An Empirical Study of Parameterized Unit Test Generalization in xUnit Framework

Xusheng Xiao
Department of Computer Science
North Carolina State University
Raleigh, NC 27695-8206
xxiao2@ncsu.edu

Di Lu
Department of Computer Science
North Carolina State University
Raleigh, NC 27695-8206
dlu@ncsu.edu

Abstract

In today's software development process, testing has become an irreplaceable part, which should be performed through the whole software development life cycle. Unit testing is considered a fundamental part of software testing. In this field, there already exist several tools that can automatically generation conventional unit tests. However, some of these tools can not ensure high code coverage unless testers write some of tests manually, which would be a burdensome and tedious task if the project size is large.

Pex, an automated white box testing tool developed by Microsoft, introduced an approach called parameterized unit tests (PUT), hoping to solve the issues above. With PUTs, Pex can generate conventional unit tests with different input values automatically, which not only saves testers from repeating tedious works, but also helps testers achieve higher code coverage.

To do the empirical study of PUTs, we propose a two-phase approach: generalize PUTs from existing conventional unit tests and write new PUTs to cover the blocks that are not covered by test generalization. We used xUnit and xUnit.extension projects, which belong to xUnit framework for programmer unit tests, to study the performance of our approach in a real project. Our study result shows that in the test generalization, the block coverage has increased from 95.45% to 96.81% and one new defect was found. Furthermore, writing new PUTs achieves 30.83% coverage increase in average and results in 100% coverage of xUnit project.

1. Introduction

In the software development process, testing plays an important and irreplaceable role, which should be performed through the whole process. As part of the testing process, unit testing is a software verification and validation approach in which the smallest testable parts of an application, called units, are individually and independently tested for appropriate operation. Additionally, unit tests produced in unit testing also reflect customer requirements and serve as a specification of the functional code and a technical documentation. In industry, unit testing is adopted widely due to the popularity of software development methods, such as extreme programming (XP) [1] and test-driven development (TDD) [7] and test execution frameworks, e.g., JUnit [16], NUnit [17] and xUnit [2]. A test suite produced by unit testing with high code coverage gives confidence in the correctness of the tested code. However, writing unit tests to achieve high coverage can be time-consuming and tedious and these test execution frameworks only automate the test executions. As the size and complexity of software system increases, these testing frameworks that are designed specifically for parameterless conventional unit tests do not scale well. To address this problem, several automatic unit test generation tools such as Parasoft JTest [4] or jCUTE [13] can automatically generate conventional unit tests. These tools, nevertheless, cannot guarantee high code coverage, unless testers manually write some tests.

Parameterized unit tests (PUT) [15] extend the current industry practice of using conventional unit tests by accepting parameters in the test method. In particular, the expected behavior or specifications of the method under test are represented with symbolic values in PUTs. In this way, PUTs state the intended program behavior for entire classes of program inputs, and not just for one particular input. Hence, PUTs are more general specifications than conventional unit tests. Given a PUT with parameters, a test-generation tool, such as Pex [14], can automatically generate tests with concrete inputs for the parameters to achieve high coverage.

Pex, a white box test input generation tool developed by Microsoft Research, explores the behaviors of a PUT using a technique called dynamic symbolic execution [9, 11]. Dynamic Symbolic Execution (DSE) is a variation of symbolic execution, which systematically explores feasible paths of the program under test by running the program with different test inputs to achieve high structural coverage. It collects the symbolic constraints on inputs obtained from predicates in branch statements along the execution and relies on a constraint solver to solve the constraints and generate new test input for exploring new path. For each set of concrete test input that leads to a new path that achieves new coverage, Pex generates a corresponding conventional unit test.

Although PUTs are more generalized than conventional unit tests and can often achieve higher structural coverage, such as basic block coverage and branch coverage [6, 8], writing PUTs requires more efforts than conventional unit tests. A well written PUT requires the abstraction of the behaviors of the method under test and provides proper parameters, assumptions, and assertions. It is not a trivial work to write PUTs from scratch for a large code base, especially to provide proper assertions for the unfamiliar code base. To do the empirical study of PUTs, we propose a two-phase approach: generalize PUTs from existing conventional unit tests and write new PUTs to cover the blocks that are not covered by the phase of test generalization based on the analysis of coverage report and original tests.

The phase of test generalization starts with introducing parameters for the arguments or the receiver object of the method under test. Then the proper assumptions and assertions would be provided to constrain the inputs and verify the results. For several conventional unit tests that test the same method with different inputs, we try to merge them into one PUT. In addition to these normal steps, there are two more issues we need to address. First, if the arguments include non-primitive objects, Pex may not be able to generate proper method-call sequences to create or modify the objects into the desired states. To address this issue, we can provide factory methods in which we construct the desired object states using simpler or primitive parameters. Second, if there are some objects interacting with environments, such as file system, database, or reflection, we can provide parameterized mock objects [12, 14] to mock the behaviors of these objects. This can assist Pex to get around the environment dependency problems and generate proper test inputs to achieve high coverage.

After test generalization, we write new PUTs that target at the uncovered blocks from the phase of test generalization. Based on the analysis of coverage report and original tests, we can identify the uncovered blocks from the phase of test generalization and find out the causes for the uncovered blocks. Using this analysis and the knowledge obtained from the phase of test generalization, we can write new PUTs with proper assumptions and assertions to achieve higher coverage.

In our study, we use the xUnit framework [2], which is built for programmer unit testing and include many features that can help programmer write clearer tests. We choose the xUnit project and xUnit.extension project to carry out our empirical study. In the phase of test generalization, the total block coverage increase from 95.45% to 96.81% and one new defect was found. In the phase of writing new PUTs, the results shows that the total coverage is increased by 2.91%, resulting in 100% coverage of xUnit project.

The rest of the paper is structured as follows: Section 2 illustrates the different methods for generalizing a new PUT with examples. Section 3 describes the characteristics of xUnit framework. Section 4 discusses the benefits of test generalization and writing new PUTs. Section 5 shows the categorization of conventional unit tests and added PUTs based on the PUT patterns and discuss their used frequency. Section 6 presents the helpful techniques for generalizing PUTs. Section 7 discusses the limitations of Pex and PUTs and Section 8 concludes.

2. Example

2.1 Simple Example

In this section, we are going to explain how to convert existing conventional unit tests into PUTs. The xUnit test to be generalized is shown in Figure 1.

```
public void CanSearchForNullInContainer()
{
    List<object> list = new List<object> { 16, "Hi there" };
    Assert.DoesNotContain(null, list);
}
```

Figure 1. Example conventional unit test from xUnit project.

It is not difficult to tell that this test is trying to verify the behavior of `Assert.DoesNotContain` when dealing with null value. As the assertion part is the test target, when we try to convert this unit test into PUT, we should still use the `Assert` Class of xUnit to make assertion.

The example in Figure 1, is a typical example of Triple-A (Arrange, Act, Assert) pattern [10]. Given this, we can set up a PUT by replacing the constant string value with symbolic value. Then we will have a very simple PUT, as shown in Figure 2.

```
[PexMethod]
public void TestDoesNotContainPUT (List<object> list)
{
    Assert.DoesNotContain(null, list);
}
```

Figure 2. Simple PUT generated from example unit test shown in Figure 1.

However, it is just a skeleton PUT. If we let Pex explore this PUT, we would encounter a lot of troubles. For example, Pex would try to crash the test by generating a null value for the list object. Therefore, an important step of generating PUT is to define assumptions. With the proper assumptions, we can tell Pex what we want to do or what we don't want to do, by adding assumptions. For example, if we don't want the object under test to be null, we can add an assumption, saying that the object under test is not null. Specifically, in this test, we also don't want list contain any null value. So we can add an assumption at the beginning of the test. By making assumption, we change the test pattern from 2.1 (Triple-A) to 2.2 (Assume, Arrange, Act, Assert). A complete PUT is shown in Figure 3.

```
[PexMethod]
public void TestDoesNotContainPUT (List<object> list)
{
    PexAssume.IsTrue(list != null);
    PexAssume.IsFalse(list.Contains(null) );
    Assert.DoesNotContain(null, list);
}
```

Figure 3. Complete PUT generated from the example unit test shown in Figure 1.

By exploring this PUT, Pex can automatically generate conventional tests with different input values to cover as many different situations as possible, which can lead to higher code coverage.

2.2 Factory Method

When the generalized PUT contains non-primitive arguments, Pex may not be able to generate proper method-call sequence to create or modify the object to the desired state. Figure 4 shows a PUT which has a non-primitive object parameter, `AssertException`. The intention of this PUT is to make sure that `AssertException` preferred `UserMessage` to normal `Message`. However, when Pex explores this PUT directly, it cannot generate any test due to the difficulties of creating an instance of `AssertException`.

To address this issue, Pex let user define a factory method that can be used to create and modify objects into desired states. Normally, a factory method receives simpler or primitive parameters and produces a desired instance of the object based on these parameters.

```
[PexMethod]
public void TestAssertExceptionPUT(AssertException ex)
{
    PexAssume.IsTrue(ex != null)
    PexAssert.AreEqual(ex.UserMessage, ex.Message);
}
```

Figure 4. A PUT that contains a non-primitive object argument.

Our factory method for the PUT in Figure 4 is showed in Figure 5. This factory method simply accepts a string parameter and constructs an instance of `AssertException` by passing in this string parameter. In fact, an instance of `AssertException` has several more properties to set, such as the properties of `SerializationInfo` and `StreamingContext`. These properties are again non-primitive objects, which explain why Pex cannot figure out how to create an instance for it. However, except the property of `UserMessage`, all of these properties are not related to the PUT. Thus, a simple factory method which only accepts a string parameter is fine enough to meet the needs of our PUT and we can just ignore other complex properties.

```
[PexFactoryMethod(typeof(AssertException))]
public static AssertException Create(string userMessage
 /*,SerializationInfo info_serializationInfo,
    StreamingContext context_streamingContext*/
)
{
    var assertException = new AssertException(userMessage);
    /*assertException.GetObjectData(info_serializationInfo,
          context_streamingContext);*/
    return assertException;
}
```

Figure 5. A factory method for PUT in Figure 4

2.3 Parameterized Mock Objects

In the unit testing of object-oriented program, mock objects are used to simulate the behavior of real objects, so that the tests can still be executed when real objects are impractical or impossible to incorporate into the testing, such as file system, database and so on. The factory method in Figure 6 illustrates the need of using mock objects.

In this example, the method parameter that implements interface `IMethodInfo` in the real code base is the wrapper object of `MethodInfo`, which is the object obtained from `System.Reflection`. To create such an object in different states, we need to figure out how to use reflection mechanism and write lots of tricky code in the PUT, which is a non-trivial

```
[PexFactoryMethod(typeof(TheoryCommand))]
public static TheoryCommand Create(IMethodInfo testMethod,
                                   string displayName, object[] parameters)
{
     var theoryCommand = new TheoryCommand(testMethod,
                                           displayName, parameters);
     return theoryCommand;
}
```

Figure 6. A factory method that needs a mock object.

job and makes the PUT hard to understand. To better solve this problem, we can mock this object and assist Pex to generate proper tests. Many mock frameworks, such as NMock [3] and Rhino Mocks [5], have been provided to ease the way to build and use mock objects. But all these frameworks require testers to manually state the return value of each methods of the mock object, which is tedious and time-consuming. Pex provides the concept of parameterized mock object, which is a better way to address this problem. By using parameterized mock object, we can turn the return values of the methods of the mock object into symbolic values, which can be used by Pex to collect the constraints encountered during the exploration and automatically generate different values for reaching higher coverage. Part of our parameterized mock object for `IMethodInfo` is showed in Figure 7.

```
00:  public class MMethodInfo : IMethodInfo
01:  {
       …… // other methods
02:     public bool HasAttribute(Type attributeType)
03:     {
04:            PexAssert.IsNotNull(attributeType);
05:            var call = PexChoose.FromCall(this);
06:            return call.ChooseResult<bool>();
07:     }
08:     public void Invoke(object testClass, params object[] parameters)
09:     {
10:          PexAssert.IsNotNull(testClass);
11:          PexAssert.IsNotNull(parameters);
12:          var call = PexChoose.FromCall(this);
13:          if (call.ChooseThrowException())
14:              throw call.ChooseException(false, new[] {typeof
                                          (TargetInvocationException)});
15:          Type.GetType(TypeName).GetMethod(Name).
                                          Invoke(testClass, parameters);
16:      }
17:  } //  end of class
```

Figure 7. Example of parameterized mock object

As we can see in Figure 7, the method `HasAttribute` calls `PexChoose.FromCall` to obtain a call object and use it to choose a result object of Boolean type. In this way, Pex can generate a symbolic value for this Boolean object and decides its value based on the constraints collected during the exploration. This is a typical example of the *Parameterized Mocks* pattern. The Invoke method starting in Line 8 demonstrates the power of Pex to deal with throwing exception in a method of a parameterized mock object, which belongs to the pattern *Parameterized Mocks With Negative Behavior*. By exploring the program under test, Pex can generate different tests in which a specific type of exception is thrown for covering the catch block if there is some.

To prevent invalid parameters, both of the methods showed in Figure 7 have the assertions over the arguments to ensure the correctness of the input parameters.

3. Open Source Project Under Test

xUnit is a testing framework which is built for programmer unit testing, specifically Test-Driven Development, but can also be very easily extended to support other kinds of testing, such as automated integration tests or acceptance tests. It includes some different features based on the lessons learnt from other NUnit framework: “single object instance per test method”, “no [Setup] or [Teardown]” and “no [ExpectedException]”.

Its popularity, code base size and large number of unit tests make it a suitable subject for our empirical study of PUTs and its unit tests can serve as the specification of the behaviors of the framework. The source code of xUnit framework includes 24K lines of code (LOC) and 549 unit tests. Table 1 shows the detailed code metrics of xUnit framework. For the purpose of demonstrating our approach, we pick up the core module, xUnit package, and the extension module, xUnit.extensions package, for our generalization and comparison.

Table 1 Characteristics of xUnit Framework

Attribute	**Value**
Total LOC	24809
xUnit LOC	4789
xUnit.extensions LOC	1295
#Total Tests	549
#xUnit Tests	310
#xUnit.extensions Tests	50

The core module, xUnit package, has the largest amount of unit tests among all the packages, which can be used to illustrate different patterns of the generalization. The extension module, xUnit.extensions package, includes some non-trivial tests which accept arguments that implement specific interfaces and interact with the reflection mechanism of C#. By transforming these tests, we can show how we

Table 2 Benefits of Test Generalization

Test Class	Unit Tests			CUT Block Coverage		PUT Block Coverage		New Block
	#CUT	#NA	#PUT	C/T	%Cov	C/T	%Cov	
EqualTests.cs	45	6	34	74/77	96.10	76/77	98.70	2
AssertExceptionTests.cs	2	0	2	3/3	100	3/3	100.00	0
ContainsTests.cs	9	0	9	52/84	61.90	52/84	61.90	0
DoesNotContainTest.cs	9	0	9	51/84	60.71	51/84	60.71	0
EmptyTests.cs	5	2	3	22/26	84.62	22/26	84.62	0
FalseTests.cs	2	2	0	6/6	100	N/A	N/A	N/A
InRangeTests.cs	7	0	3	31/72	43.06	31/72	43.06	0
TrueTests.cs	2	2	0	6/6	100	N/A	N/A	N/A
TheoryCommandTests.cs	9	0	7	44/48	91.67	46/48	93.75	2
Total of xUnit	81	12	60	166/172	96.51	167/172	97.09	1
Total of xUnit.extension	9	0	7	44/48	91.67	46/48	93.75	2
Total of both projects	90	12	67	210/220	95.45	213/220	96.81	3

introduce parameterized mock objects to deal with these difficulties.

4. Benefits of PUTs

We next present the benefits we found in the test generalization phrase and the writing new PUT phrase.

4.1 Benefits of Test Generalization

In our test generalization, we generalized 81 tests in the core xUnit project and 9 tests in the extension projects. All these tests are testing the main class `Assert` and the extension class `TheoryCommand`. Except 12 of them are not amenable for test generalization, all the remaining 78 tests are generalized into 67 PUTs. When explored by Pex, the generated tests based on our PUTs not only achieve better coverage, but also assist Pex to generate useful tests which found a defect that is not detected by the original conventional unit tests in `the` xUnit core project.

Table 2 shows the results of our test generalization. Column "Test Class" shows the name of the test classes and the column "Unit Tests" shows the details of the conventional unit tests (CUT in the table) and PUT. The "NA" sub-column shows the number of the conventional unit tests that are not amenable for generalization. The columns "CUT Block Coverage" and "PUT Block Coverage" show the obtained block coverage of the conventional unit tests and PUTs, which include the sub-columns "C/T" for "covered blocks over total blocks" and "%Cov" for the percentage of coverage. The "New Block" column shows the new covered blocks by our PUTs and the total results of both projects are showed at the bottom of the table.

```
[Fact]
public void DoubleNotWithinRange()
{
    Assert.Throws<InRangeException>(() =>
                        Assert.InRange(1.50, .75, 1.25));
}
[Fact]
public void DoubleValueWithinRange()
{
    Assert.InRange(1.0, .75, 1.25);
}
```

Figure 8. Conventional unit tests that can be generalized into a single PUT

Since PUT is more generalized than convention unit test, the number of the generalized PUTs is fewer than the original unit tests. By combining the similar conventional unit tests, we can transform them into a single PUT with less test code. In `EqualTests` and `InRangeTests` classes, the numbers of the generalized PUTs are 5 and 4 fewer than the original tests, which decrease by 12.8% and 57% respectively. Figure 8 shows the original tests that can be generalized into one PUT, which is showed in Figure 9.

In the example PUT of Figure 9, with the assumption that ensures the lower bound of the range is smaller than the upper bound, the tests generated by Pex include one case that the value is inside the range and other case that the value is outside the range. The `PexAllowedException` attribute informs Pex to capture the expected `InRangeException` thrown when the value is outside the range. However, this is not often the case. In TheoryCommandTests class, we created three PUTs for a single conventional unit test

Table 3 Benefits of Writing New PUTs

Test Class	#New PUT	C/T	%Cov	#New Block	% Orig	% Gen	% Increase
ContainsTests.cs	5	77/84	91.67	25	61.90	61.90	29.76
DoesNotContainTest.cs	5	77/84	91.67	26	60.71	60.71	30.95
EmptyTests.cs	1	26/26	100	4	84.62	84.62	15.38
InRangeTests.cs	5	65/72	90.28	34	43.06	43.06	47.22
Average of xUnit	4	61/67	92.11	22.25	62.57	62.57	30.83
Total of xUnit	16	177/177	100	10	96.51	97.09	2.91

because there are two more different situations that are not handled by the original test.

```
[PexMethod , PexAllowedException(typeof(InRangeException))]
public void TestInRangePUTDoubleValueInRange
(double i,double j, double value)
{
    PexAssume.IsTrue(i < j);
    Assert.InRange(value,i,j);
}
```

Figure 9. A single PUT generalized from tests in Figure 8

Although the number of PUTs is fewer than the original tests, the coverage achieved by PUTs is still higher. As we can see from the results, for each class, the generalized PUTs achieve higher or at least same coverage compared to the conventional unit tests. In the `EqualTests` class, although the coverage of the conventional unit tests are very high, 96.10%, our generalized PUTs still can achieve better coverage, 98.70%, by covering 2 more blocks. In the `TheoryCommandTests` class, with the help of the parameterized mock object showed in Figure 7, out PUTs again covered 2 new blocks by throwing the desired type of exception that is not captured by the original tests. These new covered blocks contribute to the increase of the total coverage from 95.45% to 96.81%.

In addition to achieve high coverage, the tests generated by Pex also have more chances to detect defects since the test values are more general. Invoking a method or accessing a property value of a null object is a common defect in object-oriented program, which can be prevented by providing proper preconditions of null check. However, if the code base is huge, it is very difficult to ensure that all of the necessary preconditions of null check are added properly. As Pex usually tries null value for object parameter for the initial case, the values generated can find out such defects very effectively. The xUnit project is tested thoroughly and used widely by lots of developers. But when we executed the generated tests, we still can find out such a defect that lack of null checking. In one of the PUTs of `Equaltests` class, Pex generated an empty string array and a string containing a `null` object for the equal test. Since these string arrays differ in the length, the test is expected to throw an `EqualException`. Instead of receiving the `EqualException`, the test reports a `NullReferenceException`. This is because the `null` object in the second string array is used to construct the error string for the `EqualException`, which shows that it lacks of a null check before the error string construction.

4.2 Benefits of Writing New PUTs

To achieve higher coverage, we wrote 16 new PUTs for 4 classes and obtained 100% coverage of `xUnit` project. Table 3 presents the new PUTs added and their corresponding coverage achieved. Column “Test Class” shows the name of the class that we chose to add new PUTs. Here we only added new PUTs for the classes whose coverage was not close to 100% in the phase of test generalization and the uncovered blocks are not caused by infeasible test requirements, such as dead code. Column “#New PUT” shows the number of PUTs added for each class and column “C/T” and “%Cov” shows the covered blocks over total blocks and the percentage of coverage. To make the comparison easily, column “%Orig” shows the original coverage achieved by conventional unit tests, column “% Gen” shows the coverage in the phase of test generalization, and column “%Increase” shows the increase of coverage achieved by the added PUTs.

Test generalization can benefit the coverage since the arguments passed to the tests are more general than the concrete test values in conventional unit tests. But generalizing arguments and tests have their limitations, which can be seen from the results showed in Table 2 and Table 3. Writing new PUTs provides a supplement

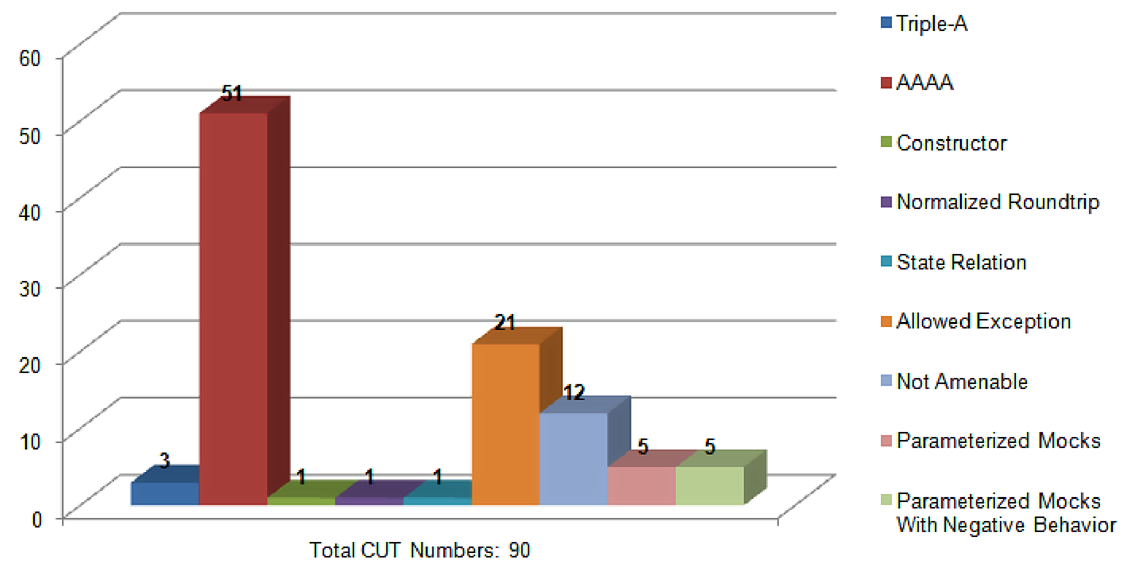

Figure 10. Results of test categorization

to this insufficiency. By carrying out the analysis of the coverage report and the original tests, we can identify the uncovered blocks and find out the causes that prevent Pex to generate test inputs to cover them. Based on this analysis, we can write new PUTs to target at the uncovered blocks and use the knowledge obtained from test generalization to provide proper assumptions and assertions. This procedure ensures that the PUTs generated by our two-phase approach not only achieve high coverage, but also provide proper assumptions and assertions to serve as more complete specifications.

As we can see from the results showed in Table 3, the average coverage increases achieved by the new PUTs for each class is 30.83% and the highest increase reaches 47.22% (`InRangeTests`). Comparing the coverages achieved by the original conventional unit tests and the phase of test generalization, we can see that these classes do not benefit from test generalization since the coverage achieved by test generalization do not gain any increase from the original coverage for each class. By analyzing the coverage report and original tests, we found that this was because these classes shared the same problem: lacking unit tests to test arguments that implement specific interfaces, such as `IEquatable` and `IComparable`. With the knowledge about `xUnit`, which is obtained from the phase of test generalization, it was much easier for us to write the new PUTs with proper assumptions and assertions to satisfy these test requirements than writing them from scratch. The result is promising: These accumulated coverage increases contribute to the full block coverage of `xUnit` project (covered all the 177 blocks of `Assert`), which indicates high testing confidence and thoroughness.

5. Test Categorization

In our study, we use the test patterns proposed by Halleux and Tillmann [10] to do the test generalization. In most of the cases, the patterns can be applied directly. They provide us with templates to write PUTs, which is an efficient way to start the generalization. However, in the tests we generalized so far, we find that only a few test patterns re frequently used, others are either too complicated to use or only used in some specific cases. In section 5.1, we present the results of test categorization and discuss the patterns according to their used frequency. Additionally, during our study, we noticed that not all conventional unit tests can be generalized into PUTs. Thus, in section 5.2, we talk about these exceptional cases in detail.

5.1 Amenable Cases

So far in our study, we have 90 conventional unit tests involved (12 of them are not amenable for test generalization), from which we generalized 67 PUTs. We categorize these conventional unit tests and the result is shown in Figure 10[1]. It suggests that only a few patterns are frequently used in the xUnit framework, such as 2.1, 2.2 and 2.10. Some of the

[1] Some conventional unit tests can be fitted in both mock pattern and PUT pattern. Thus, the number of patterns used would be more than 90.

patterns, such as 2.3, 2.5, and 2.6, are barely used. Similarly, not all the mock patterns are frequently used, which is covered by our discussion in Section 5.1.3.

5.1.1 Frequently Used Test Patterns

Pattern 2.1 has been used a lot in our study. It is the well-known *Triple-A* pattern (*Arrange*, *Act*, and *Assert*). First we set up the unit we want to test, such as a variable. This step is called *Arrange*. Then we put the variable into certain observable state, such as assigning it a value. This is *Act*. At last, we verify the state of the variable by making assertions. This is known as *Assert*.

Most of the conventional unit tests we studied are in this pattern. That is because most conventional unit tests are created to verify the behavior of certain unit. The most efficient way is to manipulate it first, and then observe it with assertions, which would be using the Triple-A pattern. However, pattern 2.1 is not the pattern we used most in generalized PUTs. According to the result shown in figure 8, we applied pattern 2.2 in more than 50% of the PUTs. Pattern 2.2 is known as *Assume, Arrange, Act, Assert* pattern. The only difference between pattern 2.1 and 2.2 is the step of *Assume*. By making assumptions, we make our PUTs more intelligent. Not only we can preclude unwanted valued from being tested, but also we can narrow down the field of desired inputs.

Another frequently used pattern is 2.10 *Allowed Exception*. In some existing unit testing tools, they also allow a test to throw an exception. But it is based on the assumption that an exception would be thrown during the test. If it does not, the test will end up with a failure, which is frustrating. However, the concept of allow exception is different in Pex. A PUT in pattern 2.10 is allowed to throw exceptions, and it would still work if no exception comes up.

5.1.2 Rarely Used Test Patterns

Before the study, we did not expect pattern 2.3, *Constructor Test* pattern, to be one of the rarely used patterns. In fact, it is very useful. But in the unit tests we generalized so far, only one test can fit into this pattern, far fewer than we expected. A possible reason is that the project we use is a unit test framework, which is designed to test its functions and properties are rarely involved. The same reason also applies for the cases of pattern 2.4 and 2.5.

For the rest test patterns, they all require very specific preconditions. For example, pattern 2.7 can only be applied when we want to compare two objects with same observable behavior. These preconditions limit the usage of these test patterns in a few specific cases.

5.1.3 Mocking Patterns

As we mentioned in section 2.3, we used parameterized mock objects for tests that interacts with environment or require arguments to implement certain interfaces. So far, we used two mocking patterns in 10 tests.

Parameterized Mocks is a very useful pattern. Generally, it state assertions for input, obtain results from Pex's choice provider, and state assumptions on the results. We frequently used it in certain unit tests in `EqualTests`, `ContainstTests`, and `DoesNotContainTests`, wherever a comparer that implements the `IComparable` interface is needed.

Parameterized Mocks With Negative Behavior is a pattern that a mock object may choose to throw a specific type of exception. An example is shown in Figure 7. In `TheoryCommandTests`, we used this pattern in 5 PUTs and achieved a higher coverage by covering 2 more blocks.

The other two mocking patterns, *Choice Provider* and *Parameterized Mocks Negative Behavior*, are not used in our test generalization.

5.2 Not Amenable Cases

In our study, we find that some exceptional cases that are not amenable to be generalized into PUTs. For example, the `FalseTest` in our candidate project includes a test as shown in Figure 11. The purpose of this test is to verify the behavior of `Assert.False` when the given value is consistent with its expected value.

```
public void AssertFalse()
{
   Assert.False(false);
}
```

Figure 11 Example of Not Amenable Case

Undoubtedly, we can convert this test into a PUT by applying pattern 2.2 on it. However, we can not benefit more from the generalized PUT. Because the purpose of this test is to test the value `false` and there is no need for Pex to generate any input value other than `false`. In this case, there is no difference between the PUT and the original conventional unit test. Thus, we classify this category of tests to be not amenable.

6. Helper Techniques for Test Generalization

In this section we are going to describe several helpful techniques that are used in our test generalization.

6.1 Factory Methods

The technique of factory method is used to help Pex in generating method-call sequence for creating instances of object in desired states, which is demonstrated in section 2.2. Provided a factory method of a specific type of object, if we have several PUTs that require the object in different states, we can state proper assumptions in the PUTs to constraint the object in the desired state. However, if the generated symbolic values from Pex are discarded in the factory method when constructing the object, then we may not be able to add the assumptions. Figure 12 shows the constructor of `TheoryCommand`, in which the symbolic value of `displayName` is discarded if it is null.

```
00:   public TheoryCommand(IMethodInfo testMethod, string
                    displayName, object[] parameters) : base(testMethod)
01:   {
02:      Parameters = parameters ?? new object[0];
03:      DisplayName = String.Format("{0}({1})",
                              displayName ?? testMethod.TypeName + "." +
                              testMethod.Name, string.Join(", ", displayValues));
04:   }
```

Figure 12 Object that discard symbolic values

Figure 6 shows the corresponding factory method for `TheoryCommand`. In our test generalization, one of the PUTs that uses this factory method is going to test the `DisplayName` property of `TheoryCommand` class by setting `displayName` null or not null.

As we can see from the constructor, the value of `DisplayName` is the concatenation of the type name and method name of the `testMethod` if `displayName` is null or it is simply the value of `displayName` if `displayName` is not null. To distinguish these two situations in our PUTs, we need to add different assumptions to constraint the value of displayName null in one PUT and not null in another PUT.

```
00:  [PexFactoryMethod(typeof(TheoryCommand))]
01:  public static TheoryCommand Create(IMethodInfo testMethod,
string displayName, object[] parameters)
02:  {
03:        PexRepository.Store("parameters", parameters);
04:        PexRepository.Store("displayName", displayName);
03:        var theoryCommand = new TheoryCommand(testMethod,
                                        displayName, parameters);
05:        return theoryCommand;
06:  }
```

Figure 13. Revised factory method with PexRepository to store symbolic values

Since displayName is discarded if it is null, when we state the assumption that “`PexAssume.IsTrue (command.Display == null)`”, Pex is not able to generate the proper state of the object. To address this problem, we create a helper object PexRepository to store the symbolic values in the factory method, which can be used later in the PUT to state proper assumptions on the symbolic values. To add the assumption of `displayName`, we can state “`PexRepository.Get<string>(“displayName”) == null`”, in which way Pex can figure it out and perform the input generation as desired.

The only problem here is that this technique exposes the value of `TheoryCommand` and violates its encapsulation. However, if Pex can integrate it into its context object or something similar and provide an interface for user to access, then we can solve this problem elegantly.

6.2 Parameterized Mock Objects

Using mock objects can assist Pex to deal with the environment dependency issues when generating tests. We have introduced an example in section 2.3, which adopts the *Parameterized Mocks* and *Parameterized Mocks With Negative Behavior* patterns. Although we do not use *Parameterized Mocks Properties* pattern in our test generalization, it is very useful because it can be used to prevent generating infeasible values or invalid states of the mock objects. However, it requires more understanding of the program under test, so that we can know which value to store for the subsequent access of the same properties. In our future work, we plan to re-examine the mock objects and try to apply this pattern wherever there is a need.

6.3 Refactoring PUTs

Generalizing PUTs from the existing conventional unit tests is not a one-step procedure. In most cases, the generalization starts with analyzing a conventional unit test and ends with a single transformed PUT. However, as our study progressed, we noticed that the tests in the same class usually have something in common, such as structures and assertions. So if we revisit the PUTs after all the PUTs of a class were created, we might be able to refactor several PUTs into one PUT or modify promoted parameters.

When we analyze the tests shown in Figure 14, we can apply pattern 2.2 to generalize two PUTs to test

`Assert.Equal` and `Assert.NotEqual` where there are no exceptions and pattern 2.10 for two PUTs that allow the exceptions.

```
[Fact]
public void Array()
{
   string[] expected = { "@", "a", "ab", "b" };
   string[] actual = { "@", "a", "ab", "b" };
   Assert.Equal(expected, actual);
   Assert.Throws<NotEqualException>(() =>
     Assert.NotEqual(expected, actual));
}

[Fact]
public void ArraysOfDifferentLengthsAreNotEqual()
{
    string[] expected = { "@", "a", "ab", "b", "c" };
    string[] actual = { "@", "a", "ab", "b" };
    Assert.Throws<EqualException>(() =>
      Assert.Equal(expected, actual));
    Assert.NotEqual(expected, actual);
}

[Fact]
public void ArrayValuesAreDifferentNotEqual()
{
    string[] expected = { "@", "d", "v", "d" };
    string[] actual = { "@", "a", "ab", "b" };
    Assert.Throws<EqualException>(() =>
      Assert.Equal(expected, actual));
    Assert.NotEqual(expected, actual);
}
```

Figure 14 Conventional array tests

However, when we revisit those PUTs, we find that they can be further improved. In these three tests, they are actually testing two methods, `Assert.Equal` and `Assert.NotEqual`. Thus, we can simply create two PUTs, each of which tests only one method. To capture the case in which they throw exceptions, we can add PexAllowedException attributes on top of the PUTs.

```
[PexMethod, PexAllowedException(typeof (EqualException))]
public void TestEqualPUTArrayTests(
[PexAssumeUnderTest] string[] i,
[PexAssumeUnderTest] string[] j)
{
    PexAssume.IsTrue(i.Length > 0);
    PexAssume.IsTrue(j.Length > 0);
    Assert.Equal(i, j);
}
```

Figure 15 Refactored PUTs

One of the refactored PUTs is shown in Figure 15. The other one can be obtained by replacing `Assert.Equal` with `Assert.NotEqual`. By doing this, not only the PUTs become neater, but they can also state the specification of the functionality more clearly.

7. Limitation or Pex or PUTs

In this section we discuss about the limitations we found during our study.

7.1 Limited string and floating number generation

The first one is that Pex would fail to run when we use `CompareTo()`[2] to make assumptions.

```
public void StringNotWithinRange()
{
    Assert.Throws<InRangeException>(() =>
   Assert.InRange("adam", "bob", "scott"));
}
public void StringValueWithinRange()
{
   Assert.InRange("bob", "adam", "scott");
}
```

Figure 16 Conventional Unit Tests of InRange Test

The tests shown in Figure 16 are to verify the behavior of `Assert.InRange` when given with string inputs. From these two tests, we generalized the PUT shown in Figure 17.

```
[PexMethod, PexAllowedException(typeof(InRangeException))]
public void TestInRangePUTStringValueWithinRange
([PexAssumeNotNull]string value, [PexAssumeNotNull]string i,
[PexAssumeNotNull]string j)
{
   PexAssume.IsTrue(i.CompareTo(j) == -1);
   Assert.InRange(value, i, j);
}
```

Figure 17 Example PUT using CompareTo() to make assumption

In order to set up a range, we assume that the instance of `i` is less than `j`. Ideally, it should generate different string values as inputs to verify the behavior of `Assert.InRange`. However, when we apply Pex on this PUT, Pex stopped with 0 run. We tried several ways to modify the generalized PUT, but the problem still remained.

Table 4 Tests generated by Pex for the PUT shown in Figure 9

Tests	i	j	value
1	0	1	0
2	0	1	-1

The second limitation of input generation is that Pex can not generate float point values properly. A good example is the test result of the PUT shown in Figure

[2] `CompareTo()` is a member of `System.String`. It returns a signed integer to indicate the result of a lexical comparison between two strings.

9. The purpose of the original test is to verify the behavior of `Assert.InRange` with float point input values.

The PUT appears to be simple and flawless. But the result of the exploration is disappointing. As shown in Table 4, Pex can only generate integer values, which makes the result less reliable.

7.2 Insufficient information for issues

When Pex is exploring a PUT for test generation, it reports different kinds of issues that make it fail to generate a new test for covering a new branch. For example, it reports object creation issue for failing to create a non-primitive object and reports uninstrumented method issue for external method calls. However, these issues are reported directly with fix suggestion but without telling why and where they cause the problem. Moreover, the number of issues may be non-trivial to go through. When applying Pex in TheoryCommandTests, we got 31 uninstrumented method issues and most of them are related to some complex native system libraries, such as `String` and `Number`. Without more information about how the issues are related to the problem, we do not know which method Pex should instrument and analyze. We hope Pex can deal better with these issues in the next version.

8. Conclusion

To do the empirical study of PUTs, we propose a two-phase approach: generalize PUTs from the existing conventional unit tests and write new PUTs to cover the blocks that are not covered in the phase of test generalization. We investigate the performance by comparing the results between original tests and PUTs generated by our approach. To do the empirical study, we use the tests in the xUnit and xUnit.extension projects, which belong to the xUnit framework. In the phase of test generalization, the generalized PUTs increase the coverage from 95.45% to 96.81% and detect a new defect in the xUnit project. In the phase of writing new PUTs, we wrote 16 new PUTs to target at the uncovered blocks, which achieve 30.83% coverage increase in average and 100% coverage of xUnit project. Our study also provides the analysis of the test categorization and discusses several useful techniques for test generalization and writing new PUTs. We also discuss some limitation of Pex and PUTs. In future work, we plan to generalize more tests from different projects in xUnit framework and use different kinds of coverage criterion to assess the performance.

9. References

[1] Bookshelf - adaptive software development: A collaborative approach to managing complex systems extreme programming explained: Embrace change, software process quality: Management and control. *IEEE Software*, 17(4), 2000.

[2] xUnit, 2007. `http://www.nmock.org/`.

[3] NMock, 2008. `http://www.ayende.com/projects/rhino-mocks.aspx`.

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

[5] Rhino Mock, 2009. `http://www.ayende.com/projects/rhino-mocks.aspx`.

[6] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 2008.

[7] Charles Ashbacher. "test-driven development: By example" by kent beck (review). *Journal of Object Technology*, 2(2):203–204, 2003.

[8] Boris Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[9] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: automatically generating inputs of death. In *ACM Conference on Computer and Communications Security*, pages 322–335, 2006.

[10] P. de Halleux and N. Tillmann. Parameterized test patterns for effective testing with Pex. Technical report, Microsoft Research Technical Report, 2008.

[11] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *PLDI*, pages 213–223, 2005.

[12] Tim Mackinnon, Steve Freeman, and Philip Craig. Endo-testing: unit testing with mock objects. pages 287–301, 2001.

[13] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *ESEC/SIGSOFT FSE*, pages 263–272, 2005.

[14] Nikolai Tillmann and Jonathan de Halleux. Pex-white box test generation for .net. In *TAP*, pages 134–153, 2008.

[15] Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests. In *ESEC/SIGSOFT FSE*, pages 253–262, 2005.

[16] JUnit, 2003. `http://www.junit.org`.

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