic procedure of retrofitting PUTs to carry out unit testing in our empirical study with an example. We use the NUnit test case `SaveAndLoadSettings` shown in Figure 1 as an illustrative example for explaining our procedure. The objective of the unit test is to verify the behavior of the class `MemorySettingsStorage`, which is primarily used for storage and retrieval of global values. To generalize the current unit test, we first identify the concrete values used in the test case. For example, the unit test includes a concrete value "`X`". We replace these concrete values with symbolic values by making them as arguments. The advantage of replacing these concrete values with symbolic values is that Pex can generate concrete values based on the constraints encountered in different paths of the method under test (MUT). Consequently, a single PUT can achieve the same test effectiveness (of high block coverage of MUT) as multiple conventional unit tests with different concrete values testing the same method. In addition to generalizing the concrete values, we also generalize the receiver object.

We then analyze the conventional unit test to identify the PUT pattern [2] the test belongs to. Identifying the pattern can help in easy generalization of the conventional unit test. For example, in the current conventional unit test, a setting is stored in the storage using the `SaveSetting` method and is verified with the `GetSetting` method. This scenario belongs to the round-trip pattern suggested in the Pex documentation. If the conventional unit test does not fall into any of the pre-defined categories, we define new patterns as shown in Section 5.2.

Figure 2 shows the skeleton of the PUT after generalizing concrete values and the receiver object. Our PUT accepts three parameters: instance of `MemorySettingsStorage`, name of the setting, and its value. The `SaveSetting` method can be used to save either an `integer` value or a `string` value (the method accepts both types for the argument). Therefore, the test requires two method calls shown in Statements 4 and 5 (Figure 1) to test the method under test works as expected for both the input value types. However, we need only one method call of `SaveSetting` in the PUT because we accept the value type as `Object`, which can cover both `integer` and `String`. Indeed, the `SaveSetting` method also accepts `bool` and `enum` types. The generalization can automatically handle these additional types too; serving as a primary advantage of PUT as it helps reduce the test code significantly without reducing the behavior tested by the conventional unit test. We transform the assertions in the conventional unit test into `PexAssert` to assert the same behavior. If the existing set of assertions are not sufficient, we add additional assertions to the PUT. Figure 4 shows the transformed PUT that can replace the conventional unit test.

For test input values generation, Pex can effectively handle primitive-type parameters such as `String` or `integer`. However, as would any test generation tool, Pex faces challenges in generating values for non-primitive arguments such as `st` in our PUT skeleton. These non-primitive arguments often require desirable

```
[PexFactoryMethod(typeof(MSS))]
     //MSS: MemorySettingsStorage (class)
     //PAUT: PexAssumeUnderTest (Pex attribute)
public static MSS Create([PAUT]String[]
 sn, [PexAssumeNotNull]Object[] sv) {
   PexAssume.IsTrue(sn.Length == sv.Length);
   PexAssume.IsTrue(sn.Length > 0);
   MSS mss = new MSS();
   for (int count = 0; count < sn.Length; count++) {
   PexAssume.IsTrue(sv[count] is string || sv[count]
     is int || sv[count] is bool || sv[count] is Enum);
     mss.SaveSetting(sn[count], sv[count]);
   }
   return mss;
}
```

Figure 3. A factory method to assist Pex.

```
00:[PexMethod]
     //MSS: MemorySettingsStorage (class)
     //PAUT: PexAssumeUnderTest (Pex attribute)
01:public void SaveAndLoadSettingsPUT1([PAUT]
02:  MSS st, [PAUT]String sn, [PAUT]Object sv) {
03:    PexAssume.IsFalse(sn.Equals(""));
04:    PexAssume.IsTrue(st.GetSetting(sn) == null);
05:    storage1.SaveSetting(sn, sv);
06:    PexAssert.AreEqual(sv, st.GetSetting(sn));
07:}
```

Figure 4. Complete PUT for the conventional unit test.

```
[PexMethod]
//MSS: MemorySettingsStorage (class)
//PAUT: PexAssumeUnderTest (Pex attribute)
//Changed is of type Delegate
public void RemoveSetting(MSS st,
     [PAUT]string settingName) {
  st.RemoveSetting( settingName );
  if (Changed != null)
    Changed(this, new SettingsEventArgs(settingName));
}
```

Figure 5. A code sample with an un-covered portion with PUTs written by transforming conventional unit tests.

states to verify different behaviors. For example, an intention in our conventional unit test to have two `SaveSetting` method calls is to verify adding a new setting when there is already an existing setting in the storage. For example, consider that there is a defect in the implementation of `SaveSetting` that can be exposed *only* when there are five elements in the storage, then the desirable state for such a non-primitive argument is to have five elements already present in the storage. Therefore, to test the method under test in various scenarios, generalizing the receiver object helps in this case. However, the primary challenge in constructing desirable states for non-primitive arguments is to construct a sequence of method calls that create and mutate objects. We use the factory method provided by Pex to assist Pex in generating effective method-call sequences that can help achieve desirable object states. Figure 3 shows our factory method to assist Pex in generating effective method-call sequences. Our factory method accepts two arrays of setting names and values, and adds those 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.

Another important aspect of writing generalized PUTs is to define preconditions. For example, without any preconditions provided, Pex by default generates `null` values for the PUT arguments. To address the preceding issue, we annotate the methods with a tag `PexAssumeUnderTest`[5], which describes that the argument should not be `null` and should be the same type of the actual argument type. We add further assumptions based on the behavior verified by the unit test. For example, the conventional unit test requires a precondition that the setting to be added should not already exist in the storage. We use `PexAssume` to add these additional preconditions to the PUT such as Statement 4 (in Figure 4) in `SaveAndLoadSettingsPUT1`.

In few cases, we identity that the direct generalization of conventional unit tests might not achieve a 100% block coverage. There could be several reasons such as the portions of the code are not covered by the behavior tested by conventional unit tests. In those cases, we identity the un-covered portion of the code and write new PUTs or modify the transformed PUTs to cover these additional scenarios. Consider a sample code example shown in Figure 5. We highlight the un-covered portion of the code in **bold**. The reason for the un-covered code portion in this code example is that the code portion requires a delegate handler to be defined in the class. A delegate handler can be treated as a pointer to a function. These delegates can be used to encapsulate a method with a specific signature and return type. To achieve 100% coverage of the `RemoveSetting` method in the preceding code example, we created a trivial delegate handler and set the value to `Changed`. We present details on the usage of helper techniques in our study in Section 6.

5. Categorization of PUTs

We used existing test patterns [7] to generalize the conventional unit tests. These test patterns are suggested to help the developers in writing effective PUTs. Though with PUTs, the problem of writing different conventional unit tests to test the code under test with different input values is reduced, writing test oracles in PUTs could be a complex task. The developer should be able to give sufficient assertions for testing the various consequences of executing the code under test. To deal with this complexity, developers can use test patterns to answer the important questions of “what” scenarios of the code under test need to be tested and “how” they can be asserted. In our study, we found that a few test patterns are predominantly applicable while others are helpful in a few specific cases. In addition, we found that each PUT can be categorized into more than one test pattern. In all, the patterns supported by Pex give a high code coverage and reduce the efforts for both test driven development and testing after the application is developed. We found that it was not easy to write PUTs for few scenarios of the code under test using these patterns. To address this issue, we proposed two new patterns that can help in writing PUTs. We first explain the classification of the existing categories as *frequently used* and *rarely used* based on the number of PUTs that use the

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

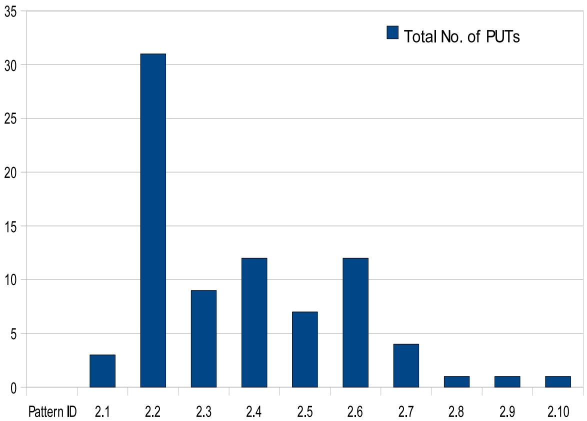

Figure 6. Distribution of test patterns for 70 PUTs.

test pattern and later present our proposed new patterns.

5.1 Existing Test Patterns

We next present the classification of our PUTs into existing test patterns. Table 2 shows the 15 suggested test patterns [7]. Column 1 provides the classification of each pattern in our study. We classify the existing test patterns into two categories based on their frequency of usage: frequently used and rarely used. Column 2 provides the pattern identifier. We will use these identifiers to refer to the patterns in this section. Column 3 provides the pattern name and Column 4 gives the attributes[6] and methods supported by Pex for that pattern.

1. **Frequently used**: We classify a pattern under this category if that pattern was used at least once when writing PUTs. We used 9 Patterns (from Pattern 2.1 to 2.9) for writing PUTs. Figure 6 shows the distribution of pattern usage across all the PUTs. The $x\ axis$ of the chart shows the used patterns and the $y\ axis$ shows the number of PUTs that use these patterns. For each test pattern, we show the number of PUTs in our study.

2. **Rarely used**: We classify a pattern under this category if that pattern was not used in test generalization. We found that 5 out of the 15 Patterns were not used in writing PUTs in our study; from 2.10 to 2.15. The possible reason for not using these patterns in our test generalization is the lack of motivation for these patterns in our current study. For example, Patterns 2.14 and 2.15 are applied in the context of regression testing. However, in our current study our focus is not on regression testing.

5.2 New Patterns

We next describe two new patterns that can be supported, that we found useful during our test generalization.

[6]Pex provides a set of custom attributes to tag Pex class, test methods or parameters

```
[PexMethod()]
public void TestClearRoutinesPUT([PAUT]String[] key) {
  PexAssume.IsTrue(key.Length > 1);
  for (int j = 0; j < key.Length; j++) {
      PexAssume.IsNotNull(key[j]);
  }
  NUnitRegistry.TestMode = true;
  using (RegistryKey mainKey =
      NUnitRegistry.CurrentUser) {
      //enabling appending values to a list
      PexStore.Pool("keys",mainKey);
      for (int j = 0; j < key.Length; j++) {
          RegistryKey parentKey =
              PexStore.Pick("keys") as RegisterKey;
          RegistryKey subKey =
              mainKey.CreateSubKey(key[k - 1]);
          PexStore.Pool("keys",subKey);
      }
      NUnitRegistry.ClearTestKeys();
      PexAssert.IsTrue(mainKey.SubKeyCount == 0);
  }
}
```

Figure 7. A new test pattern that helps reuse previously generated values using a pool.

5.2.1 *Pick a random value from the pool*

Purpose: To reuse generated values by maintaining a pool and randomly picking from those values.

Motivation: We next show the motivation for such a pattern with an illustrative example. In the test generalization phase, we needed to generalize an existing unit test that requires to verify whether adding a `RegisterKey` to an `NUnitRegistry` and then clearing the `NUnitRegistry` works as expected. The `NUnitRegistry` class holds `RegisterKeys` as a tree structure with a main key and an unrestricted number of subkeys in a tree-structured form. Manually adding values and creating a tree structure to check both horizontal and vertical cases at the same time was possible in the existing conventional unit test case. However, when the test case was parameterized, we were able to write PUTs to add all `RegisterKeys` either to one main key or add each `RegisterKey` to the last added key. Due to such restriction, the unit tests generated by PUTs resulted in a reduced block coverage when compared to the conventional unit tests, although the number of PUTs are more than the conventional unit tests. The rationale is that the PUT should be able to generate a tree structure of keys to achieve high code coverage. As the existing test patterns do not meet our current requirement, we designed a new pattern where we maintain a pool of existing `RegisterKeys` and randomly select a key from this pool to add the newly created key as a subkey. Our new pattern can help construct several forms of tree structures automatically. Although we explain our motivation using the `RegisterKeys` test example, our pattern is general and can be applied to test cases that require to reuse previously generated values.

Proposed Pattern: When the input is of the type collection or an array, the values generated by Pex can be added to a pool using our proposed method `PexStore.Pool(<name>,<value>)`. Later, these values added to the pool can be picked randomly using another proposed method `PexStore.Pick(<name>,<value>)`. Figure 7 shows an application of our proposed pattern.

5.2.2 *Unique value generation*

Purpose: To generate unique values when a parameter is a col-

Table 2. Existing Test Patterns Supported By Pex

Classification	Pattern#	Pattern Name	Pex Supported Attributes or Methods
Frequently used	2.1	Arrange, Act, Assert	
	2.2	Assume, Arrange, Act, Assert	
	2.3	Constructor Test	
	2.4	Roundtrip	
	2.5	Sanitized Roundtrip	
	2.6	State Relation	
	2.7	Same Observable Behavior	
	2.8	Commutative Diagram	
	2.9	Cases	PexAssert.Case()
Rarely used	2.10	Allowed Exception	[PexAllowedException]
	2.11	PexAllowedException	[PexExpectedGoals]
	2.12	Parameterized Stub	[PexAssumeUnderTest]
	2.13	Manual Output Review	PexStore.Value()
	2.14	Regression Tests	PexStore.ValueForValidation()
	2.15	Differential Regression Test Suite	

lection of values instead of a single value.

Motivation: In our test generalization phase, for four PUTs, we were unable to achieve the same test effectiveness by the generated conventional unit tests as that of the existing unit tests. We detected the reason to be a required object state; the required object state for the four scenarios can be created only when a unique set of elements is available. In such scenarios, we had to write additional code to ensure that Pex generates only unique values. For example, in the following PUT, we include a `for` loop with a `PexAssume` (as shown below) on each element of the collection to make sure that Pex generates a set of unique values.

```
//PAUT: PexAssumeUnderTest
public void SubstorageSettingsPUT1(
    [PAUT]String subName, [PAUT]String[] name)
{
     PexAssume.IsNotNull(value);
     PexAssume.IsTrue(name.Length == value.Length);
    //assist Pex to generate unique values
     for (int i = 0; i < name.Length; i++) {
         for (int j = 0; j < name.Length; j++) {
           PexAssume.IsFalse(name[i].Equals(name[j]));
         }
     }
  ......}
```

Proposed Pattern: When the input is of the type collection or an array, we propose a new pattern using a custom attribute called `PexGenerateUnique` that can inform Pex to generate unique values. For example, applying the proposed pattern to the preceding example can result in the following easy to write code. We also assume that `PexGenerateUnique` subsumes the properties of `PexAssumeUnderTest`.

```
//PAUT: PexAssumeUnderTest
 public void SubstorageSettingsPUT1([PAUT]String
    subName, [PexGenerateUnique]String[] name)
{
     PexAssume.IsNotNull(value);
     PexAssume.IsTrue(name.Length == value.Length);
  .....}
```

For the four scenarios that required generation of a set of unique values, we could apply the proposed pattern as shown in the above code example and achieve the effectiveness.

6. Helper Techniques Used in Writing PUTs

We next detail on the helper techniques used in writing PUTs. In writing PUTs, we use the factory method and mock object helper techniques. We used additional technique of input-space paritioning to achieve higher code coverage in our Phase 2. We next describe these helper techniques provided by Pex and the others that we used in our study.

6.1 Factory Methods

A commonly used helper technique in test generalization to assist effective test input generation is 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 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. An example factory method we used in our study is shown in Figure 3. 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.

6.2 Mock Objects

Mock objects help test features in isolation by replacing unrelated features 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 of saving of an `NUnitProject`. When a new `NUnitProject` is created, a `xml` file is saved in the project directory that saves 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 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* (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 rightly 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 (that is constructed based on the configurations that are added). Generalization of these conventional unit-tests is not straight forward 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 the 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 the external factors 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 being able to generate 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 only mocked the methods used by the code under test. Figure 8 shows a code snippet from the mocked object.

By using the mock object helper technique, we were able to achieve generalization of both the existing tests, `SaveEmptyConfigs` and `SaveNormalProject` resulting in a block coverage of 61.64% and 67.92%, respectively.

```
.........
.........
public MockXmlTextWriter(string filename,
  Encoding encoding)
{
     this.fileName = filename;
}

public void WriteAttributeString
  (string attributeName, string value)
{
     xmlString = xmlString + " " + attributeName
          + "=" + "" + value + "";
}

public void WriteEndElement()
{
     xmlString = xmlString + " />";
}

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

Figure 8. Sample code from the `MockXmlTextWriter` mock object

6.3 Input-Space Partitioning

Input-space partitioning 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. We defined two blocks where the first block includes integers and the second block includes strings. We wrote separate PUTs for covering these blocks. 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.

7. Benefits of PUTs

We next present the benefits of retrofitting conventional unit tests with PUTs. In test generalization we transformed 57 conventional unit tests of 10 test classes resulting in 49 PUTs. In order to achieve more block coverage, we wrote 21 new PUTs for 6 of the 10 classes under study. Table 3 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, followed by the percentage of amenable test cases, 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" is the number of PUTs written Phase 1 and "#New PUT" shows the number of new PUTs written in Phase 2 to achieve more coverage. Section 8 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 test cases. 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 also new blocks that are not covered by conventional unit tests. Column "#Defects" shows the number of defects detected by the PUTs that were not detected by the existing conventional unit tests. This column again shows a cumulative

Table 3. 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	77.36	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

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. Nevertheless Phase 2 has its significance in being able to detect why the coverage was low in Phase 1 and the amount of effort we took in writing the new PUTs in comparison to generalizing the existing conventional unit tests and will be discussed in Section 9. We found three major benefits of retrofitting conventional unit tests for PUTs: code coverage, new defects, and test code. We observed that generalization increases code coverage and detected new defects that were not discovered 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 2 and Column 5 in Table 3). Sections 7.1, 7.2, and 7.3 explain these details.

7.1 Coverage

As generalized test cases often help cover more scenarios, we found that test generalization helped to have a significant increase in the block coverage as shown in Table 3. For example, test generalization of the `RegistrySettingsStorage` class shows an increase in the coverage of 45.24%. In addition, the tests generated by the PUTs in the test generalization achieved coverage of new blocks that are not covered by the existing conventional unit tests. For example, the generalization of conventional unit tests in the `RecentFilesTests.cs` covered 14 additional blocks (on average for that class for all PUTs). In one scenario of `RecentFilesTest.cs`, there is a decrease in the % of coverage when the conventional unit tests were transformed to PUTs; however, there is a significant increase in the number of blocks covered by the PUTs. The decrease in the coverage is due to the increase in the number of blocks covered by the PUT when compared to the corresponding conventional unit-test. The difference is primarily due to the Pex strategy of reporting block coverage as Pex cannot detect the total number of blocks correctly when some methods are not analyzed. For example, the numbers of blocks covered by two PUTs in the `RecentFilesTest.cs` were 56 and 58, while the conventional unit tests accounted for 23 and 25, respectively. The ratio of blocks covered to the total number of blocks accounted by PUTs are 41/56 (73.21%) and 41/58 (70.69%). Similarly, for conventional unit tests the ratio was 18/23 (78.26%) and 18/25 (72.00%). Therefore, despite an increase in the total number of blocks covered, there is a decrease in the % of code coverage. 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).

7.2 Defects

We found 7 new defects after test generalization that are 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. When testing, adding a key hierarchy and on checking for the count or on clearing the keys, we found dissimilar behavior for two test cases. The PUT was written to take three test inputs. For one of the test cases 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.

7.3 Test code

Test generalization also helped significantly reduce the test code as shown in Column "#PUT" of Table 3. Often, test generalization either helps reduce the amount of code in a single test method or helps combine several 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

```
[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 case in the PUT. In addition, the PUT achieved higher coverage compared to the five conventional test cases as the `MaxValue` is now accepted as an argument and the concrete values are generated from the argument based on captured constraints.

8. Conventional Unit Tests Not Amenable to Test Generalization

In our study, we found that 12.75% of conventional test cases are not amenable to test generalization. As shown in the Table 3, only a % of existing conventional unit tests were amenable to test generalization. There are two common scenarios under which we found that the test cases are not amenable to test generalization.

Default checks: We found that there are several conventional unit tests that verify the default values, which are often static values. We believe that the purpose of these tests could be to make sure that the developers do not change the static values accidentally. We believe that these test cases are not amenable for test generalization as values verified by these test cases are constants.

Missing test oracles: We found that test generalization may cause the loss of test oracles in a few scenarios. These scenarios can often occur when generalizing the test cases that belong to the PUT patterns *Roundtrip* and *Commutative diagram*. We explain this issue using the illustrative example shown below.

```
public void Canonicalize() {
  PexAssert.AreEqual(@"C:/folder1/file.tmp",
      PathUtils.Canonicalize(@"C:/folder1/./folder2/
      ../file.tmp"));
}
```

The `Canonicalize` method in `PathUtils` accepts a `string` parameter and uses a complex procedure to transform the input `string` into a standard form. It is easy to identify the expected output for the concrete strings such as “`C:/folder1/./folder2/../file.tmp`”. However, when the conventional unit test is generalized with a variable, it is non-trivial to identify the expected output either using *Roundtrip* and *Commutative diagram* PUT patterns. Developers will need to implement a logic to assert the behavior of the method under test. The amount of effort required in such cases could be higher than the effort required to write the actual method under test and therefore such test cases are not amenable to test generalization.

9. Discussion

One of the major limitations of PUTs is that PUTs require more efforts from developers than conventional unit tests. Though PUTs reduce the complexity of having to write 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 give a test oracle that will be able to deal with the generality of test inputs. We show that to reduce the complexity of writing PUTs, we carried out 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 helper techniques to write more PUTs to assist Pex in generating high covering test cases.

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 4 with an example. In Phase 2, we discovered the uncovered code portions and wrote more PUTs to assist Pex to generate test cases to cover the un-covered code portions. Based on our experience, we intuitively believe that to make use of the test effectiveness achieved by writing PUTs and to reduce the cost involved in writing PUTs, an optimal solution could be retrofitting unit testing with PUTs by transforming the 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, though the usage of the suggested test patterns and the helper techniques can reduce the effort of developers in writing PUTs and allow the test generation tool to generate high-covering test cases, the complexity lies in being able to use them. In general, developers might consider it a tougher job to learn the helper techniques and use them to write PUTs than write all the 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 complexity of writing PUTs is a trade-off between quality and effort.

10. Related Work

In the recent past, a lot of work has been focused to generalize test case specifications to generate test inputs automatically. Pex accepts PUTs and uses symbolic execution to generate test inputs, similarly, other existing tools such as Parasoft Jtest [5] and CodePro AnalytiX [6] adopt the design-by-contract approach [10] and allow developers to specify method preconditions, postconditions and class invariants for the unit under test and carry out symbolic execution to generate test inputs. More recently, Saaf et al. [11] propose theory-based testing and generalize six subjects to show that the proposed theory-based testing is more effective compared to the traditional example-based testing. A theory is a partial specification of a program behavior and is a generic form of test methods where the assertions should hold for all inputs that follows the assumptions described in the test methods. A theory is very similar to a PUT and Saaf et al.'s approach uses these defined theories and applies constraint solving mechanism based on path coverage to generate test inputs similar to Pex.

11. Threats to validity

The threats to validity primarily include the degree to which the subject programs, faults, and test cases are representative of true practice. The subject used in the empirical study is small, although the defects detected in them are real. These threats could be reduced by conducting more experiments on wider types of subjects in future work. The threat to internal validity is our assumption that the existing conventional unit tests are complete in themselves as this assumption can bias our results of PUTs being able to detect defects that were not detected by the conventional unit tests. One threat to measuring the coverage information validity is that our experiment makes use of the coverage report generated by Pex that report the block coverage. We hope that these reports give us precise reports of the actual coverage for each PUT.

12. Conclusion

We conducted an empirical study to investigate the utility of PUTs in unit testing. We first generalize the existing conventional test cases, which is a process of 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 several benefits of test generalization such as increase in the block coverage by 9.68% (on average) 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 categories of conventional test cases that are not amenable for test generalization and also proposed new PUT patterns. We present details on the utility of the suggested test patterns and helper techniques that help in writing PUTs. We further discuss the limitations of retrofitting unit testing with PUTs in terms of the effort required in writing these PUTs.

13. References

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

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

[3] Test-Driven Development in .NET, The NUnit Testing Framework, 2006. `http://www.developerfusion.com/article/5240/testdriven-development-in-net/2/`.

[4] JUnit for Test Driven Development, 2008. `http://www.junit.org/`.

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

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

[7] 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`.

[8] 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.

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

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

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

[12] K. Sen. Concolic testing. In *Proceedings of the 2007 IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 571–572, 2007.

[13] 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.

[14] 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.

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