An Empirical Study of Test Generalization in NUnit Framework

author1
Department of Computer Science
North Carolina State University
Raleigh, NC

author2
Department of Computer Science
North Carolina State University
Raleigh, NC

Abstract

Owing to the significance of unit testing in software development life cycle, there are several existing approaches on automating the generation of conventional unit tests. However, these approaches often fail to generate a minimal set of test cases for ensuring a high code coverage and manually writing such test cases is a labor-intensive task. To address these preceding issues with conventional unit tests, parameterized unit tests (PUT) are introduced where programmers can describe the expected behavior or specifications using symbolic values. Pex, an automatic test generation tool, accepts these PUTs and generates a minimal set of conventional unit tests that ensure a high code coverage. This process of converting conventional unit tests into PUTs is referred as test generalization. In our empirical study, we used an open source C# project called NUnit to study the benefits of test generalization. In our study, we found that the test generalization has increased the block coverage by 9.68% (on average) and also detected 7 new defects.

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. As unit testing has been widely adopted by the software industry, several automatic unit test generation tools such as Parasoft JTest [6] or Agitar JUnit factory [4] are developed to automatically generate unit tests for the code under test. These tools generate conventional unit tests that do not accept any parameters and include two major parts: test inputs and test oracles. In general, test inputs include method calls that modify the state of the object, whereas test oracles include method calls (often the state-preserving method calls) that verify the modified state. However, these tools often do not create a comprehensive minimal set of unit tests [11] that can guarantee a high code coverage. On the other hand, writing all unit tests manually is a prohibitively expensive task.

Parameterized Unit Tests (PUTs) are a new advancement in unit testing that accept parameters unlike conventional unit tests, which do not accept any parameters. In particular, PUTs represent the expected behavior or specifications of the method under test with symbolic values. Pex [10], a Microsoft test generation tool, accepts these PUTs and generates a minimal set of conventional unit tests that tries to achieve a high code coverage of the method under test. Conventional unit tests can be treated as instances of PUTs. Pex uses an approach called symbolic execution [8, 9] for generating conventional unit tests from PUTs. Initially, Pex explores the program with random values and collects constraints along the path executed in the method under test. Later, Pex explores alternate paths in the method under test by systematically flipping the captured constraints and generates concrete values (using a constraint solver) that can cover those alternate paths. For each set of concrete values that explore a new path, Pex generates a conventional unit test case.

Although PUTs are more effective and can often achieve higher code 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 the preceding challenge, we use the following procedure where we initially take existing conventional unit tests (or write if the conventional unit tests are not already available). We 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. We observe that there can be several challenges in directly executing transformed 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, Pex provides the factory method feature that lets users provide methods for
We next explain our procedure for the test generalization of conventional unit tests using Pex. 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 case is to verify the behavior of the class `MemorySettingsStorage`, which is primarily used for storage and retrieval of global values. To generalize the current test case, we initially identify the concrete values used in the test case. For example, the test case 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 regenerate concrete values based on the constraints encountered in different paths of the method under test (MUT).

We initially verify the conventional unit test to identify which PUT pattern 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 [2]. If the conventional unit test does not fall into any of the pre-defined categories, we define new patterns.

Figure 2 shows the skeleton of the PUT after generalizing concrete values. Our PUT accepts three parameters: instance of `MemorySettingsStorage`, name of the setting, and its value. In the conventional unit test, the `SaveSetting` method is used for both `integer` and the `String`. Therefore, the test requires two method calls shown in Statements 4 and 5 (Figure 1). 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 advantages of PUT that help 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.

Pex can effectively handle primitive-type parameters such as `String` or `integer`. However, Pex faces challenges in generating values for non-primitive arguments such as `st` in our PUT skeleton. These non-primitive arguments often require desirable 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. The primary challenge in constructing desirable states for non-primitive arguments is to construct a sequence of method calls that create and mutate objects. Pex provides the concept of factory methods to let users assist Pex in the generation of 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. 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 test generalization process is to define assumptions. For example, without any assumptions provided, Pex by default generates null values for the PUT arguments. To address the preceding issue, we annotate the methods with a tag `PexAssumeUnderTest`, 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 an assumption that the setting to be added should not already exist in the storage. We add these additional assumptions to the PUT such as Statement 4 (in Figure 4) in `SaveAndLoadSettingsPUT1`. Figure 4 shows the complete PUT for the conventional unit test.

### 3. 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 test fixtures and test methods. The rationale behind choosing NUnit for our test generalization is the amount of documentation available on the project homepage and 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 53KLOC. The test code includes 264 source files with 25KLOC. This significant amount of test code made this project a good subject for our test generalization 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 7.2KLOC with 72 files and 326 methods. The test files, test methods, and test LOC account to 32, 335, and 3.4KLOC, respectively. The motivation for choosing the Util package is not just its significance but also the existence of non-trivial code fragments such as the example shown in Figure 5, that enable us to study the test generalization process.

### 4. Benefits of Test Generalization

In the Util project of the NUnit framework, we found that there are 32 test classes with 335 test methods (conventional unit tests). We generalized 10 test classes that include 57 test methods. We transformed these 57 conventional unit tests into 49 PUTs. Although we are able to generalize most of the test methods in these classes, we found a few percentage of methods are not amenable for the test generalization.

Table 2 shows the results of our test generalization. Column “Test Class” shows the name of the test class.
Figure 5. A non-trivial code fragment from the Util package of the NUnit framework.

```csharp
01: public static string Canonicalize( string path )
02: {
03:     ArrayList parts = new ArrayList(path.Split(04:         DirectorySeparatorChar, AltDirectorySeparatorChar));
05:     for( int index = 0; index < parts.Count; )
06:     {
07:         string part = (string)parts[index];
08:         switch( part )
09:         {
10:             case "*:"
11:                 parts.RemoveAt( index );
12:                 break;
13:             case "..":
14:                 parts.RemoveAt( index );
15:                 if ( index > 0 )
16:                     parts.RemoveAt( --index );
17:                     break;
18:             default:
19:                 index++;
20:             break;
21:         }
22:     }
24:                     (string[])parts.ToArray( typeof( string ) ) );
25: }
```

4.1 Coverage

As generalized test cases often help cover more scenarios, we found that test generalization helped to have a significant increase in the coverage as shown in Table 2. For example, test generalization of the RegistrySettingsStorage class shows an increase in the coverage of 45.24%. In addition, test generalization also helped achieve coverage of new blocks that are not covered by the 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). However, we found that in some scenarios, PUTs achieved a lower coverage compared to the conventional unit tests. These scenarios are primarily due to the limitations in the current Pex tool. We describe our limitations in Section 8.

4.2 Defects

We found 7 new defects after generalization that are not detected by the 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 checking for the count or 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 a subkey and the other as a subkey 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
### Table 2. Benefits of Test Generalization.

<table>
<thead>
<tr>
<th>Test Class</th>
<th>Test Methods</th>
<th>% Coverage</th>
<th>Avg. New Blocks</th>
<th>#Defects</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUnitProjectSave.cs</td>
<td>3</td>
<td>1</td>
<td>35.71</td>
<td>10</td>
</tr>
<tr>
<td>NUnitRegistryTests.cs</td>
<td>2</td>
<td>5</td>
<td>33.33</td>
<td>1</td>
</tr>
<tr>
<td>TestAgentTests.cs</td>
<td>5</td>
<td>5</td>
<td>100.00</td>
<td>0</td>
</tr>
<tr>
<td>RegistrySettingsStorageTests.cs</td>
<td>6</td>
<td>5</td>
<td>83.33</td>
<td>0</td>
</tr>
<tr>
<td>MEMORYSettingsStorageTests.cs</td>
<td>6</td>
<td>4</td>
<td>66.67</td>
<td>2</td>
</tr>
<tr>
<td>PathUtilTests.cs</td>
<td>7</td>
<td>3</td>
<td>42.86</td>
<td>0</td>
</tr>
<tr>
<td>RecentFilesTests.cs</td>
<td>22</td>
<td>21</td>
<td>95.45</td>
<td>14</td>
</tr>
<tr>
<td>ServerUtilityTests.cs</td>
<td>2</td>
<td>2</td>
<td>100.00</td>
<td>0</td>
</tr>
<tr>
<td>SettingsGroupTests.cs</td>
<td>5</td>
<td>5</td>
<td>100.00</td>
<td>2</td>
</tr>
<tr>
<td>ProcessRunnerTests.cs</td>
<td>0</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Figure 6. Single PUT constructed from five conventional tests.

```csharp
[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(() => MaxValue == recentFiles.MaxFiles);
}
```

5 Categorization of Conventional Unit Tests

We used test patterns proposed by Halleux and Tillmann [7] to generalize the conventional unit tests. We found that a few test patterns are predominantly applicable while others are helpful in a few specific cases. 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 categorized the amenable test cases into the patterns that were applied to generalize them. However, apart from a few test cases not amenable to test generalization for the reasons discussed in Section 6, a few other test cases were not easy to generalize using existing patterns. We further propose new patterns that can help writing test code. Section 5.1 provides details on the patterns used for test generalization and Section 5.2 provides details on the new patterns with examples.

5.1 Conventional Unit Tests Amenable to Test Generalization

Test generalization also helped significantly reduce the test code as shown in Table 2. 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 6 shows an example PUT of the pattern type *Cases* that combined five conventional unit tests. Each conventional unit test verifies 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 the detected constraints.

4.3 Test code

"\0") and returned a message, "expected 2, got 1". This defect shows that the failure was possibly due to missing check on invalid characters.
2. **Rarely used:** We classify a pattern under this category if that pattern was not used in generalizing the test cases. We found that 5 out of the 14 Patterns were not used in our test generalization; from 2.9 to 2.14. 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 2.13 and 2.14 are applied in the context of regression testing. However, in our current study, our focus is not on regression testing. One more reason is that the current study focuses on generalizing existing conventional unit tests rather than developing new PUTs. As these conventional unit tests do not have any in-out parameters, we could not use patterns such as Pattern 2.12. In future work, we plan to write more PUTs for increasing the code coverage. We expect that these rarely used patterns shall be helpful in our future work.

5.2 New Test Patterns For Test Generalization

We next describe two new test patterns that we found useful during our test generalization.

- **Unique value generation**
  
  **Purpose:** To generate unique values when the input is a collection of values instead of a single value.
  
  **Motivation:** During test generalization, we found four scenarios where the required object state can be created only when a unique set of elements is available. In such scenarios, we have to write additional code to ensure that Pex generates only unique values. For example, in the following example, 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.

  ```java
  //FAULT: PexAssumeUnderTest
  01: public void SubstorageSettingsPUT1(
    [PAUT]String subName, [PAUT]String[] name)
  02:{
    03: PexAssume.IsTrue(name.Length == value.Length);
    04: PexAssume.IsNotNull(value);
    05: for (int j = 0; j < name.Length; j++) {
      06:    for (int i = 0; i < name.Length; i++) {
        07:      PexAssume.IsFalse(name[i].Equals(name[j]));
      08:    }
    09:  }
  10:  }
  
  Proposed Pattern: When the input is of the type collection or an array, we propose a new test pattern called `PexGenerateUnique` that can inform Pex to generate unique values. For example, applying the proposed pattern to the preceding example results in the following code example. We also assume that `PexGenerateUnique` subsumes the properties of `PexAssumeUnderTest`.

  ```java
  //FAULT: PexGenerateUnique
  01: public void SubstorageSettingsPUT1(
    [PAUT]String subName, [PGenerateUnique]String[] name)
  02:{
    03:  PexAssume.IsNotOdd(value);
    04:  PexAssume.IsTrue(name.Length == value.Length);
    05:  ......}
  
  For the four scenarios that required generation of unique values, the proposed pattern can be applied.

- **Pick a random value from the pool**
  
  **Purpose:** To reuse generated values by maintaining a pool and randomly picking from those values.
  
  **Motivation:** When converting conventional unit tests to PUTs, we needed to parameterize a unit test that requires to verify whether adding a `RegisterKey` to a `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 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 8 shows an application of our proposed pattern.
00:[PexMethod{}]
 01:public void TestClearRoutinesPUT([PAUT]String[] key) {
 02: PexAssume.IsTrue(key.Length > 1);
 03: for (int j = 0; j < key.Length; j++) {
 04: PexAssume.IsNotNull(key[j]);
 05: }
 06: NUnitRegistry.TestMode = true;
 07: using (RegistryKey mainKey =
 08: NUnitRegistry.CurrentUser){
 09: //enabling appending values to a list
10:  PexStore.Pool("keys","mainKey");
11:  for (int j = 0; j < key.Length; j++) {
12:   RegisterKey parentKey =
13:    PexStore.Pick("keys") as RegisterKey;
14:   RegistryKey subKey =
15:    mainKey.CreateSubKey(key[k - 1]);
16:  PexStore.Pool("keys",subKey);
17:  }
18:  NUnitRegistry.ClearTestKeys();
19:  PexAssert.IsTrue(mainKey.SubKeyCount == 0);
20: }
21: }

Figure 7. Distribution of test patterns for 49 PUTs.

The Canonicalize method in PathUtils accepts an input string and uses a complex procedure to transform the 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.

7. Helper Techniques for Test Generalization

We next describe several helper techniques provided by Pex and used in our test generalization process.

7.1. Factory Methods

Pex provides the factory method helper technique to enable users assist Pex in generating method-call sequences for constructing desirable object states for non-primitive arguments. In our test generalization, we defined 5 factory methods for assisting Pex in generating desirable object states. We found that Pex's heuristic demand-driven 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. Instead, we observed that Pex faces challenges if the constructor does not explicitly state the argument types or the method under test expects a specific object state for the argument. The first issue can happen when the constructor accepts an interface or an abstract class. As example is the RecentFilesService class provided by Pex. The constructor of this class accepts an ISettings interface, which is used to access all user settings. In the util project,
the `SettingsGroup` class implements this interface. Without any factory method, Pex generated only one method-call sequence that accepts a `null` value, which is not useful in our scenario. We expect that Pex can be enhanced to address this issue, where Pex can determine the classes implementing the interface automatically and instantiate those classes for constructing method-call sequences. In the current scenario, only the `SettingsGroup` class implements the `ISettings` interface. Using this information, Pex can be modified to generate the following method-call sequence for constructing an object of `RecentFilesService`.

```java
MemorySettingsStorage mss = new MemorySettingsStorage();
SettingsGroup sg = new SettingsGroup(mss);
//SettingsGroup implements the interface ISettings.
RecentFilesService rfs = new RecentFilesService(sg);
```

Regarding the second issue of generating method-call sequences for constructing desirable object states (unlike generating an object with any state), we expect the problem is still an open issue in object-oriented testing. For example, for the PUT shown in Figure 4, consider that the desirable object state is that the `storage1` object should contain four elements. We can generate such object states using the factory method shown in Figure 3.

### 7.2. Mock Objects

Mock objects help to test features in isolation by replacing unrelated features with mock objects. In our current study, we have not found any scenarios yet where we needed mock objects. In future work, we expect that we will use mock objects to increase the coverage achieved by the PUTs. For example, in the `SettingGroupTests` test class, we achieved a coverage of only 72.22%, which is more than what the conventional unit test achieved. However, we could not achieve 100% coverage because the uncovered code portion can be covered only when an exception is thrown. In future work, we will replace the existing call with a mock object that can throw necessary exceptions.

### 7.3. New Assertions

In our current study, we have not added any new assertions. However, we did generalize a few assertions to handle more generic behavior.

### 8. Limitations of Pex or PUTs

One of the major limitations of PUTs is that PUTs require more efforts from programmers compared to conventional unit tests. As PUTs are more generic compared to conventional unit tests, programmers require additional expertise in developing such PUTs.

Regarding Pex limitations in generating conventional unit tests from PUTs, often the values generated by Pex do not resemble the actual values used in program executions. One workaround can be that Pex can identify possible values used at other locations within in the project under test and reuse those values.

### 9. Conclusion

We conducted an empirical study to investigate the benefits of test generalization, which is a process of transforming conventional unit tests into parameterized unit tests (PUTs). In our study, we generalized 57 conventional unit tests in the NUnit framework to develop 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. We also identified categories of conventional test cases that are not amenable for test generalization and proposed new PUT patterns. In future work, we plan to generalize more conventional unit tests to explore further the strengths and weaknesses of test generalization.

### References