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 the software development life cycle, unit testing has been widely adopted by software industry to ensure high-quality software. Recent advances in software testing introduced Parameterized Unit Tests (PUTs), which are a generalized form of Conventional Unit Tests (CUT). With PUTs, test data can be generated automatically using the Dynamic-Symbolic-Execution (DSE) approach. PUTs are powerful and provide an effective means of guided test generation when used with a DSE-based tool such as Pex from Microsoft Research. To leverage the benefits of PUTs in testing existing legacy applications in software industry, we propose the first systematic procedure of test generalization (transforming existing CUTs to PUTs). We also conduct an empirical study, on three real-world applications, that shows benefits of test generalization. We show that test generalization reduces 407 CUTs to 224 PUTs (45%), thereby reducing the effort of maintaining test code. Along with achieving higher code coverage (a maximum increase of 53% for one class under test and 10% for one application used in our study), test generalization helped detect 16 new defects that are not detected by existing CUTs. A few of these defects are quite complex and are hard to be detected using CUTs. Given these benefits of test generalization, we expect that developers in software industry can use our systematic procedure to transform their CUTs to PUTs to exploit the benefits of PUTs, and thereby increase the quality of their developed software.

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 does not accept any parameters and includes 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) [23] are a new advance in the field of software testing. Unlike Conventional Unit Tests (CUT), PUTs accept parameters. The major advantage of PUTs compared to CUTs is that developers do not need to provide test data in PUT, but need to provide only 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) [14, 10, 20] approach. Given code under test, DSE explores the code under test with random or default 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. Pex [22, 6] from Microsoft Research is an example state-of-the-art DSE-based test generation tool that accepts PUTs and uses DSE to generate test data. Pex explores PUTs and generates a CUT (for each set of concrete values) that explores a new path in the PUT and the code under test. PUTs thus provide a generic representation of CUTs and are more powerful since PUTs provide a means for guided test generation when used with an automatic test generation tool such as Pex.

Although it is known that PUTs are powerful, it is quite challenging to write PUTs for existing legacy applications. Despite the fact that PUTs relieve developers from providing test data in CUTs, PUTs still require test scenarios (such as method-call sequences) for exercising the code under test. Writing realistic test scenarios requires the knowledge regarding the behavior of the code under test. On the other hand, most of the existing applications in software industry have CUTs that include valuable information of test scenarios. In practice, CUTs are often used as specifications to understand the behavior of the code under test. Therefore, either developers of existing applications or third-party testers can transform these CUTs to PUTs by reusing test scenarios described in those CUTs. We refer to this process of transforming CUTs to PUTs for exploiting the benefits of PUTs as *test generalization*.

In practice, test generalization has three major benefits. First, from PUTs, DSE-based approaches can be used to automatically generate CUTs that achieve a high coverage of the code under test. Therefore, test generalization helps generate new CUTs that cover additional paths (in the code under test), which are not covered by the legacy CUTs. Second, test generalization increases the fault-detection capabilities of the test suite by appending more CUTs to the existing CUTs. Third, test generalization helps reduce the efforts required in maintenance of test code since a single PUT can represent multiple CUTs.

Given the benefits of test generalization, in this paper, we propose a systematic procedure for assisting developers in generalizing CUTs to PUTs. To the best of our knowledge, ours is the first empirical study that proposes a systematic procedure for generalizing CUTs to PUTs and also shows an empirical evidence of the

benefits of test generalization. The main objective of our systematic procedure is to take existing CUTs and transform them to PUTs to test the same or generalized behaviors as the ones tested by the existing CUTs. Our systematic procedure describes challenges that are faced during test generalization and proposes solutions of how developers can address those challenges. These challenges can be classified into two categories: challenges specific to our test generalization and challenges in general for the existing state-of-the-art DSE-based approaches in generating CUTs from PUTs.

An example challenge faced during our test generalization (first category) is that Pex or any other DSE-based approach requires guidance in generating legal values for the parameters of PUTs. Without these legal values, the detected defects turn out to be false positives. These legal values are the values that satisfy preconditions of the code under test and help setting up test scenarios to pass test assertions. For example, consider a method under test that accepts an integer parameter and includes a precondition that the parameter should be greater than 1000. Pex requires guidance to generate a legal value greater than 1000 for satisfying the precondition of the method under test. To address this issue, in our systematic procedure, we suggest developers how to add necessary assumptions to PUTs in guiding Pex to generate legal values. Another example challenge during test generalization is developers may face difficulties while generalizing test oracles in CUTs. We propose several test patterns [9] that help developers in addressing the challenges faced during generalization of test oracles.

Our systematic procedure also presents alternative solutions for addressing the second category of challenges, which are generally faced by the existing state-of-the-art DSE-based approaches in generating CUTs from PUTs [21]. For example, Pex faces challenges in generating method-call sequences for achieving desirable object states of non-primitive parameters of PUTs. These desirable object states are the states that help explore paths in the code under test by covering `true` or `false` branches. Another example challenge is related to PUTs that require desired states of the environment, external to the application, such as a file needs to exist in the file system for each generated CUT. In our systematic procedure, we suggest developers of how to assist Pex in addressing these challenges.

We show the benefits of generalizing CUTs to PUTs using our systematic procedure with three open source applications that are widely used in software industry (as shown by their number of downloads). As we are not developers of these applications, we generalize CUTs of these applications as third-party testers using our systematic procedure. Our results show that test generalization reduces 407 CUTs to 224 PUTs (45%), showing the significance of test generalization in reducing the effort of maintaining test code. Test generalization also helped detect 16 new defects that are not detected by existing CUTs of the applications. A few of these defects are quite complex and are hard to be detected using the existing CUTs.

In summary, the paper makes the following major contributions:

- A systematic procedure for assisting developers in generalizing existing legacy unit tests to PUTs to leverage the benefits of PUTs.
- The first empirical study that investigates the benefits of test generalization. In our empirical study, we show that test generalization increases branch coverage by 4% (with a maximum increase of 53% for one class under test and 10% for one application under analysis) on an average for all three applications used in our empirical study. We show that test generalization detects 16 new defects that are not detected by existing CUTs. We also show that test generalization reduces the number of unit tests by 45% (407 CUTs are transformed to 224 PUTs), thereby reducing the efforts required in maintaining the test code. The detailed results of our empirical study are available at `https://sites.google.com/site/asergrp/projects/putstudy`.
- Supporting techniques and PUT patterns that can assist developers in writing PUTs.

The rest of the paper is structured as follows. Section 2 presents background information about Pex. Section 3 presents our systematic procedure. Section 4 describes how the PUT patterns and supporting techniques can assist developers in writing PUTs. Section 5 presents the results of our empirical study. Section 6 discusses the limitations of PUTs. Section 7 discusses the related work. Section 8 discusses the threats to validity. Finally, Section 9 concludes.

2. BACKGROUND

We use Pex [6] as an example state-of-the-art DSE-based test generation tool for generating CUTs using PUTs. Pex is a white-box test generation tool for .NET programs. Parts of Pex may be integrated into a future version of Microsoft Visual Studio, benefiting an enormous number of developers in industry. Pex accepts PUTs and symbolically executes the PUTs and the code under test to generate a set of CUTs that can achieve high coverage of the code under test [14]. Pex has been widely used both in academia and industry, which is reflected by its download counts (Feb. 2008 - Oct. 2009), 17366 and 13022, respectively. Pex is applied on industrial code bases and is known to have detected serious defects in a software, which had already been extensively tested [22].

3. SYSTEMATIC PROCEDURE

In this section, we present how a developer can generalize existing CUTs to PUTs by using our systematic procedure. We use illustrative examples from the NUnit framework [12] for explaining our procedure. Our procedure includes four major steps. First, the developer promotes concrete values and other local variables in the CUT as parameters for PUTs (S1). Second, the developer identifies the test pattern for the PUT that best matches the existing CUT to be generalized (S2). Identification of test pattern helps developer in generalizing test oracles. Third, the developer adds necessary assumptions to PUTs to guide Pex in generating legal values for the parameters of PUTs (S3). Fourth, the developer uses supporting techniques to assist Pex while generating unit tests from PUTs (S4). We first present a method under test and a CUT from the NUnit framework and next explain each step in detail.

3.1 Method Under Test and CUT

Figure 1 shows a method under test `SaveSetting` from the `SettingsGroup` class 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 CUT for testing the `SaveSetting` method. The CUT saves two setting values (of types `int` and `string`) and verifies whether the values are set properly using the `GetSetting` method. The CUT verifies the expected behavior of the `SaveSetting` method only for the setting values of types `int` and `string`. This

```
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 under test.

```
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 CUT to test the `SaveSetting` method (shown in Figure 1)

CUT is the only test for verifying `SaveSetting` and includes two major issues. First, the CUT does not verify the behavior for the types `bool` and `enum`. Second, the CUT does not cover the `true` branch in Statement 8 of Figure 1. The reason is that the CUT does not invoke the `SaveSetting` method more than once with the same setting name. This CUT achieves 10% branch coverage of the `SaveSetting` method. We next explain how the developer can generalize CUT to a PUT and address these two major issues.

3.2 S1: Concrete values and Local Variables

To generalize this CUT, the developer first identifies concrete values and local variables in the CUT and promotes them as parameters. For example, the unit test includes a `string` "`Charlie`" in Statement 4. The developer replaces this concrete value with a symbolic value by promoting the value as a parameter 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 under test. Here, the developer can promote the `string` "`Charlie`" and the `int 5` as a single parameter of type `object` for the PUT since `SaveSetting` accepts the parameter of type `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. This example shows that a single PUT can achieve the same test effectiveness as multiple CUTs with different concrete values. In addition to promoting concrete values as parameters of PUTs, the developer promotes other local variables such as the receiver object (`testGroup`) of `SaveSetting` as parameters. Promoting such receiver objects as parameters can help generate different object states (for those receiver objects) that can help cover additional paths in the code under test. Figure 3 shows the PUT generalized from the CUT shown in Figure 2.

3.3 S2: PUT Patterns and Test Oracles

The developer next analyzes the CUT to identify pattern [1] for

```
//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: A PUT for the CUT 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`.

the PUT that best match the existing CUT. Identifying PUT patterns can help in generalization of the CUT since these patterns can serve as guidelines for designing the PUT and for adding test oracles. In our CUT, a setting is stored in the storage using `SaveSetting` and is verified using `GetSetting`. An analysis of the CUT 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, the developer can find that the test oracle can include the `GetSetting` method to assert the behavior of the `SaveSetting` method under test. Section 4 presents more details on the use of PUT patterns during test generalization. In our empirical study, we identify that these patterns cover a broad range of CUTs and assist during test generalization.

3.4 S3: Assumptions

A challenge faced during test generalization is that Pex requires guidance in generating legal values for the parameters of PUTs. These legal values are the values that satisfy preconditions of the code under test and help setting up test scenarios to pass test assertions. For example, without any assumptions, Pex by default generates `null` values for non-primitive parameters such as `st` of the PUT. To guide Pex in generating legal values, the developer can add sufficient assumptions to PUTs. For example, in our PUT, the developer annotates each parameter with the tag `PexAssumeUnderTest`[1], which describes that the parameter should not be `null` and the type of generated object should be the same as the specified type. The developer can add further assumptions such as Statements 3 and 4 to PUTs based on the behavior verified by the CUT.

3.5 S4: Supporting Techniques

In general, Pex (or any other existing DSE-based approaches) faces challenges in generating CUTs from PUTs that include parameters of non-primitive types or require interactions with the environment. To address these challenges, the developer needs to provide assistance to Pex. To assist Pex in addressing the first challenge related to non-primitive types, developers can write factory methods that produce instances of non-primitive types such as `st` in the PUT. Figure 4 shows an example factory method for the `MemorySettingsStorage` class. To address the second challenge related to the interactions with the environment, developers

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

can write mock objects. These mock objects help test features in isolation especially when the test interacts with environments such as file system. Section 4 provides more details on how to use these two supporting techniques: factory methods and mock objects.

3.6 Transformed PUT

Figure 3 shows the PUT after applying Steps 1 to 4 on the CUT. 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 CUT requires two method calls shown in Statements 3 and 4 of Figure 2 to verify whether `SaveSetting` correctly handles these types. On the other hand, only one method call is sufficient in the 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 CUT did not include tests for verifying these two types. Our generalized PUT automatically handles these additional types, serving as a primary advantage of PUT since it helps reduce the test code significantly without reducing the behavior tested by the CUT.

When we applied Pex on the PUT, Pex generated 8 CUTs from the PUT. These CUTs 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 CUTs, resulting in a reduced test code maintenance. Furthermore, the CUT used for generalization achieved a branch coverage of 10%, whereas the CUTs generated from the PUT has achieved a final branch coverage of 90%. Although PUT achieved a higher code coverage compared to the CUT, the PUT still could not cover the `true` branch of Statement 16. Developers while doing generalization can verify these not covered portions and can either enhance PUTs or write new PUTs for achieving additional coverage of those not covered portions[2].

4. PATTERNS AND SUPPORTING TECHNIQUES

We next detail on the PUT patterns and supporting techniques that can be used in writing PUTs. In our study, we primarily use patterns to help generalize test oracles. In addition, we use the supporting techniques of factory methods and mock object to deal with the issues of desirable object states and interactions of the code under test with the environment, respectively. We next describe these techniques in detail.

4.1 PUT Patterns

A major challenge of writing PUTs is writing a generalized test oracle. When writing CUTs, it is not challenging to write test oracles since for a particular CUT, expected output values can be easily determined for the given input test data based on the method under test. In contrast, when testing with PUTs, determining the expected output values is not trivial; a PUT should be designed to assert output values based on a number of input values generated by Pex for that PUT. To address this issue, we propose 15 PUT patterns [9] that help developers to deal with this issue of generalizing test oracles. Table 1 shows 13 of the proposed 15 PUT patterns[3].

[2]Recall that the objective of our study is to generalize CUTs to PUTs for comparing the benefits of PUTs over existing CUTs. Therefore, we did not write additional PUTs that can achieve better branch coverage.

[3]The other two PUT patterns are related to regression testing, which is not within the scope of our paper.

#	Pattern Name
1	Arrange, Act, Assert (AAA)
2	Assume, Arrange, Act, Assert (AAAA)
3	Constructor Test
4	Roundtrip
5	Sanitized Roundtrip
6	State Relation
7	Same Observable Behavior
8	Commutative Diagram
9	Cases
10	Allowed Exception
11	Reachability
12	Parameterized Stub
13	Manual Output Review

Table 1: PUT Patterns

```
01: public void Clear()
02: {
03:  root = null;
04:  Count = 0;
05: }
```

Figure 5: `Clear` method of `AvlTree` class of DSA.

```
01: public void ClearCUT()
02: {
03:  AvlTree<int> actual = new AvlTree<int> { 1,3,5,6,9,7 };
04:  Assert.AreEqual(6, actual.Count);
05:  actual.Clear();
06:  Assert.AreEqual(0, actual.Count);
07: }
```

Figure 6: Existing CUT to test the `Clear` method.

The proposed PUT patterns serve two purposes: (1) they assist in designing a PUT based on the developers' test objective, and (2) they assist in determining test oracles that can deal with the generality of the PUT. Patterns 1, 2, 9, 10, and 11 (shown in Table 1) primarily assist developers in designing PUT based on the test objective. For example, developers can use Pattern 2 (AAAA) when the method under test needs to be tested for a particular range of input values (the PUT includes assumptions), and the behavior of the method under test can be asserted using assertion statements (the PUT includes assertions). Similarly, developers can use Pattern 10 (Allowed Exception) when a method under test may throw a particular exception, and such an exception being raised does not indicate that the unit test fails. The remaining patterns listed in the table assist developers in writing test oracles. For example, developers can use Pattern 6 (State Relation) when a method under test causes an internal change to the object state, which in turn can be observed through other methods.

We next illustrate the use of PUT patterns in designing a PUT and in determining a test oracle. Figure 5 shows the `Clear` method from `AvlTree` of the DSA [11] library. The `Clear` method clears the contents of the `AvlTree` object. Figure 6 shows the existing CUT to test the `Clear` method. The objective of the CUT is to create an `AvlTree` object with the given list of items, invoke the `Clear` method, and assert if the elements are removed. To transform this CUT to a PUT, developers promote the input list to the constructor of `AvlTree` as a parameter to the PUT. Based on the test objective, developers identify two patterns to write the PUT: (1) AAAA, there should be an assumption on the parameter since the parameter should be non-null, and the result of the `Clear` method should be asserted, and (2) State Relation, since the behavior of

```
01: public void ClearPUT([PAUT]List<int> values)
02: {
03:  AvlTree<int> actual = new AvlTree<int>(values);
04:  PexAssert.AreEqual(values.Count, actual.Count);
05:  actual.Clear();
06:  PexAssert.AreEqual(0, actual.Count);
07: }
```

Figure 7: PUT transformed from CUT in Figure 6, using the AAAA and State Relation patterns.

the `Clear` method can be observed using the `Count` field, i.e., observe the change in the object state. Using the identified patterns, developers can write a PUT shown in Figure 7.

4.2 Factory Methods

Pex faces challenges in handling PUT parameters of non-primitive types, because these parameters require method-call sequences (that create and mutate objects of non-primitive types) to generate desirable object states. These desirable object states are the states that help explore new paths or branches 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` object should already include a value for the setting name `sn`. Although Pex includes a demand-driven strategy for generating method-call sequences, Pex's strategy can generate method-call sequences effectively in certain limited scenarios where the constructors either accept primitive parameters or explicitly state the actual type of the parameter.

To assist Pex in generating effective method-call sequences that can help achieve desirable object states, developers can write method-call sequences inside factory methods, supported by Pex. Figure 4 shows an example factory method for the `MemorySettingsStorage` class. The 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 the factory method for creating a desirable object state with five elements in the storage[4]. The same factory method can be reused for all other PUTs using the `MemorySettingsStorage` class as parameter types.

4.3 Mock Objects

A mock object [15] is an implementation for simulating environments such as file systems. Mock objects help test unit behaviors in isolation where the interactions of the unit under test with the real environment are replaced with the mock objects. Automatic test generation tools such as Pex could require various desirable environment states to generate high-covering unit tests. Creating these desirable environment states requires extensive test setup. Since such test setup can generate a lot of garbage in the environment (such as adding extra files in the file system), a significant amount of test cleanup is also required. We next describe how developers can use mock objects with an example from our study.

In the `NUnitProject` class of the NUnit application, to test the `save` method, the developer can use a mock object to replace the interactions of code under test with the `XmlTextWriter`. The purpose of the `save` method is to write configuration information to an XML file. This XML file is expected to be created when the project is created, i.e., an instance of `NUnitProject` is created. Two existing CUTs `SaveEmptyConfigs` and `SaveNormalProject` test this `save` method. These CUTs add configurations to the XML project files and assert whether the files are saved in the right format

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

```
01: public class MockXmlTextWriter{
02:  public MockXmlTextWriter(string filename,
     Encoding encoding)
03:  {
04:     this.fileName = filename;
05:  }

06:  public void WriteAttributeString
     (string attributeName, string value)
07:  {
08:     xmlString = xmlString + " " + attributeName
           + "=" + "" + value + "";
09:  }

10:  public void Close()
11:  {
12:     xmlString = xmlString.Replace("/> />", "/>" +
           System.Environment.NewLine + "</"
              + startString + ">");
13:     CreatedProjects.currentProject = xmlString;
14:  }
     ............
15: }
```

Figure 8: Sample code from the `MockXmlTextWriter` mock object. In the `WriteAttributeString` method, the argument string is appended to a global string `xmlString` and this global string represents the content written to the xml file.

and contain the added configuration information. Both the CUTs depend on a default empty project that is created using the test setup method. Therefore, the requirement to test the `save` method is to provide the project configuration file (the XML file) in a *specific location*, i.e., the directory location where the project is saved (when a project is saved, a new directory is created as the project directory and the XML file is created in this directory).

To generalize these CUTs, the developer can promote the project path (a parameter to the `save` method and a local variable in the CUTs) as a parameter to the PUT. However, generalization of these CUTs is not straightforward. By promoting the project path as the PUT's paramater, every unit test generated by Pex using the PUT requires an XML file in the location (reflected by the generated value for the parameter). The reason is that when the `save` method is invoked, the XML file is accessed using `XmlTextWriter`. Without the XML file, the `save` method throws an exception. Therefore, to avoid the complexity of creating a "real" file at a "real" location and to prevent the exception thrown by the `save` method, the developer can mock the expected behavior of `XmlTextWriter` with `MockXmlFileWriter`. This mock object simulates the behavior of `XmlTextWriter`, but unlike the real object, it appends the input text to a `string`, still preserving the output of the actual behavior of `XmlTextWriter`, and avoiding the interactions with the file system. Figure 8 shows a code snippet of the `MockXmlTextWriter` class. By using the mock-object technique, the developer can achieve generalization of both the CUTs to test the `save` method.

5. EMPIRICAL STUDY

We conducted an empirical study using three real-world applications to show the benefits of generalizing CUTs to PUTs in specific and also to show the applicability of our systematic procedure in general. In our empirical study, we show the benefits of PUTs over existing CUTs in terms of three metrics: branch coverage, the number of detected defects, and the number of tests (their LOC) being reduced by test generalization. In particular, we address the following three research questions in our empirical study:

- **RQ1: Branch Coverage.** How much higher percentage of *branch coverage* is achieved by PUTs compared to existing CUTs? Since PUTs are a generalized form of CUTs, this

Subject	Downloads	Code Under Test					Existing Test Code		
		#Classes	#Methods	#KLOC	Avg. Complexity	Max. Complexity	#Classes	#Methods	#KLOC
NUnit	$193,563^a$	9	87	1.4	1.48	14	9	49	0.9
DSA	2241	27	259	2.4	2.09	16	20	337	2.5
QuickGraph	7969	56	463	6.2	1.79	16	9	21	1.2

[a]Number of reads as shown in sourceforge for the year 2009

Table 2: Details of the subject applications

research question helps to address whether PUTs can achieve additional code coverage compared to CUTs.

- **RQ2: Defect Detection.** 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 higher fault-detection capabilities compared to CUTs.
- **RQ3: Test-code maintenance.** How many tests are reduced by generalizing CUTs to PUTs? This research question helps to address whether PUTs require less effort in maintaining test code compared to CUTs.

To address these research questions, we execute the existing CUTs (of each application) and the unit tests generated by Pex with the help of the transformed PUTs, and measure the preceding three metrics. Initially, we execute the existing CUTs and meticulously record the achieved branch coverage and also the failing tests as defects. In addition, we measure the number of CUTs and lines of code (LOC) of CUTs for all applications. We then generalize these CUTs to PUTs based on our systematic procedure and use Pex to generate unit tests. We then execute the new tests generated from PUTs and record achieved branch coverage, the number of failing tests, the number of PUTs, and the number of LOC of PUTs. Furthermore, we manually analyze all failing tests to ensure that the failing tests are not due to invalid PUTs, such as missing assumptions on the parameters. We thus confirm that the finally recorded failing tests are due to defects in the code under test.

For measuring the branch coverage of the code under test, we use a coverage measurement tool, called NCover[5]. For measuring the code metrics such as LOC and cyclomatic complexity, we use the Count Lines of Code[6] (CLOC) tool.

5.1 Subject Applications

We use three open source applications in our study: NUnit [12], DSA [11], and Quickgraph [8]. The criteria for choosing these applications are (1) popular usage of these application in industry as shown by their download records in their hosting web sites, (2) the existing conventional unit tests, (3) the number of lines of the code under test of these applications, and (4) the code under test itself including non-trivial logic to validate the feasibility of our systematic procedure. Table 2 shows the characteristics of the three subject applications. Column "Subject" gives the name of the subject application. Column "Downloads" gives the number of downloads of the application as listed in its hosting web site. Column "Code Under Test" gives details of the code under test (of the application) in terms of the number of classes ("#Classes"), number of methods ("#Methods"), number of lines of code ("#LOC"), and the average complexity of the code under test. Column "Existing Test Code" provides details on the existing test code similar to the details of the code under test. Subcolumns "#Classes", "#Methods", and "#LOC" give the corresponding metrics of the test code. We next provide details of all the subject applications.

[5]http://www.ncover.com/

[6]http://cloc.sourceforge.net/

5.1.1 NUnit

We use an open source application, called NUnit [12], as one of the subject applications for our study. NUnit, a counterpart of JUnit for Java [3], is a widely used open source unit-testing framework for all .NET languages. NUnit is written in C# and uses attribute-based programming model [2] through a variety of attributes such as `[TestFixture]` and `[Test]`. The rationale behind choosing NUnit for test generalization is its popularity (increase in the usage from 450 reads in the year 2002 to $193,563$ in the year 2009) as listed by its hosting website SourceForge[7]. Furthermore, the large number of manually written unit tests available with the application makes NUnit a suitable subject for test generalization. For the purpose of the study, we used nine classes from the `util` package (`nunit.util.dll`), which is one of the core components of the framework. We chose the `util` package with two reasons: (1) it is one of the basic modules developed for the framework and (2) it is an independent module and is not dependent on the other modules of the NUnit framework. We chose the nine classes as the code under test since the logic in these classes is non-trivial; the average complexity is 1.48 (maximum of 14) as shown in Table 2. For our study, we used NUnit release version 2.4.8.

5.1.2 DSA

Data Structures and Algorithms (DSA) [11] is a library that contains implementation of data structures and algorithms, a few of which are not available in the .NET 3.5 framework. There are two major packages in the library's source code, `Dsa.DataStructures` and `Dsa.Algorithms`. The DSA library includes 27 classes and 259 methods. The existing test suite available with the DSA library includes 20 test classes and 337 CUTs. The rationale behind choosing DSA is the size of its test suite available with the release version 0.6. The existing test suite includes CUTs that test all the classes of the DSA source code. Furthermore, another reason for choosing DSA for our study is the influence of DSA in the software industry as shown by the number of downloads, a total of $2,236$ downloads as shown by its hosting site, Codeplex[8].

5.1.3 QuickGraph

QuickGraph [8] is a C# graph library that provides various directed/undirected graph data structures. QuickGraph also provides algorithms such as depth-first search, breadth-first search, and A* search. QuickGraph includes 56 classes and interfaces with 6.2 KLOC. The existing test suite available with the source code release of the library incldues 9 test classes and 21 CUTs. The existing test suite includes CUTs that primarily test two search algorithms, one sorting algorithm, and other graph concepts. We generalized only these existing CUTs of this subject application. Since QuickGraph includes graph search algorithms, the code under test includes non-trivial logic and is therefore a suitable subject for test generalization. QuickGraph is also popularly used as reflected by $7,969$ downloads (shown by CodePlex).

[7]http://sourceforge.net/

[8]http://codeplex.com/

Subject	Namespace\ Class	Branch Coverage		Coverage Increase
		CUTs	PUTs	
NUnit				10%
	MemorySettings Storage	100.00%	100.00%	
	NunitProject	76.00%	76.00%	
	NunitRegistry	85.00%	85.00%	
	PathUtils	79.00%	79.00%	
	RegistrySettings Storage	**48.00%**	**86.00%**	
	RemoteTestAgent	100.00%	100.00%	
	ServerUtilities	91.00%	91.00%	
	SettingsGroup	**39.00%**	**92.00%**	
	TestAgency	86.00%	86.00%	
DSA				1%
	Algorithms	**93.00%**	**94.00%**	
	DataStructures	**99.00%**	**100.00%**	
	Properties	96.00%	96.00%	
	Utility	78.00%	78.00%	
Quick Graph				1%
	Default	87.00%	87.00%	
	Algorithms	**90.00%**	**93.00%**	
	Collections	97.00%	97.00%	
	Concepts	72.00%	70.00%	
	Exceptions	100.00%	100.00%	
	Predicates	83.00%	83.00%	
	Representations	**82.00%**	**84.00%**	

Table 3: Coverage of the existing CUTs and the unit tests generated by Pex using the generalized PUTs

5.2 RQ1: Branch Coverage

We next describe our empirical results for addressing RQ1. We execute the existing test suite (CUTs) and measure the branch coverage achieved by these CUTs. We then measure the branch coverage achieved by the unit tests generated by Pex from the given PUTs. Consequently, we compare the branch coverage achieved by both the CUTs and PUTs for each class or namespace.

Table 3 shows the branch coverage achieved by executing the existing CUTs and the unit tests generated by Pex using the generalized PUTs. Column "Coverage Increase" shows the overall increase in the branch coverage from using the existing CUTs and the generalized PUTs. We use NCover to measure these reported branch coverages. For the subjects DSA and QuickGraph, since we generalized all unit tests in the existing test suite, we report the coverage for each namespace. For NUnit, we generalized nine test classes and therefore, we report branch coverage individually for these nine classes under test. For NUnit, we excluded branch coverage for the classes that were covered by the unit tests but were not directly a part of our target code under test. However, the branch coverage for those classes (excluded from the table) were the same for both CUTs and PUTs.

Since generalized tests often help cover more paths in the code under test, we found that test generalization helped to achieve effective increase in the branch coverage. For example, for `RegistrySettingsStorage`, Table 3 shows increase in branch coverage by 38%. We next present an illustrative example to show how test generalization helps increase branch coverage of the code under test. Figure 9 shows the `RemoveSetting` method and Figure 10 shows the existing CUT for `RemoveSetting`. On executing the

```
01: public void RemoveSetting(string settingName) {
02:  int dot = settingName.IndexOf( '.' );
03:    if (dot < 0)
04:     storageKey.DeleteValue(settingName, false);
05:    else {
06:     using(RegistryKey subKey = storageKey.OpenSubKey(
              settingName.Substring(0,dot),true)){
07:       if (subKey != null)
08:          subKey.DeleteValue(
                 settingName.Substring(dot + 1)); } }
09: }
```

Figure 9: `RemoveSetting` method whose coverage is increased by 60% due to test generalization.

```
01: public void CUT() {
02:  storage.SaveSetting("X",5);
03:  storage.SaveSetting("NAME","Charlie");
04:  storage.RemoveSetting("X");
05:  Assert.IsNull(storage.GetSetting("X"),
                 "X not removed");
06:  Assert.AreEqual("Charlie",
                 storage.GetSetting("NAME"));
07:  storage.RemoveSetting("NAME");
08:  Assert.IsNull(storage.GetSetting("NAME"),
                              "NAME not removed");
09: }
```

Figure 10: Existing CUT to test the `RemoveSetting` method.

test and analyzing the code portions that are not covered in the `RegistrySettingsStorage` class, we observed that the code inside the `else` block (Statements 5-8) was not covered by the unit test, i.e., the `false` branch was not covered. Figure 11 shows the PUT that was generalized from the existing CUT. On executing Pex with this PUT, we observed that the code portions not covered by the CUT, i.e., Statements 5-8 of the `RemoveSetting` method were covered by the new generated unit tests. This example shows that Pex was able to generate unit tests that covered both the `true` and `false` branches of the branching condition shown in Statement 3. Using the generalized PUT, two types of test inputs for the "`settingName`" parameter were generated, those that contained '.' and those that did not contain '.'. Therefore, in contrast to the 20% branch coverage achieved by executing the CUT, 80% branch coverage was achieved by executing the unit tests generated by Pex using the PUT.

For NUnit, the coverage of two classes is increase by 38% and 53%, respectively. In DSA and QuickGraph, for two namespaces, the branch coverage is increase by 1% and 2%, respectively. One major reason for not achieving a significant increase in the coverage for DSA and QuickGraph is that the existing CUTs already achieved a high branch coverage and PUTs help achieve more coverage than the existing CUTs. In summary, for the three subjects, generalizing the existing unit tests resulted in an overall average increase in coverage by 10%, 1%, and 1%, respectively.

5.3 RQ2: Defects

To address RQ2, we identify the number of defects detected by PUTs. Since we used the CUTs that are available with the sources, we did not find any failing CUTs, i.e., no defects are reported by the existing unit tests for any of the subjects. Therefore, any failing tests among the unit tests generated from PUTs are considered as new defects that are not detected by the CUTs. However, before confirming whether a failing unit test indicates a defect in the code under test, we manually verify that the generated test data is valid and the defect is not a false positive due to an ineffective PUT.

Through test generalization of the three subjects in our study, we found 13 new defects in the DSA application and 3 new defects in the NUnit application. Since we were sure that the resulting failing tests were due to defects in the code under test, we reported the fail-

```
01: public void PUT([PAUT]String[] name,
                           [PAUT]Object[] value) {
02:  ............
03:  for (int i = 0; i < name.Length; i++) {
04:     storage.SaveSetting(name[i], value[i]); }
05:  for (int i = 0; i < name.Length; i++) {
06:    if (storage.GetSetting(name[i]) != null) {
07:       storage.RemoveSetting(name[i]);
08:       PexAssert.IsNull(storage.GetSetting(name[i]),
                         name[i] + " not removed");
08:   }}}
```

Figure 11: Transformed PUT of the CUT shown in Figure 10.

```
//To test Remove item not present
01: public void RemoveCUT(){
02:  Heap<int> actual = new Heap<int> {2, 78, 1, 0, 56};
03:  Assert.IsFalse(actual.Remove(99));
04: }
```

Figure 12: Existing CUT to test the `Remove` method of `Heap`.

```
01: public void RemoveItemPUT(
             [PAUT]List<int> input, int item) {
02:  Heap<int> actual = new Heap<int> (input);
03:  if (input.Contains(item)) {
04:     ..... }
05:  else {
06:     PexAssert.IsFalse(actual.Remove(randomPick));
07:     PexAssert.AreEqual(input.Count, actual.Count);
08:     CollectionAssert.AreEquivalent(actual, input);}
09: }
```

Figure 13: Transformed PUT of the CUT shown in Figure 12.

ing tests on its hosting website and are waiting for the confirmation from the developers[9].

We next explain an example defect detected in the `Heap` class from the DSA application by test generalization. The `Heap` class is a heap implementation in the `DataStructure` namespace. This class includes methods to add, remove, and to heapify the elements in the heap. The `Remove` method of the class takes an item to be removed as a parameter and returns `true` when the item to be removed is in the heap, and returns `false` otherwise. Figure 12 shows the existing CUT that tests whether the `Remove` method returns `false` when an item that is not in the heap is passed as the parameter. On execution, this CUT passes exposing no defect in the code under test and there are no other CUTs in the test suite that test the behavior of the method. However, from our generalized PUT shown in Figure 13, a few of the generated unit tests failed exposing a defect in the `Remove` method. The test data for the failing unit tests had the following common characteristics: the heap size is less than 4 (the `input` parameter of the PUT is of size less than 4), the item to be removed is 0 (the `item` parameter of the PUT), and the item 0 was not already added to the heap (the generated value for `input` did not contain the item 0). When we manually analyzed the cause of the failing unit tests, we found that in the constructor of the `Heap` class, a default array of size 4 (of type `int`) is created to store the items. In C#, an integer array is by default assigned values zero to the elements of the array. Therefore, there is always an item 0 in the heap unless an input list of size greater than or equal to 4 is passed as parameter. Therefore, on calling the `Remove` method to remove the item 0, even when there is no such item in the heap, the method returns `true` indicating that the item has been successfully removed and causing the assertion statement to fail (Statement 6 of the PUT). However, this defect was not detected by the CUT shown in Figure 12 since the unit test assigns the heap with 5 elements (Statement 2) and therefore the defect-exposing scenario of heap size < 4 cannot be exercised.

This example defect detected in our study demonstrates the strength

[9]Reported bugs can be found at the DSA CodePlex website with defect IDs from 8846 to 8858 and the NUnit SourceForge website with defect IDs from 0 to 0.

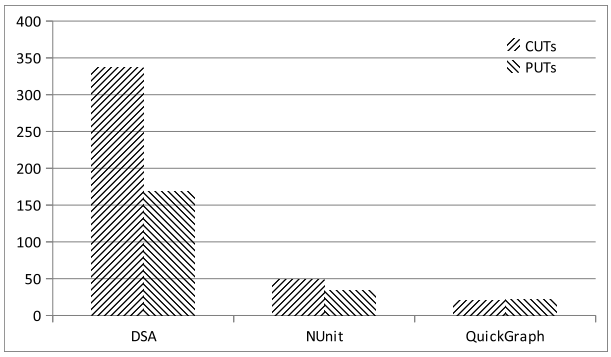

Figure 14: Comparison of the number of CUTs and PUTs

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

Table 4: Mapping of the number of CUTs and their transformed PUTs.

of generalized unit tests such as PUTs. The 16 defects reported in our study that were not detected by the existing unit tests show that PUTs an effective means for rigorous testing of the code under test.

5.4 RQ3: Test Code Maintenance

We next address RQ3 of whether test generalization can reduce the effort 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 lower the number of CUTs or PUTs, the less is the effort required in maintaining the test code. The reason is that whenever the behaviors of code under test is changed via code modifications, all failing tests need to be modified based on the new expected behavior of the code under test. Therefore, a lower number of PUTs can reduce the effort 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 lower number of PUTs with a high number of LOC does not help in reducing the effort required in maintaining the test code.

Figure 14 shows the comparison of the number of CUTs with PUTs for all subject applications. The x-axis shows the subject application and y-axis shows the number of CUTs or PUTs. In total, we generalized 407 CUTs to 224 PUTs that achieved higher branch coverage than CUTs and also detected new defects that are not detected by the CUTs. The figure shows that there is a significant reduction in the number of tests for the subjects DSA and NUnit. This reduction in the number of tests is because multiple CUTs are generalized to a single PUT. Table 4 shows the mappings between the number of CUTs and their transformed PUTs. For example, Row 1 shows that one CUT is transformed to one PUT in 129 occurrences. Similarly, the last row shows that 15 CUTs are transformed into a single PUT in one occurrence. Rows 2 and 3 show exceptional cases where a CUT is generalized to more than one PUT. Section 6 discusses more about these exceptional cases.

Figure 15 shows an example PUT, which is a result of gen-

```
//PAUT = PexAssumeUnderTest
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 15: PUT for the `AddFirst` method under test.

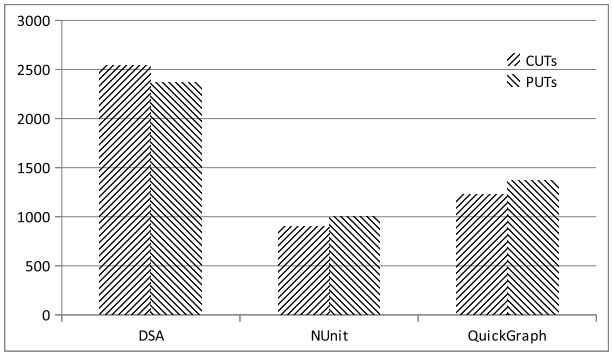

Figure 16: Comparison of Lines of Code of CUTs and PUTs.

eralizing four CUTs of the `SinglyLinkedList` class of DSA. The objective of all four CUTs is to test the `AddFirst` method (method under test) that adds an element to a list object of type `SinglyLinkedList`. These four CUTs verify different behaviors by adding one element or two elements to a list and by verifying whether the `head` and `tail` values of the list are updated correctly. We generalized all these four CUTs into a single PUT shown in Figure 15. The CUTs verify the behavior of the `AddTest` method by adding only a fixed number of elements (with fixed values) to the list. In contrast, our PUT can verify the behavior of the `AddFirst` method with a varied number of elements in the list.

Figure 16 shows the results of comparing the LOC of CUTs and PUTs. For DSA, the LOC of PUTs is less than the LOC of CUTs, whereas for the other two subjects, LOC of PUTs is slightly more than the LOC of CUTs. We identify that among new LOC written for PUTs, many statements are related to the additional `using` statements or new annotations added during our test generalization. On average, the LOC of PUTs is almost the same as the LOC of CUTs. Therefore, our results show that test generalization can help reduce the number of tests and thereby reduce the efforts in maintaining test code.

6. DISCUSSION

In our empirical study, we identify that it is difficult to generalize test oracles in a few cases. For example, consider the following CUT:

```
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 into a standard form. It is easy to identify the expected output for concrete strings such as `C:/folder1/.../folder2/.../file.tmp`. However, when the CUT is generalized with a parameter for the input string, it is challenging to identify the expected output. Although a developer can use our commutative diagram pattern and provide an alternative implementation for the `Canonicalize` method, the amount of required effort could be higher than the effort required to write the implementation of the actual method under test. In our empirical study, we still generalize these CUTs but do not generalize the test assertions, and instead replace assertions with statements that print outputs suitable for manual review, i.e., use the Manual Output Review pattern in our proposed patterns. These PUTs can still help in detecting defects related to exceptions.

In our results related to test-code maintenance, there are a few cases where the number of PUTs are more than the number of CUTs. The reason is that Pex is not able to generate any CUT from the transformed PUT since Pex reached the upper-bound of the number of explored branches (by default, pre-configured to avoid long running of Pex). Although we tried to set higher upper-bound values, Pex was still not able to generate CUTs from those PUTs. Therefore, we performed a partial generalization by splitting the CUT into multiple PUTs.

We conducted our empirical study as third-party testers since we do not have the knowledge of the subject applications. We expect that our test-generalization results can be much better if the test generalization is performed by the developers of these subject applications. The reason is that developers can incorporate their domain knowledge during test generalization to write more effective PUTs.

7. RELATED WORK

Pex [22, 23, 24] accepts PUTs and uses dynamic symbolic execution to generate test inputs. Other existing tools such as Parasoft Jtest [4] and CodeProAnalytiX [5] adopt the design-by-contract approach [17] 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. [19] 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 unit tests where assertions should hold for all inputs that satisfy the assumptions specified in the unit tests. 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. In contrast to our study, their study does not provide a systematic procedure of writing generalized PUTs or show empirical evidence of benefits of PUTs as shown in our study.

There are existing approaches [18, 7, 13] 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 [18] 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, since 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, when Pex's demand-driven strategy faces challenges.

In our previous work [16], we presented an empirical study to analyze the use of parameterized mock objects 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 our other previous work

with PUTs [25], 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 systematic procedure of retrofitting CUTs for parameterized unit testing.

8. THREATS TO VALIDITY

The threats to external validity primarily include the degree to which the subject programs, defects, and CUTs are representative of true practice. The subject applications used in our empirical study range from small-scale to medium-scale applications that are widely used in software industry as shown by their number of downloads. We tried to alleviate the threats related to detected defects by inspecting the source code and by reporting the defects to the developers of the application under test. These threats could further 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 transforming CUTs to PUTs. Our study results can be biased based on our experience and knowledge of the subject applications. These threats can be reduced by conducting more case studies with more subject applications and other human subjects. The results in our study can also vary based on other factors such as the effectiveness of transformed PUTs and test-generation capability of Pex.

9. CONCLUSION

Recent advances in software testing introduced Parameterized Unit Tests (PUT), which are a generalized form of conventional unit tests (CUTs). With PUTs, developers do not need to provide test data (in PUTs), which is generated automatically using the dynamic-symbolic-execution approach. Although PUTs are more beneficial than CUTs in assuring the quality of the code under test, PUTs are still not widely adopted in software industry. In practice, many developers still rely on CUTs as a primary means for ensuring the quality of their developed software. These CUTs can be transformed to PUTs, referred to as test generalization, to exploit the benefits of PUTs. In this paper, we proposed the first systematic procedure that assists developers in performing test generalization. In our study, we also showed that test generalization reduces 407 CUTs to 224 PUTs (45%), thereby reducing the effort of maintaining test code. Along with achieving higher branch coverage (a maximum increase of 53% for one class under test and 10% for one application under analysis), test generalization detected 16 new defects that are not detected by existing CUTs. A few of these defects are quite complex and are hard to be detected using CUTs. Given these benefits of test generalization, we expect that developers in software industry can use our systematic procedure and transform their CUTs to PUTs to exploit the benefits of PUTs, and thereby increase the quality of their developed software.

10. REFERENCES

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

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

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

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

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

[6] Pex - automated white box testing for .NET, 2009. `http://research.microsoft.com/Pex/`.

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

[8] J. de Halleux. Quickgraph, graph data structures and algorithms for .NET, 2006. `http://quickgraph.codeplex.com/`.

[9] 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`.

[10] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. PLDI*, pages 213–223, 2005.

[11] Granville and L. D. Tongo. Data structures and algorithms, 2006. `http://dsa.codeplex.com/`.

[12] C. P. Jamie Cansdale, Gary Feldman and M. C. Two. NUnit, 2002. `http://nunit.com/index.php`.

[13] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proc. TACAS)*, pages 553–568, 2003.

[14] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[15] T. Mackinnon, S. Freeman, and P. Craig. Endo-testing: unit testing with mock objects. pages 287–301, 2001.

[16] 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 *Proc. AST, Business and Industry Case Studies*, pages 149–153, 2009.

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

[18] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proc. ICSE*, pages 75–84, 2007.

[19] D. Saff, M. Boshernitsan, and M. D. Ernst. Theories in practice: Easy-to-write specifications that catch bugs. Technical Report MIT-CSAIL-TR-2008-002, MIT Computer Science and Artificial Intelligence Laboratory, 2008. `http://www.cs.washington.edu/homes/mernst/pubs/testing-theories-tr002-abstract.html`.

[20] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *Proc. ESEC/FSE*, pages 263–272, 2005.

[21] S. Thummalapenta, T. Xie, N. Tillmann, P. de Halleux, and W. Schulte. MSeqGen: Object-oriented unit-test generation via mining source code. In *Proc. ESEC/FSE*, pages 193–202, 2009.

[22] N. Tillmann and J. de Halleux. Pex - white box test generation for .NET. In *Proc. TAP*, pages 134–153, 2008.

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

[24] N. Tillmann and W. Schulte. Unit tests reloaded: Parameterized unit testing with symbolic execution. *IEEE Software*, 23(4):38–47, 2006.

[25] T. Xie, N. Tillmann, P. de Halleux, and W. Schulte. Mutation analysis of parameterized unit tests. In *Proc. Mutation*, pages 177–181, 2009.