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 input values 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. Furthermore, unit tests provide a specification of the functional code, server as a technical documentation and reflect customer requirements. In industry, unit testing is adopted widely due to the popularity of software development methods, such as extreme programming (XP) [?] and test-driven development (TDD) [?] and test execution frameworks, e.g. 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 [?] or Agitar JUnit factory [?] 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

2.1 Simple Example

In this section, we are going to explain how to convert existing conventional unit tests into PUTs. The xUnit test case 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 cases from xUnit project.

It is not difficult to tell that this test case 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 case 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 []. Given this, we could 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 cases 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 could add an assumption, saying that the object under test is not null. Specifically, in this test case, we also don’t want list contain any null value. So we could add an assumption at the beginning of the test case. By making assumption, we change the test pattern from 2.1 (Triple-A) to 2.2 (Assume, Arrange, Act, Assert) [Ref : Parameterized Test Patterns For Effective Testing with Pex]. 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 example unit test cases shown in Figure 1.

By exploring this PUT, Pex could automatically generate conventional test cases with different values in list, including as many different situations as possible, which could 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 could not generate any test case due to the difficulties of creating a instance of `AssertException`.

```
[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 parameter.

To address this issue, Pex let user define a factory method which could be utilized to create instances of the object and modify them into desired states. Normally, a factory method receives simpler or primitive parameters and produces a desired instance of the object based on these parameters. Our factory method for the PUT in Figure 4 is showed in Figure 5.

```
[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

This factory method simply accepts a string parameter and constructs a 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 could not 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 could just ignore other complex properties.

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 could 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.

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

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

In this example, the method parameter that implements interface `IMethodInfo` could be mocked and assist Pex to generate proper test cases. Many mock frameworks, such as NMock[] and Rhino Mocks[], 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 could turn the return values of the methods of the mock object into symbolic values, which could 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.

As we could 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 could 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 could generate different test cases in which a specific type of exception is thrown for covering the catch block if there is some.

```
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

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 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 Characteristics 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 of PUTs

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 67 tests are generalized into 84 PUTs. When explored by Pex, the generated test cases based on our PUTs not only achieve better coverage, but also assist Pex to generate useful test cases which found a bug 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

Table 2 Results of Test Generalization

Test Class	Unit Tests			CUT Block Coverage		PUT Block Coverage		New Block
	#CUT	#NA	#PUT	C/T	Ratio	C/T	Ratio	
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

blocks over total blocks“ and “Ratio” for the ratio 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.

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 could 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 could be generalized into one PUT, which is showed in Figure 9.

```
[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 could be generalized into a single PUT

```
[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

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

Although the number of PUTs is fewer than the original tests, the coverage achieved by PUTs is still higher. As we could 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 could 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 test cases generated by Pex also have more chances to detect bugs since the test values are more general. Invoking a method or accessing a property value of a null object is a common bug in object-oriented program, which could 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 could find out such bugs very effectively. The xunit project is tested thoroughly and used widely by lots of developers. But when we executed the generated tests, we still could find out such a bug that lack of null checking. In one of the PUTs of `Equaltests` class, Pex generated a 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.

5. Test Categorization

In our study, we use the test patterns proposed in *Parameterized Test Patterns For Effective Testing with Pex* [pattern] to convert the conventional unit tests into PUTs. Generally, the patterns are very helpful. They provide us with templates to write PUTs, which would be an efficient way to achieve high code coverage. However, in the test cases we generalized so far, we found that only a few test patterns are frequently used, others are either too complicated to use or only used in some specific cases. In section 5.1, we will present the result of categorization of generalized PUTs. We will discuss patterns according to their used frequency. At the same time, during our study, we noticed that not all conventional unit tests could be generalized into PUTs. Thus, in section 5.2, we will discuss these exceptional cases in detail.

5.1 Amenable Cases

So far in our study, we have 90 conventional unit tests involved, from which we generalized 87 PUTs. We categorize these PUTs and the result is shown in figure 8. It suggests that only few patterns are frequently used, such as 2.1, 2.2 and 2.10. Some of the patterns, such as 2.3 2.5 and 2.6, are barely used. Thus, we will discuss these patterns in the following sections according to their used frequency.

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 test cases 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.

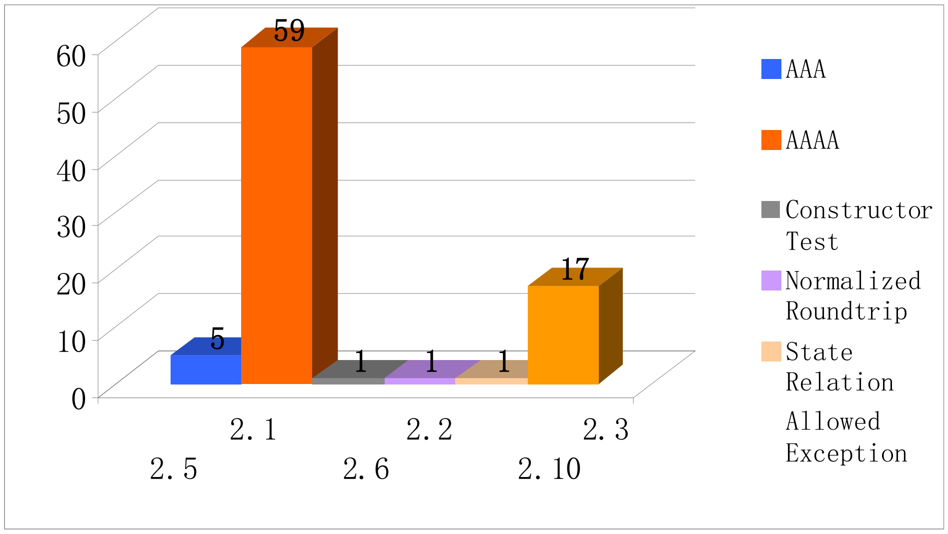

Figure 10. A single PUT generalized from tests in Figure 8

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 case 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 Barely Used Test Patterns

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

To the rest test patterns, they all require very specific preconditions. For example, pattern 2.7 could only be applied when we want to compare two objects with same observable behavior. These preconditions decide that these test patterns could only be applied on specific cases.

5.2 Not Amenable Cases

In our study, we found some exceptional cases that could not be generalized into PUTs. Among these cases, some are so simple that it is not necessary to generalize them into PUTs. For example, suppose we have a conventional unit test. First it assigns true to the variable, and then verifies if its value is true. In this particular case, the target under test has only one desired value. Thus, it is meaningless to generalize it into PUT. The `FalseTest` and `TrueTest` in our candidate project are typical examples of this case.

6. Helper (xiao)

Wherever Times is especified, 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>.