An Empirical Study of Test Generalization in taglib-sharp

Xiaokang Xiang
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
xxiang4@illinois.edu

Brandon Carlson
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
blcrlsn2@illinois.edu

Abstract

There are many approaches to automating the generation of conventional unit tests across multiple programming languages. Although these approaches tend to fail at ensuring the necessary code coverage to find the majority of the issues in the source code. Furthermore, the manual writing of test cases is labor-intensive and the individual will typically only cover the minimal number of test cases. One of the ways to help resolve both of these issues with conventional unit tests is to use parameterized unit tests (PUT) where a developer can describe the expected behavior or specifications using symbolic values. For C# , a tool called Pex which is an automatic test generation tool will accept these PUTs and generate a set of conventional unit tests. Pex will attempt to achieve high code coverage by exploring as many branches in the code as possible. This process of converting conventional unit test into PUTs is referred to as test generalization. In our empirical study, we used an open source C# project called taglib-sharp to study the benefits of test generalization. In our study, we found that the test generalization has increased the block coverage by X.XX% (on average) and also detected X new defects.

1. Introduction

In the software development lifecycle, the creation of unit tests is a critical step in the process. Not only does it help to detect defects at an early stage and ensure quality, but it also allows for future developers to easily make changes to the source code. The wide adoption of unit testing has led to the creation of a number of automatic unit test generation tools. A few of those tools include Evosuite [2], Parasoft JTest [6], and Agitar JUnit factory [1] all of which generate unit tests for the selected source code in their own unique way. All of these tools only generate conventional unit tests and currently do not accept any parameters. While each of these tools is getting better at generating unit tests they are still unable to guarantee full code coverage. Although when compared to the time-consuming task of manually writing these unit tests these tools do tend to improve the quality of code.

An improvement on conventional unit tests was the creation of Parameterized Unit Tests (PUTs), where unlike the conventional tests these accept parameters that provide the inputs to each individual unit test. These PUTs allow you to provide symbolic values to represent the expected behavior or specifications of the method that is under test. A tool that does this type of unit testing is called Pex [4] developed by Microsoft as a test generation tool. This tool tries to achieve high code coverage by taking these PUTs and generate a minimal set of conventional unit tests for the method under test. The process that Pex uses is called symbolic execution [7] to generate the conventional unit tests from PUTs. When Pex first explores the method under test it uses random values to collect the constraints along the executed path. From that information, it uses a constraint solver to explore alternate paths in the method by systematically flipping the captured constraints and generate concrete values. If Pex discovers a new path it will generate a new conventional unit test case.

While using PUTs can achieve better code coverage and is typically more effective it is nontrivial to write PUTs. When compared to writing conventional unit tests it requires more time and effort, especially to write efficient PUTs. To address this issue, we use the following procedure, where we initially look at and pick a subset of the existing conventional unit tests to convert to PUTs. We then identify the arguments and receiver objects of the unit test and where possible promote them to be method arguments. Next, we convert the conventional unit test assertions into Pex assertions, PexAssert. During this process, we did experience a number of challenges that prevented some of the conventional unit tests from being converted to PUTs. One example of this would be when then the method under test required a special file type in order to perform the unit test. In this instance, Pex is unable to generate the required input

for the method under test. To partially help address this issue we used the feature that Pex provides to create factory methods that let the user provide a method for generating the desired object state for the PUT argument. While this did not solve all of the issues it did help in some instances. Another issue that is experienced is that the method under test has a set of assumptions about the inputs to the unit test. Based on the observations of both the current conventional unit tests and the source code we added a series of assumptions to the PUTs. In order to do this, we used Pex's built-in method PexAssume to add the required assumptions. For example, if the input is required to be not null or empty we would use the PexAssume.IsNotNullOrEmpty to ensure the input provided by Pex met the expected requirement. //TODO: Add sentence about if you propose test patterns XXX

In our study, we used the taglib-sharp [5] from the Mono Project. The Mono Project is sponsored by Microsoft and is an open source implementation of Microsoft's .NET Framework based on the ECMA standards for C# and the Common Language Runtime. The taglib-sharp project is their library for reading and writing metadata in media files, including video, audio, and photo formats. During our research, we found that the test generations XXXXXX the code coverage on average by X.X% with a maximum of XX.XX% for a test class. Additionally, we found that the test generational help cover XX new blocks that were not covered by the original conventional unit test and discovered X new defects.

The rest of the paper is structured as follows Section 2 presents an example from taglib-sharp and explains the procedure for the test generalization. Section 3 describes the characteristics of taglib-sharp. Section 4 presents the benefits of test generalization found within our study. Section 5 presents the categories of conventional unit tests into PUT test patterns. Section 6 describes the categories of tests that are not amenable for test generalization. Section 7 describes how the helper techniques provided by Pex are used in our test generalization. Section 8 discusses the limitations of Pex. Then finally in Section 9 the conclusion.

2. Example

In this section, we will explain our procedure for the test generalization of conventional unit tests using Pex. We use the taglib-sharp test case TestTitle shown in Figure 1 as an illustrative example for explaining our procedure. The objective of this unit test is to verify the behavior of the title field in the DivXTag class. To generalize this test case, we initially identify the concrete values used in the test case. In this example, it includes the concrete value "012345678901234567890123z567890123456789012...." We can replace this concrete and similar values with a

Figure 1 Conventional Unit Test from the Riff folder in taglib-sharp.

```
01: public void TestTitle ()
02: {
03:   Riff.DivXTag tag = new Riff.DivXTag ();
04:   Assert.IsTrue (tag.IsEmpty, "Initially empty");
05:   Assert.IsNull (tag.Title, "Initially null");
06:   ByteVector rendered = tag.Render ();
07:   tag = new Riff.DivXTag (rendered);
08:   Assert.IsTrue (tag.IsEmpty, "Still empty");
09:   Assert.IsNull (tag.Title, "Still null");
10:   tag.Title = "012345678901234567890123"
11:            +"4z567890123456789012345678 9";
12:   Assert.IsFalse (tag.IsEmpty, "Not empty");
13:   Assert.AreEqual ("012345678901234567"
14:      +"8901234567890123456789012345678 9", tag.Title);
15:   rendered = tag.Render ();
16:   tag = new Riff.DivXTag (rendered);
17:   Assert.IsFalse (tag.IsEmpty, "Still not empty");
18:   Assert.AreEqual ("0123456789012345678"
19:      +"9012345678901", tag.Title);
20:   tag.Title = string.Empty;
21:   Assert.IsTrue (tag.IsEmpty, "Again empty");
22:   Assert.IsNull (tag.Title, "Again null");
23:   rendered = tag.Render ();
24:   tag = new Riff.DivXTag (rendered);
25:   Assert.IsTrue (tag.IsEmpty, "Still empty");
26:   Assert.IsNull (tag.Title, "Still null");
27: }
```

Figure 2 PUT Skeleton for the Conventional Unit Test.

```
01: [PexMethod(MaxRunsWithoutNewTests = 200)]
02: public void FullTagClearTest(
03:        Riff.DivXTag tag,
04:        string title, string[] performers,
05:        uint year, string comment,
07:        string[] genres )
08: {
09:    tag.Title = title;
10:    tag.Performers = performers;
11:    tag.Comment = comment;
12:    tag.Genres = genres;
13:    tag.Year = year;
14:    PexAssert.AreEqual(tag.Title, title);
15:    PexAssert.AreEqual(tag.Performers.Length,
16:          performers.Length);
17:    PexAssert.AreEqual(tag.Comment, comment);
18:    PexAssert.AreEqual(tag.Genres.Length,
19:         genres.Length);
20:    PexAssert.AreEqual(tag.Year, year);
21:    //Clear
22:    PexAssert.AreEqual(false, tag.IsEmpty);
23:    tag.Clear();
24:    PexAssert.IsTrue(tag.IsEmpty);
25:    }
```

symbolic value making them arguments for a PUTs. An advantage of replacing these concrete values with symbolic values is that Pex can generate concrete values based on the constraints encountered by taking different paths of the method under test.

Initially, we used the conventional unit tests to identify which PUT pattern the test belongs based on Halleux and Tillmann [3]. Using the patterns in the paper can in the process of generalizing conventional unit test into PUTs. In our example in Figure 1, it matches the pattern for Roundtrip, where the test first sets the Title field and then asserts that it matches the value from the field's getter method. Should one of the conventional unit test not fall into any of the predefined patterns we will define a new pattern for the conventional unit test.

In Figure 2 it shows the skeleton of the PUT after generalizing the concrete values and combining multiple conventional unit tests into a single PUT. The PUT accepts six different parameters: Riff.DivXTag tag, title, list of performers, year, comment, and a list of genres. In the original conventional unit test TestTitle, it switched back and forth testing whether the title field was empty or matched the provided value. This was consistent with the rest of the related unit tests like TestComment which executed a very similar repeating pattern. In our PUT skeleton, Figure 2, we simplified the series of unit tests into one PUT that can test each of the fields at once thereby reducing the number of unit tests to maintain. Additionally, our PUT is able to test multiple values for title compared to the static values provided in TitleTest, Figure 1, as shown in lines 10-18. These are two of the advantages of using PUTs over conventional unit test as these two simplifications increased the code coverage and reduced the number of conventional unit tests without changing the behavior being tested. As illustrated in Figure 2 we also converted the assertions to PexAssert to test the behavior. The only additional assertions added to Figure 2 was combining the TestClear conventional unit test into our PUT as a separate unit test is not necessary and is more efficient to test this functionality while testing all of the fields.

While Pex can handle parameters that are of primitive types like string and integer, it, however, has problems when it comes to non-primitive types like the Riff.DivXTag in our example, Figure 2. Pex will attempt to guess at how to build the object, but you may have to assist in the process by creating a factory method to tell Pex how to create the object. Additionally, to assist Pex, users can provide assumptions in the factory method to aid in the creation of the object, just like in the PUTs themselves. For example, the factory for the Riff.DivXTag object, as shown in Figure 3, is fairly simple where it constructs a Riff.DivXTag object with no parameters. This particular factory may need to be further explored in the future as the class provides multiple ways to construct the Riff.DivXTag object.

Figure 3 Factory Method to assist Pex

```
01: public static partial class DivXTagFactory
02: {
03:   [PexFactoryMethod(typeof(DivXTag))]
04:   public static DivXTag Create()
05:    {
06:      DivXTag divXTag = new DivXTag();
07:      return divXTag;
08:     }
09: }
```

The last phase of test generalization process is to define any assumptions that need to be made for Pex to use. Without these assumptions, Pex by default will generate null and empty values for the PUT arguments. To address this issue for the Riff.DivXTag we annotate it with the tag PexAssumeUnderTest, which tells Pex that this value should not be null and be of the same type as the argument. For those arguments that we do not want Pex to provide null or empty, we add PexAssume.IsNotNullOrEmpty, which tells Pex we do not want either of these cases. Instead of null or empty Pex will provide one or more "
0" to those arguments. Examples of both the PexAssumeUnderTest and PexAssume are shown in Figure 4 on lines 3 and lines 9-12. Figure 4 also shows our completed PUT for the conventional unit test.

Figure 4 Complete PUT for the Conventional Unit Test.

```
01: [PexMethod(MaxRunsWithoutNewTests = 200)]
02: public void FullTagClearTest(
03:    [PexAssumeUnderTest]Riff.DivXTag tag,
04:        string title, string[] performers,
05:        uint year, string comment,
07:        string[] genres )
08: {
09:    PexAssume.IsNotNullOrEmpty(title);
10:    PexAssume.IsNotNull(genres);
11:    PexAssume.IsNotNull(performers);
12:    PexAssume.IsNotNullOrEmpty(comment);
13:    tag.Title = title;
14:    tag.Performers = performers;
15:    tag.Comment = comment;
16:    tag.Genres = genres;
17:    tag.Year = year;
18:    PexAssert.AreEqual(tag.Title, title);
19:    PexAssert.AreEqual(tag.Performers.Length,
20:          performers.Length);
21:    PexAssert.AreEqual(tag.JoinedPerformers,
22:          string.Join("; ", performers));
23:    PexAssert.AreEqual(tag.Comment, comment);
24:    PexAssert.AreEqual(tag.Genres.Length,
25:         genres.Length);
26:    PexAssert.AreEqual(tag.JoinedGenres,
27:         string.Join("; ", genres));
28:    PexAssert.AreEqual(tag.Year, year);
29:    PexAssert.AreEqual(false, tag.IsEmpty);
30:    tag.Clear();
31:    PexAssert.IsTrue(tag.IsEmpty);
32: }
```

3. Open Source Project Under Test

taglib-sharp is part of the Mono Project which is an open source implementation of Microsoft's .NET Framework based on the ECMA standards for C# and the Common Language Runtime [5]. The taglib-sharp library of the Mono Project is focused on the reading and writing metadata in media files, including video, audio, and photo formats. It is written in C# and is actively being developed as it has over 71 commits in the last year. It is also being actively followed with 83 watchers, 425 stars, and 170 forks on GitHub as of November 2017. For its current test library, it is using conventional unit test and making use of test fixtures along with a number of sample files used for testing.

The reason for choosing taglib-sharp for our test generalization is the amount of documentation and examples available in the GitHub repository. Additionally, it contains a large number of unit tests that cover 72.26% of the current code base, which allows for a better understanding of the library and provides a number of potential unit tests that could be converted to PUTs. The current source code of the taglib-sharp library includes 242 classes and about 15K LOC. The test code includes 175 classes with about 19K LOC. For the purpose of this study, we primarily choose to focus on the tagging classes within the different files types handled by taglib-sharp. Since these are a better fit for test generalization and use by Pex than trying to work with the sample files themselves which Pex currently does not handle well.

Attribute	Value
Number of Classes	242
Number of Methods	3,498
Number of Public Methods	2,928
Lines of Code	15,578
Number of Test Classes	175
Number of Test Methods	980
Lines of Test Code	19,007

Table 1. Code Metrics for taglib-sharp

4. Benefits of Test Generalization

For the taglib-sharp project of the Mono framework, there are a total of 175 test classes that contain a total of 980 conventional test methods. Of these 175 test classes, we generalized XX test classes that contain XX conventional unit tests into XX PUTs. While we tried to generalize as many of the test methods as possible in these classes a certain percentage of the test methods were not amenable for test generalization. The results of our test generalization are shown in Table 2. The column labeled "Test Class" shows the name of the test class we choose to convert to PUTs. The next column labeled "Test Methods" show the statistics of the existing conventional unit test methods, the test methods that were not amenable to test generalization, the percentages of those test that were amenable, and finally the number of PUTs that were created. The details of the test methods that were not amenable to test generalization can be found in section 6 of this paper. The next column labeled "Coverage" shows the block coverage reported from executing test cases of each type of unit test conventional and PUT. The final two columns show the number of new blocks covered by PUTs and the number of new defects found from using PUTs. In the majority of cases, we found that PUTs were able to achieve high code coverage while also covering new blocks that were not covered by the conventional unit tests.

//TODO: if there was a decrease include a sentence here about it and talk about a couple of examples from the filled in table

During our test generalization process, we found that there were a number of benefits to include the increase in code coverage, the discovery of new defects, and the reduction in conventional unit tests. For example, there were numerous cases where we could combine a number of conventional unit tests into a single PUT that not only covered the original test cases but provided better code coverage and found new defects. The next three subsections 4.1 Code Coverage, 4.2 Defects, and 4.3 Test Code will explain these in more detail.

4.1. Code Coverage

When using generalized unit test cases, you typically cover more conditions that can drastically increase code coverage as is shown in Table 2.

//TODO: add an example of the largest code coverage increase

This additional code coverage will typically include new blocks of code that were not covered by the original conventional unit tests.

//TODO: add an example of increased block coverage

While PUTs many have increased code coverage and covered new blocks of code this was not the case in every scenario.

//TODO: add an example where coverage is lower than conventional.

In section 8 we describe the limitations of Pex that we experienced during our empirical study.

4.2. Defects

Our generalized conventional unit tests discovered XX new defects that were not detected by the original conven-

Test Class	Test Methods				Coverage		New Blocks	Defects
	Conventional	Amenable	Percent	PUT	Conventional	PUT		
AacFormatTest.cs	5	0	0%	11	67.78%	75.56%	7	3
ApeTest.cs	29	29	100%	106	62.09%	83.88%	154	5
AsfTest.cs	28	28	100%	92	81.62%	77.26%	0	15
DivXTest.cs	7	7	100%	20	88.06%	100%	16	17
Id3V1Test.cs	9	9	100%	2	87.88%	94.55%	11	16

Table 2. Benefits of Test Generalization

tional unit tests.

//TODO: add an example of a new defect detected and explain to include what the PUT looks like

4.3. Test Code

The process of test generalization was able to significantly reduce the test code as shown in Table 2. In a number of cases relating to the different types of Tag classes, we were able to combine multiple conventional unit tests into a single PUT. One example of this is in Figure 4, which is of the pattern type Roundrip, where we were able to combine six separate conventional unit tests into one PUT. This one PUT was able to achieve higher code coverage and discover new defects in the process. Additional details about the example in Figure 4 can be found in section 2. Example.

5. Categorization of Conventional Unit Tests

5.1. Conventional Unit Tests Amenable to Test Generalization

You shall categorize your PUTs generalized from conventional unit tests into the test patterns proposed by de Halleux and Tillmann [3]. You shall list the statistics of your PUTs falling into each pattern.

You shall also propose new patterns to accommodate those PUTs that you cannot categorize into any of the patterns proposed by de Halleux and Tillmann [3]. You shall describe the definition of these new test patterns and the example PUTs for these patterns. If you run out of space, you can refer the readers to the wiki entry links for the details of your new test patterns. Note that you shall at the same time prepare your wiki entries for your new test patterns no matter whether you describe your new patterns here in details.

You shall list the statistics of your PUTs falling into each of your new pattern being proposed by you.

If you cannot find any conventional unit test to be amenable to test generalization, you can state so (but you are expected to find at least one conventional unit test to be amenable to test generalization).

6. Conventional Unit Tests Not Amenable to Test Generalization

You shall summarize the types of conventional unit tests that are not amenable to test generalization. It would be better if you can categorize them into anti-generalization patterns (if so, you shall also create wiki entries for your anti-generalization patterns).

7. Helper Techniques for Test Generalization

7.1. Factory Methods

You shall summarize the cases where you use factory methods to help Pex to generate better test inputs for your generalized PUTs. Again, it would be better if you can summarize patterns or categories for these cases.

If you find no such cases, you can state so.

7.2. Mock Objects

You shall summarize the cases where you use mock objects to deal with some complications that you face in test generalization. You shall categorize these cases into the mock object patterns proposed by de Halleux and Tillmann [3]. If you cannot categorize them into these patterns, propose new patterns and document them in wiki similar to the preceding guidelines for documenting your new normal PUT patterns.

If you find no such cases, you can state so.

7.3. New Assertions

You shall summarize the cases where you add new assertions to the generalized PUTs in order to improve the PUTs to capture more behaviors or properties.

If you find no such cases, you can state so.

8. Limitations of Pex or PUTs

The most substantial limitation we found through test generalization from conventional unit test to PUTS, using Pex, is the amount of time and effort it takes to create good PUTs compared to conventional unit tests. While the conventional unit tests may not cover all of the possible branches they take less time to write. This could change in the future after developers have spent more time using Pex or a similar tool and become more familiar and experts writing PUTs.

Another limitation in using Pex to generate conventional unit tests from PUTs is the values that Pex currently generates do not always represent actual values used in the program. While the values used may help find issues within the method it may not cover a particular set of values you would like to test. This may result in needing a few conventional unit test to supplement the PUTs to ensure your testing the particular values you want to test.

In our empirical study using taglib-sharp, the most consistent limitation we discovered was Pex not understanding the File class that is used extensively through taglib-sharp's library. Pex does try to guess on how to create the file but is unable to do so successfully. In a couple of cases, we use Pex's factory methods to tell Pex to use a sample file that is included with the project's test suite. However, this process has had limited success.

9. Conclusion

In our empirical study, we investigated the benefits of test generalization. We did this through the process of taking existing conventional unit tests and converting them into parameterized unit tests (PUTs). The open source project used for this process was taglib-sharp where we generalized XX conventional unit tests into XX PUTs. From the test generalization process, we found the following benefits: an increase to block coverage, on average, by X.XX% and X new defects found. For future work, to further explore the strengths and weaknesses of test generalization we plan to generalize more conventional unit tests and create new PUTs where necessary. This will allow us to potentially find new defects and increase the overall block coverage.

//TODO: Add a sentence about "identified categories of conventional test cases that are not amenable for test generalization and proposed new PUT patterns" IF true.

References

[1] Agitar agitarone, 2007. http://www.agitar.com/solutions/products/agitarone.html.

[2] M. M. Almasi, H. Hemmati, G. Fraser, and A. Arcuri. An industrial evaluation of unit test generation: Finding real faults in a financial application. In *39th International Conference on Software Engineering, ICSE 2017, Software Engineering in Practice Track*, pages 263–272. IEEE, 2017.

[3] P. de Halleux and N. Tillmann. Parameterized test patterns for effective testing with Pex. Technical report, Microsoft Research, Redmond, WA, October 2010.

[4] Microsoft. Microsoft Research Pex and Moles - Isolation and White Box Unit Testing for .NET, 2016. https://www.microsoft.com/en-us/research/project/pex-and-moles-isolation-and-white-box-unit-testing-fo

[5] Mono. taglib-sharp: Library for reading and writing metadata in media files, 2017. https://github.com/mono/taglib-sharp.

[6] Parasoft Jtest 7.0, 2007. http://www.parasoft.com/jsp/products/home.jsp?product=Jtest.

[7] N. Tillmann and P. de Halleux. Pex - white box test generation for .net. In *Proc. of Tests and Proofs (TAP'08)*, volume 4966, page 134153, Prato, Italy, April 2008. Springer Verlag.