# 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. In the field of unit testing, which is considered a fundamental part of software testing, there already exist several tools that could automatically generation conventional unit tests. Nevertheless, some of these tools could not ensure high code coverage, unless testers write some of test cases manually, which would be a burdensome and tedious work.*

*Pex, an automated white box testing tool developed by Microsoft, introduced a new method called parameterized unit tests (PUT), hoping to solve the issues above. With PUTs, Pex could generate conventional unit tests with different inputs automatically. This feature not only could save testers from those tedious works, but also could help us achieve higher code coverage. To do the empirical study of PUTs, we propose an approach to generalize PUTs from existing conventional unit tests and compare the performance of the conventional unites and the generalized PUTs. We used xUnit, which is an open source C# project, to study the performance of our approach in a real project. Our study result shows that compare to the original test cases, the block coverage has increased by ??% and ?? new defects were found.*


## 1. Introduction

Testing has been played an important and irreplaceable role in the software development process, which should be performed through the whole process. As part of the testing process, unit testing is a software verification and validation method in which the smallest testable parts of an application, called units, are individually and independently tested for appropriate operation. Well-written unit tests provide a working specification of the functional code and the technical documentation, reflect customer requirements and protect the functional code against change. Unit testing is adopted widely in industry due to the popularity of methods like XP ("extreme programming")[?], test-driven development (TDD)[?] and test execution frameworks like JUnit, NUnit or VSTest. A test suite produced by unit testing with high code coverage gives confidence in the correctness of the tested code. However, writing unit tests 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 [6] or Agitar JUnit factory [4] could automatically generate conventional unit tests. Nevertheless, these tools could not guarantee high code coverage, unless testers manually write some test cases.

Parameterized unit tests (PUT)[] 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 were represented with symbolic values in PUTs. Thus, PUTs are more general specifications than conventional unit tests: PUTs state the intended program behavior for entire classes of program inputs, and not just for one particular input. This feature could be taken advantage of by a tool, such as Pex[], to automatically generate test cases with inputs for the PUTs 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 []. 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, Z3 for Pex, 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, Pex generates a corresponding conventional unit test case.

Although PUTs are more generalized than conventional unit tests and can often achieve higher structural coverage[], like basic block coverage, branch coverage and implicit branch coverage, writing PUTs requires more efforts than conventional unit tests. A well written PUT requires the generalization of parameters and proper assumption and assertions. It is not a trivial work to write PUTs from scratch for a large code base. To do the empirical study of PUTs, we propose an approach to generalize PUTs from existing conventional unit tests and compare the performance of the existing conventional unit tests and the PUTs generalized by us. The 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 generated inputs and verify the results. For several conventional unit tests that test same method with different inputs, we would try to merge them into one PUT. Besides with these normal steps, there are two more issues we shall address. Firstly, 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 could provide a factory method for each such object by constructing the desired object states using simpler or primitive parameters. Secondly, if there is some argument implementing a specific interface, we could provide a parameterized mock object[<mock>] which implements the same interface for the generalized PUT. This could guide Pex to get around the environment dependency problem, like file system, database or reflection, and generate proper test inputs to achieve high coverage.

The rest of the paper is structured as follows: Section 2 illustrates with an example of the different methods for writing a new PUT. Section 3 describes the characteristics of xUnit framework. Section 4 presents the benefits of test generalization. In Section 5, we categorize the PUTs under different patterns and propose newly discovered patterns. Section 6 presents the helpful techniques for writing PUTs. In Section 7, we discuss the limitations of Pex and PUTs. We conclude with Section 8.

## 2. Example

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

```
public class ArrayTests
{
   [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 1. Example conventional unit test cases from xUnit project.

It is not difficult to tell that they are trying to verify the behavior of `Assert.Equal` and `Assert.NotEqual`, by using arrays with different characteristics. Obviously, these test cases are of the same pattern. The only difference is the values of the strings. Given this, we could set up a PUT by replacing the concrete string values with symbolic values and merge them into one PUT. A complete example of PUT is shown in Figure 2.

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

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

For this PUT, Pex could automatically generate conventional test cases with different values in `string i` and `string j`, including both the situations (may be more) described in the conventional test cases.

Although Pex is powerful enough to let us write a PUT with 9 lines of code instead of three test cases with 33 lines of code, and achieve the same goal. We still could not simply generate conventional tests into PUTs without analysis and processing. An important step of generating PUT is to define assumptions. Take the PUT shown in Figure 2 for example, Pex would generate null elements for `string i` and `j` by default. Then we would meet some troubles when executing the tests. In order to prevent this, we could use `PexAssume` to preclude null elements, as shown in Figure 2.

## 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 could 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.

| Attribute | Value |
|---|---|
| Total LOC | 24809 |
| xUnit LOC | 4789 |
| xunit.extensions LOC | 1295 |
| #Total Tests | 549 |
| #xUnit Tests | 310 |
| #xunit.extensions Tests | 50 |

Table 1 Code Metrics of xUnit Framework

The core module, xUnit package, has the largest amount of unit tests among all the packages, which could 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 could show how we introduce parameterized mock objects to deal with these difficulties.

## 4. Benefits

(Xiao)

Author names and affiliations are to be centered beneath the title and printed in Times 12-point, non-boldface type. Multiple authors may be shown in a two- or three-column format, with their affiliations italicized and centered below their respective names. Include e-mail addresses if possible. Author information should be followed by two 12-point blank lines.

## 5. Categorization

(Ludi)

(Not amenable)

## 6. Helper (xiao)

Wherever Times is specified, Times Roman or Times New Roman may be used. If neither is available on your word processor, please use the font closest in appearance to Times. Avoid using bit-mapped fonts if possible. True-Type 1 fonts are preferred.

## 7. Limitation(Xiao)

Type your main text in 10-point Times, single-spaced. Do **not** use double-spacing. All paragraphs should be indented 1/4 inch (approximately 0.5 cm). Be sure your text is fully justified—that is, flush left and flush right. Please do not place any additional blank lines between paragraphs.

**Figure and table captions** should be 10-point boldface Helvetica (or a similar sans-serif font). Callouts should be 9-point non-boldface Helvetica. Initially capitalize only the first word of each figure caption and table title. Figures and tables must be numbered separately. For example: "Figure 1. Database contexts", "Table 1. Input data". Figure captions are to be centered *below* the figures. Table titles are to be centered *above* the tables.

# 8. First-order headings

For example, "1. Introduction", should be Times 12-point boldface, initially capitalized, flush left, with one blank line before, and one blank line after. Use a period (".") after the heading number, not a colon.

## 8.1. Second-order headings

As in this heading, they should be Times 11-point boldface, initially capitalized, flush left, with one blank line before, and one after.

**8.1.1. Third-order headings.** Third-order headings, as in this paragraph, are discouraged. However, if you must use them, use 10-point Times, boldface, initially capitalized, flush left, preceded by one blank line, followed by a period and your text on the same line.

# 9. Footnotes

Use footnotes sparingly (or not at all) and place them at the bottom of the column on the page on which they are referenced. Use Times 8-point type, single-spaced. To help your readers, avoid using footnotes altogether and include necessary peripheral observations in the text (within parentheses, if you prefer, as in this sentence).

# 10. References

List and number all bibliographical references in 9-point Times, single-spaced, at the end of your paper. When referenced in the text, enclose the citation number in square brackets, for example [1]. Where appropriate, include the name(s) of editors of referenced books.

[1] A.B. Smith, C.D. Jones, and E.F. Roberts, "Article Title", *Journal*, Publisher, Location, Date, pp. 1-10.

[2] Jones, C.D., A.B. Smith, and E.F. Roberts, *Book Title*, Publisher, Location, Date.

# 11. Copyright forms and reprint orders

You must include your fully-completed, signed IEEE copyright release form when you submit your paper. We **must** have this form before your paper can be published in the proceedings. The copyright form is available as a Word file, <copyright.doc>, as a PDF version, <copyright.pdf>, and as a text file in <authguid.txt>.

Reprints may be ordered using the form provided as <reprint.doc> or <reprint.pdf>.