Retrofitting Unit Tests for Parameterized Unit Testing

Madhuri R. Marri*, Suresh Thummalapenta*, Tao Xie*, Nikolai Tillmann†, Jonathan de Halleux†
*Department of Computer Science, North Carolina State University, Raleigh, NC
†Microsoft Research, One Microsoft Way, Redmond, WA
*{mrmarri, sthumma, txie}@ncsu.edu, †{nikolait, jhalleux}@microsoft.com

***Abstract*—Maintenance of software applications requires unit tests with high fault-detection capability. In practice, developers face two major challenges when manually writing unit tests with high fault-detection capability. First, developers may not be able to write test data that test all important behaviors of methods under test. Second, developers may unintentionally write redundant unit tests that exercise the same behavior of methods under test. To address these issues, developers can write Parameterized Unit Tests (PUTs), instead of unit tests without parameters, referred to as Conventional Unit Tests (CUTs). A major benefit of PUTs compared to CUTs is that developers do not need to provide test data in PUTs, since test data can be generated automatically using a test-generation tool. However, writing PUTs is more challenging than writing CUTs. For example, PUTs typically encode algebraic specifications, which are more abstract and general than sample-point behaviors encoded by CUTs. On the other hand, existing applications often include CUTs, which can be used to write PUTs with low effort. To exploit the benefits of PUTs in practice, we propose a novel approach to retrofit existing CUTs into PUTs. We conducted an empirical study on three real-world applications to show the benefits of retrofitting CUTs into PUTs. In our empirical study, we show that our approach retrofits 407 CUTs (4.6 KLOC) to 224 PUTs (4.0 KLOC). Along with achieving higher branch coverage (a maximum increase of 52% for one class under test and 10% for one application) than existing CUTs, our approach helps detect 19 new defects not detected by existing CUTs. Some of these defects are quite complex and difficult to detect using existing automatic test-generation tools alone.**

I. Introduction

Unit tests are widely adopted in software industry for ensuring the high quality of production code. In general, maintenance of software applications requires unit tests with high fault-detection capability. Although automatic test-generation tools [1], [2], [3], [4], [5] can be used to generate unit tests automatically, these tools cannot generate test oracles. Therefore, these tools can detect only robustness-related defects such as `null` pointer dereferencing exceptions [6]. Due to such limitations of automatic test-generation tools, manually writing unit tests is still a common practice in software industry.

When writing unit tests manually, developers can use their domain knowledge in writing test oracles. Nevertheless, it is challenging for developers to write all possible important values for test data (such as argument values of methods invoked in unit tests) that comprehensively exercise methods under test in production code. This challenge in writing test data could result in two major issues. First, developers may not be able to write test data that exercise all important behaviors of methods under test, thereby resulting in unit tests with low fault-detection capability. Second, developers may write different test data that exercise the same behavior of methods under test, thereby resulting in redundant unit tests. These redundant unit tests increase only the testing time and do not increase the fault-detection capability. To show illustrative examples of these issues, consider the three unit tests shown in Figure 1 for testing the `Push` method of an unsigned-integer stack class `UIntStack`. These three unit tests exercise the `Push` method with different test data in different test scenarios. For example, CUT1 and CUT2 exercise `Push` with different argument values, when the stack is empty, while CUT3 exercises `Push`, when the stack is not empty. Consider that there is a defect (in `Push`) that can be detected by passing a negative value as the argument to `Push`. These three tests cannot detect the preceding defect, since these tests do not pass a negative integer value as an argument. Furthermore, CUT2 is a redundant unit test, since `UIntStack` has the same behavior for all non-negative integers passed as arguments to `Push`.

To address these preceding issues with writing unit tests manually, developers can use Parameterized Unit Tests (PUTs) [7] that accept parameters, instead of unit tests that do not accept parameters, referred to as Conventional Unit Tests (CUTs). Indeed, CUTs can be considered as instances of PUTs with test data for parameters. We next give three reasons why writing PUTs is more beneficial than writing CUTs.

```
01:public void CUT1() {
02:  int elem = 1;
03:  UIntStack stk = new UIntStack();
04:  stk.Push(elem);
05:  Assert.AreEqual(1, stk.Count()); }
06:public void CUT2() {
07:  int elem = 30;
08:  UIntStack stk = new UIntStack();
09:  stk.Push(elem);
10:  Assert.AreEqual(1, stk.Count()); }
11:public void CUT3() {
12:  int elem1 = 1, elem2 = 30;
13:  UIntStack stk = new UIntStack();
14:  stk.Push(elem1);
15:  stk.Push(elem2);
16:  Assert.AreEqual(2, stk.Count()); }
```

Fig. 1. Three CUTs that test an unsigned-integer stack that does not accept negative integers.

```
01:public void PUT(int[] elem) {
02:  UIntStack stk = new UIntStack();
03:  foreach (int i in elem) {
04:    stk.Push(i); }
05:  Assert.AreEqual(elem.Length, stk.Count()); }
```

Fig. 2. A single PUT replacing the three CUTs shown in Figure 1.

First, developers do not need to provide *test data* in a PUT and instead need to provide only the variables that represent the test data as parameters. The values for these parameters can be generated automatically using a test-generation tool such as a dynamic symbolic execution (DSE) engine [8], [1], [9], [2]. Section II presents more details on how DSE generates values for parameters of PUTs. Second, since test data are automatically generated by DSE-based approaches tend to exercise all feasible paths in the methods under test, the *fault-detection capability* of PUTs is higher than that of CUTs. Third, a single PUT can represent multiple CUTs, thereby reducing the *size of test code*. For example, the PUT shown in Figure 2 tests the same or more behaviors of the method under test as the three CUTs shown in Figure 1. Using the PUT, a DSE-based approach can test the method under test with other scenarios, such as pushing an element into a stack with two or three elements, to achieve high structural coverage of the method under test. Therefore, using the PUT, a DSE-based approach can automatically generate test data that passes a negative value to the `Push` method based on its implementation, thereby detecting the defect not detected by the existing three CUTs.

Although PUTs help in addressing the issues with CUTs, writing PUTs directly is quite challenging for the following two reasons. First, in general, developers have to define the expected behavior (in terms of *test oracles*) for all possible test data in PUTs, whereas in CUTs, the expected behavior is defined only for a sample test data. In fact, PUTs typically encode algebraic specifications [10]. Second, although PUTs do not encode test data, PUTs still need to encode necessary test scenarios (such as method-call sequences) for exercising the code under test to achieve test objectives, which are described using test oracles. For these two major reasons, PUTs are still not widely adopted in software industry[1].

On the other hand, existing applications often include CUTs [12], and the test oracles and test scenarios encoded in these CUTs can be used to address the two major issues in writing PUTs. To exploit the benefits of PUTs in practice, we propose an approach that assists developers in retrofitting existing CUTs into PUTs. We refer to the process of generalizing CUTs to PUTs as *test generalization*. The key insight of our approach, which helps achieve test generalization, is that existing applications often include CUTs and the test oracles and test scenarios of these CUTs can assist developers in writing PUTs effectively with low effort. Our approach also includes techniques that address the challenges that are classified into two categories: challenges specific to test generalization and challenges in general for the existing DSE-based approaches in generating test data for PUTs. Section III presents more details on these two categories of challenges and describes how our techniques help address these challenges.

In summary, our approach has two major benefits. First, our approach provides a practical solution to adopt PUTs for generating unit tests with high fault-detection capability. Second, even developers or third-party testers with little knowledge of the production code can use our approach to write PUTs with low effort during software maintenance. Our vision is to leverage existing artifacts, such as existing CUTs, in bridging the gap between existing practices in software industry and the new advancements in software-engineering research.

This paper makes the following major contributions:

- The first approach for assisting developers in retrofitting existing CUTs into PUTs for leveraging the benefits of PUTs in practice. To the best of our knowledge, we are the first to propose a test-generalization approach.
- A set of techniques that can assist developers during test generalization in addressing the challenges of test-generalization and the limitations of existing DSE-based approaches.
- The first empirical study to show the benefits of test generalization with three popular open-source applications. Our results show that test generalization increases branch coverage by 4% (with a maximum increase of 52% for one class under test and 10% for one application under analysis) on average for all three applications used in our empirical study. Our results show that test generalization helps detect 19 new defects not detected by existing CUTs. Our results also show that neither the increase in branch coverage nor all these defects can be detected by adding additional tests via automatic test-generation tools such as Randoop [5]. Finally, our results also show that test generalization helps reduce the number of unit tests by 45%; 407 CUTs (4.6 KLOC) are retrofitted into 224 PUTs (4.0 KLOC), thereby reducing the test-code size for potentially helping in better management of test code.

II. Background

We use Pex [11] as an example state-of-the-art DSE-based test generation tool for generating CUTs using PUTs. Pex, a part of Microsoft Visual Studio, is a white-box test generation tool for .NET programs. Pex accepts PUTs and symbolically executes the PUTs and the code under test to generate a set of CUTs that can achieve high coverage of the code under test. Initially, Pex explores the code under test with random or default values and collects constraints along the execution path. Pex next systematically negates parts of the collected constraints and uses a constraint solver to generate concrete values that guide program execution through alternate paths. Pex has been widely used both in academia and industry, which is reflected by its download counts (Feb. 2008 - Oct. 2009) that is greater than 30000. Pex is applied on industrial code bases and detected serious defects in a software component, which had already been extensively tested [2].

III. Approach

We next present our approach that assists developers in achieving test generalization. Although we explain our approach using Pex, our approach is independent of Pex and

[1]The concept of PUTs was introduced in 2005; however, only a few developers currently use PUTs [11]

can be used with other DSE-based test generation tools [9]. Our approach is based on the following two requirements.

- **R1**: the PUT generalized from a passing CUT should not result in false-positive failing CUTs being generated from the PUT.
- **R2**: the PUT generalized from a CUT should help achieve the same or higher structural coverage than the CUT and should help detect the same or higher number of defects than the CUT.

We next describe more details on these two requirements. R1 ensures that test generalization does not introduce false positives. In particular, a CUT generated from the PUT can fail for two reasons: a defect in the method under test (MT) or a defect in the PUT. Failing CUTs for the second reason are considered as false positives. These failing CUTs are generated when generalized PUTs do not satisfy either necessary preconditions of the MT or assumptions on the input domain of the parameters required for passing the test oracle. On the other hand, R2 ensures that test generalization does not introduce false negatives. The rationale is that PUTs provide a generic representation of CUTs, and should be able to guide a DSE-based approach in generating CUTs that exercise the same or more paths in the MT than CUTs, and thereby should have the same or higher fault-detection capability.

We next provide an overview of how a developer generalizes existing CUTs to PUTs by using our approach to satisfy the preceding requirements and then explain each step in detail using illustrative examples from the NUnit framework [13].

A. Overview

Algorithm 1 shows the overview of our approach, which includes five major steps: (S1) *Parameterize*, (S2) *Generalize Test Oracle*, (S3) *Add Assumptions*, (S4) *Add Factory Method*, and (S5) *Add Mock Object*. In our approach, Steps S1 and S2 are mandatory, whereas Steps S3, S4, and S5 are optional and are used when R1 or R2 is not satisfied.

For an MT, the developer uses our algorithm to generalize the set of CUTs of that MT, one CUT at a time. First, the developer identifies concrete values and local variables in the CUT and promotes them as parameters for a PUT (Line 7). Second, the developer generalizes the assertions in the CUT to generalized test oracles in the PUT (Line 8). After generalizing test oracles, the developer applies Pex to generate CUTs from PUTs (Line 9). When any of the generated CUTs fails (Line 11) the developer checks whether the reason for the failing CUT(s) is due to illegal values generated by Pex for the parameters (Line 12), i.e., whether the failing CUTs are false-positive CUTs. To avoid these false-positive CUTs and thereby to satisfy R1, the developer adds assumptions on the parameters to guide Pex to generate legal input values (Line 13). The developer then applies Pex again and continues this process of adding assumptions till either no generated CUTs fail or the generated CUTs fail due to defects in the MT.

After satisfying R1, the developer checks whether R2 is also satisfied, i.e., the structural coverage achieved by generated CUTs is at least as much as the coverage achieved by the existing CUTs. If R2 is satisfied, then the developer proceeds to the next CUT. On the other hand, if R2 is not satisfied, then there could be two issues: (1) Pex was not able to create desired object states for a non-primitive parameter [14], and (2) the MT includes interactions with external environments [15]. Although DSE-based test-generation tools such as Pex are effective in generating CUTs from PUTs whose parameters are of primitive types, Pex or any other DSE-based tool faces challenges in cases such as generating desirable objects for non-primitive parameters. To address these two issues, the developer writes factory methods (Line 21) and mock objects [15] (Line 24), respectively, to assist Pex. More details on these two steps are available in subsequent sections.

The developer repeats the last three steps till the requirements R1 and R2 are met, as shown in Loop 10-29. Often, multiple CUTs can be generalized to a single PUT. Therefore, to avoid generalizing a CUT that is already generated by a previously generalized PUT, the developer checks whether the new CUT to be generalized belongs to already generated

Algorithm 1 Test Generalization

Require: $CUTs$ for an MT M
Ensure: $PUTs$

```
1:  Set PUTs = φ, gAllCUTs
2:  for all c ∈ CUTs do
3:      if gAllCUTs.Contains(c) then
4:          Continue
5:      end if
6:      Set p = φ, gCUTs = φ, break = false
7:      p = Parameterize(c)
8:      p = GeneralizeTestOracle(c, p)
9:      gCUTs = GenerateCUTs(p)
10:     repeat
11:         while !Execute(gCUTs) do
12:             if LegalValueIssue(gCUTs) then
13:                 p = AddAssumptions(p)
14:             else
15:                 ReportDefect()
16:                 Continue
17:             end if
18:         end while
19:         if Cov(M, gCUTs) < Cov(M, CUTs) then
20:             if NPTypeParam(p) then
21:                 p = AddFactoryMethod(p)
22:             end if
23:             if EnviInteractionIssue(M) then
24:                 p = AddMockObj(p)
25:             end if
26:         else
27:             break = true
28:         end if
29:     until break
30:     PUTs.Add(p), gAllCUTs.Add(gCUTs)
31: end for
32: return PUTs
```

```
00:public class SettingsGroup {
01:  MemorySettingsStorage storage; ...
02:  public SettingsGroup(MemorySettingsStorage storage) {
03:    this.storage = storage;
04:  }
05:  public void SaveSetting(string sn, object sv) {
06:    object ov = storage.GetSetting( sn );
07:    //Avoid change if there is no real change
08:    if (ov != null ) {
09:      if (ov is string && sv is string &&
(string)ov == (string)sv ||
10:        ov is int && sv is int && (int)ov == (int)sv ||
11:        ov is bool && sv is bool && (bool)ov == (bool)sv ||
12:        ov is Enum && sv is Enum && ov.Equals(sv))
13:          return;
14:    }
15:    storage.SaveSetting(sn, sv);
16:    if (Changed != null)
17:      Changed(this, new SettingsEventArgs(sn));
18:  }}
```

Fig. 3. The `SettingsGroup` class of the NUnit framework with the `SaveSetting` method under test.

```
00://testGroup is of type SettingsGroup
01:[Test]
02:public void TestSettingsGroup() {
03:  testGroup.SaveSetting("X", 5);
04:  testGroup.SaveSetting("NAME", "Charlie");
05:  Assert.AreEqual(5, testGroup.GetSetting("X"));
06:  Assert.AreEqual("Charlie", testGroup.GetSetting("NAME"));
07:}
```

Fig. 4. A CUT to test the `SaveSetting` method (shown in Figure 3)

CUTs (Lines $3-5$). If so, the developer ignores the new CUT; otherwise, the developer generalizes the new CUT. We next illustrate each step of our approach using an MT and a CUT from the NUnit framework shown in Figures 3 and 4, respectively.

B. Example

1) Method under test and CUTs: Figure 3 shows an MT `SaveSetting` from the `SettingsGroup` class of the NUnit framework. The `SaveSetting` method accepts a setting name `sn` and a setting value `sv`, and stores the setting in a storage (represented by the member variable `storage`). The setting value can be of type `int`, `bool`, `string`, or `enum`. Before storing the value, `SaveSetting` checks whether the same value already exists for that setting in the storage. If the same value already exists for that setting, `SaveSetting` returns without making any changes to the storage.

Figure 4 shows a CUT for testing the `SaveSetting` method. The CUT saves two setting values (of types `int` and `string`) and verifies whether the values are set properly using the `GetSetting` method. The CUT verifies the expected behavior of the `SaveSetting` method only for the setting values of types `int` and `string`. This CUT is the only test for verifying `SaveSetting` and includes two major issues. First, the CUT does not verify the behavior for the types `bool` and `enum`. Second, the CUT does not cover the `true` branch in Statement 8 of Figure 3. The reason is that the CUT does not invoke the `SaveSetting` method more than once with the same setting name. This CUT achieves 10% branch coverage of the `SaveSetting` method. We next explain how the developer generalizes the CUT to a PUT and addresses these two major issues via our test generalization.

2) S1 - Parameterize: For the CUT shown in Figure 4, the developer promotes the `string` "`Charlie`" and the `int` `5` as

```
//PAUT: PexAssumeUnderTest
00:[PexMethod]
01:public void TestSettingsGroupPUT([PAUT] SettingsGroup st,
02:  [PAUT] string sn, [PAUT] object sv) {
03:    st.SaveSetting(sn, sv);
04:    PexAssert.AreEqual(sv, st.GetSetting(sn));}
```

Fig. 5. A PUT for the CUT shown in Figure 4.

```
//MSS: MemorySettingsStorage (class)
//PAUT: PexAssumeUnderTest (Pex attribute)
00:[PexFactoryMethod(typeof(MSS))]
01:public static MSS Create([PAUT]string[]
02:     sn, [PAUT]object[] sv) {
03:    PexAssume.IsTrue(sn.Length == sv.Length);
04:    PexAssume.IsTrue(sn.Length > 0);
05:    MSS mss = new MSS();
06:    for (int count = 0; count < sn.Length; count++) {
07:      mss.SaveSetting(sn[count], sv[count]);
08:    }
09:    return mss;
10:}
```

Fig. 6. An example factory method for the type `MemorySettingsStorage`.

a single parameter of type `object` for the PUT. The advantage of replacing concrete values with symbolic values (in the form of parameters) is that Pex generates concrete values based on the constraints encountered in different paths in the MT. Since `SaveSetting` accepts the parameter of type `object` (shown in Figure 5), Pex automatically identifies the possible types for the `object` type such as `int` or `bool` from the MT and generates concrete values for those types, thereby satisfying R2. In addition to promoting concrete values as parameters of PUTs, the developer promotes other local variables such as the receiver object (`testGroup`) of `SaveSetting` as parameters. Promoting such receiver objects as parameters can help generate different object states (for those receiver objects) that can help cover additional paths in the MT. Figure 5 shows the PUT generalized from the CUT shown in Figure 4.

3) S2 - Generalize Test Oracles: The developer next generalizes test oracles in the CUT. In the CUT, a setting is stored in the storage using `SaveSetting` and is verified using `GetSetting`. By analyzing the CUT, the developer generalizes the test oracle of the CUT by replacing the constant value with the relevant parameter of the PUT. The test oracle for the PUT is shown in Line 4 of Figure 5.

In practice, generalizing the test oracle is a complex task, since determining the expected output values for all the generated inputs is not trivial. Therefore, to assist developers in generalizing test oracles, we proposed 15 PUT patterns, which developers can use to analyze the existing CUTs and generalize test oracles. More details of the patterns are available in Pex documentation [16].

4) S3 - Add Assumptions: A challenge faced during test generalization is that Pex requires guidance in generating legal values for the parameters of PUTs. These legal values are the values that satisfy preconditions of the MT and help setting up test scenarios to pass test assertions (i.e., test oracles). These assumptions help avoid generating false-positive CUTs, thereby satisfying R1. For example, without any assumptions, Pex by default generates illegal `null` values for non-primitive parameters such as `st` of the PUT shown in Figure 5. To guide Pex in generating legal values, the developer adds sufficient assumptions to the PUT. In the PUT, the developer annotates

```
01:public class MockXmlTextWriter {
02:  public MockXmlTextWriter(string filename,
                Encoding encoding) {
03:     this.fileName = filename;}

05:  public void WriteAttributeString(string att,
                              string val) {
06:     xml = xml + " " + att + "=" + "" + val + "";}

07:  public void Close() {
08:     xml = xml.Replace("/> />", "/>" + System.
          Environment.NewLine + "</" + startString + ">");
09:     CreatedProjects.currentProject = xmlString;}
10:}
```

Fig. 7. Sample code from the `MockXmlTextWriter` mock object.

each parameter with the tag `PexAssumeUnderTest`[2], which describes that the parameter should not be `null` and the type of generated objects should be the same as the parameter type. The developer adds further assumptions to PUTs based on the behavior exercised by the CUT and the feedback received from Pex.

5) S4 - Add Factory Method: In general, Pex (or any other existing DSE-based approaches) faces challenges in generating CUTs from PUTs that include parameters of non-primitive types, since these parameters require method-call sequences (that create and mutate objects of non-primitive types) to generate desirable object states [14]. These desirable object states are the states that are required to exercise new paths or branches in the MT, thereby to satisfy R2. For example, a desirable object state to cover the `true` branch of Statement 8 in Figure 3 is that the `storage` object should already include a value for the setting name `sn`.

To assist Pex in producing effective method-call sequences that can help achieve desirable object states, developers write method-call sequences inside factory methods, supported by Pex. The factory methods can have parameters and Pex handles these parameters similar to the parameters of PUTs. Figure 6 shows an example factory method for the `MemorySettingsStorage` class. The factory method accepts two arrays of setting names (`sn`) and values (`sv`), and adds these entries to the storage. This factory method helps Pex to generate method-call sequences that can create desirable object states. For example, Pex can generate five names and five values as arguments to the factory method for creating a desirable object state with five elements in the storage[3]. The same factory method can be reused for all other PUTs using the `MemorySettingsStorage` class as a parameter type.

6) S5 - Add Mock Object: Pex (or any other existing DSE-based approaches) also faces challenges in handling PUTs or MT that interacts with the external environment such as the file system. To address this challenge related to the interactions with the environment, developers write mock objects for assisting Pex [15]. These mock objects help test features in isolation especially when PUTs or MT interact with environments such as a file system. We next describe how developers use mock objects with an illustrative example.

[2]`PexAssumeUnderTest` is a custom attribute provided by Pex.

[3]Note that the factory methods provide only assistance to Pex in achieving the desirable object states, and Pex generates these object states based on the branching conditions in the MT.

In the `NUnitProject` class of NUnit, the `save` method writes configuration information using the `XmlTestWriter` class to an XML file. This XML file is expected to be created when the project is created, i.e., when an instance of `NUnitProject` is created. Two existing CUTs `SaveEmptyConfigs` and `SaveNormalProject` test this `save` method. These CUTs add configurations to the XML project files and assert whether the files are saved in the right format, and contain the added configuration information. Both the CUTs use on a default instance (of `NUnitProject`) that is created using the test setup method. Therefore, the requirement to test the `save` method is to provide the project configuration file (the XML file) in a *specific location*, i.e., the directory location where the project is saved (when a project is saved, a new directory is created as the project directory and the XML file is created in this directory).

To generalize these CUTs, the developer promotes the project path (the parameter of the `save` MT and a local variable in the CUTs) as a parameter to the PUT. However, generalization of these CUTs is not straightforward. By promoting the project path as the PUT's parameter, all generated CUTs require XML files in specific locations (reflected by generated values for the parameter). When the `save` method is invoked, these XML files should be available for being accessed using `XmlTextWriter`. Without such XML files, the `save` method throws an exception, resulting false-positive CUTs, thereby violating R1.

To avoid the complexity of creating a "real" file at a "real" location for each generated CUT and to satisfy R1, the developer mocks the expected behavior of `XmlTextWriter` to `MockXmlFileWriter` (shown in Figure 7). This mock object simulates the behavior of `XmlTextWriter`. However, unlike the real object, the mock object uses a `string` field and appends the input text to the field. Thus the mock object replicates the actual behavior of `XmlTextWriter` while avoiding the interactions with the physical file system.

7) Generalized PUT: Figure 5 shows the final PUT after the developer follows our approach. The PUT accepts three parameters: an instance of `SettingsGroup`, the name of the setting, and its value. The `SaveSetting` method can be used to save either an `int` value or a `string` value (the method accepts both types for its arguments). Therefore, the CUT requires two method calls shown in Statements 3 and 4 of Figure 4 to verify whether `SaveSetting` correctly handles these types. On the other hand, only one method call is sufficient in the PUT, since the variable is promoted to a PUT parameter of type `object`. Pex automatically explores the MT and generates CUTs that cover both `int` and `string` types. Indeed, the `SaveSetting` method also accepts `bool` and `enum` types. The existing CUTs did not include test data for verifying these two types. Our generalized PUT automatically handles these additional types, highlighting a primary advantage of test generalization in reducing the test code significantly without reducing the behavior exercised by existing CUTs.

When we applied Pex on the PUT shown in Figure 5,

Pex generated 8 CUTs from the PUT. These CUTs test the `SaveSetting` method with different setting values of types such as `int` or `string` or other non-primitive object types. As described earlier, a single PUT can substitute multiple CUTs, resulting in reduced test code. Furthermore, the CUT used for generalization achieved branch coverage of 10%, whereas the CUTs generated from the generalized PUT achieved branch coverage of 90%. Although the PUT achieved higher code coverage compared to the existing CUT, the PUT still could not cover the `true` branch of Statement 16 of the `SaveSetting` method (Figure 3). The developer while doing generalization can analyze these not-covered portions and then either enhance PUTs or write new PUTs for achieving additional coverage of those not-covered portions[4].

IV. Empirical Study

We conducted an empirical study using three real-world applications to show the benefits of our approach in generalizing CUTs to PUTs. In our empirical study, we show the benefits of PUTs over existing CUTs using *four metrics*: branch coverage, the number of detected defects, the number of tests (their LOC) being reduced by test generalization, and the time taken for test generalization. In particular, we address the following four research questions in our empirical study:

- **RQ1: Branch Coverage.** How much higher percentage of *branch coverage* is achieved by PUTs compared to existing CUTs? Since PUTs are a generalized form of CUTs, this research question helps to address whether PUTs can achieve additional branch coverage compared to CUTs. We focus on branch coverage, since detecting defects via violating test assertions in unit tests can be mapped to covering implicit branches within those test assertions.
- **RQ2: Defect Detection.** How many new *defects* (that are not detected by CUTs) are detected by PUTs and vice-versa? This research question helps to address whether PUTs have higher fault-detection capabilities compared to CUTs.
- **RQ3: Test-code-size Reduction.** How many tests are reduced by generalizing CUTs to PUTs? This research question addresses whether test generalization helps reduce the test-code size for potentially helping better management of test code.
- **RQ4: Generalization Effort.** How much effort required for generalizing CUTs to PUTs? This research question helps to show that the effort required for generalization is worthwhile, considering the generalization benefits.

We first present the details of subject applications and next describe our setup for empirical study. Finally, we present the results of our empirical study. The detailed results of our empirical study are available at our project website https://sites.google.com/site/asergrp/projects/putstudy.

[4]Recall that the objective of our study is to generalize existing CUTs to PUTs for comparing the benefits of PUTs over existing CUTs. Therefore, here we do not show the step of writing additional PUTs that can achieve additional branch coverage.

A. Subject Applications

We use three popular open source applications (as shown by their download counts in their hosting web sites) in our study: NUnit [13], DSA [17], and Quickgraph [18]. Table I(a) shows the three subject applications. NUnit [13], a counterpart of JUnit for Java, is a widely used open source unit-testing framework for all .NET languages. Data Structures and Algorithms (DSA) [17] is a library that contains implementation of data structures and algorithms, a few of which are not available in the NET 3.5 framework. QuickGraph [18] is a C# graph library that provides various directed/undirected graph data structures. While we used all namespaces and classes for DSA and QuickGraph in our study, for NUnit, we used nine classes from its `Util` namespace, which is one of the core components of the framework.

Table I(b) shows the characteristics of the three subject applications. Column "Downloads" shows the number of downloads of the application (as listed in its hosting web site). Column "Code Under Test" shows details of the code under test (of the application) in terms of the number of classes ("#Classes"), number of methods ("#Methods"), number of lines of code ("#KLOC"), and the average and maximum cyclometic complexity of the code under test. Similarly, Table I(c) shows the statistics of existing CUTs in these subject applications.

B. Empirical Study Setup

We next describe the setup of our study conducted by the first and second authors of this paper for addressing the preceding research questions. The authors are the third and fourth year PhD students, respectively, with the same experience of two years with PUTs and Pex. Before joining the PhD program, the authors had three and five years of programming experience, respectively, in software industry. Each of the author conducted test generalization for half of CUTs across all three subjects. The authors do not have the knowledge of subject applications and conducted the study as third-party testers. We expect that our test-generalization results can be much better, if the test generalization is performed by the developers of these subject applications. The reason is that these developers can incorporate their domain knowledge during test generalization to write more effective PUTs.

To address the preceding research questions, the authors used three categories of CUTs. The first category of CUTs is the set of existing CUTs available with subject applications. The second category of CUTs is set of CUTs generated from PUTs. To generate this second category of CUTs, the authors generalized existing CUTs to PUTs and applied Pex on those PUTs. The authors also measured the time taken for generalizing each CUT to compute the generalization effort for addressing RQ4. The measured time includes the amount of time taken for performing all steps described in our approach and also applying Pex to generate CUTs from PUTs. The third category of CUTs is the set of existing CUTs + new CUTs (hereby referred to as *RTs*) that were generated using an automatic random test-generation tool, called Randoop [5].

TABLE I

(a) (b) Characteristics of subject applications. (c) Statistics of existing CUTs.

Subject Applications	Downloads	Code Under Test					Existing Test Code		
		#Classes	#Methods	#KLOC	Avg. Complexity	Max. Complexity	#Classes	#CUTs	#KLOC
NUnit	193,563[5]	9	87	1.4	1.48	14	9	49	0.9
DSA	2241	27	259	2.4	2.09	16	20	337	2.5
QuickGraph	7969	56	463	6.2	1.79	16	9	21	1.2

Subject	Branch Coverage			Overall Inc.	Max. Inc.
	CUTs	CUTs + RTs (#)	PUTs		
NUnit	78%	78% (144)	88%	10%	52%
DSA	91%	91% (615)	92%	1%	1%
QuickGraph	87%	88% (3628)	89%	2%	11%

TABLE II
BRANCH COVERAGE ACHIEVED BY THE EXISTING CUTS, CUTS + RTS, AND CUTS GENERATED BY PEX USING THE GENERALIZED PUTS.

```
01: public void RemoveSetting(string settingName) {
02:  int dot = settingName.IndexOf( '.' );
03:    if (dot < 0)
04:     storageKey.DeleteValue(settingName, false);
05:    else {
06:     using(RegistryKey subKey = storageKey.OpenSubKey(
              settingName.Substring(0,dot),true)) {
07:       if (subKey != null)
08:           subKey.DeleteValue(
                    settingName.Substring(dot + 1)); }
09:    }
10: }
```

Fig. 8. `RemoveSetting` method whose coverage is increased by 60% due to test generalization.

This third category (CUTs + RTs) helps show that the benefits of test generalization cannot be achieved by simply generating additional tests using tools such as Randoop. To address RQ1, the authors measured branch coverage using a coverage measurement tool, called NCover[6]. To address RQ2 and RQ3, the authors measured the number of failing tests and computed the code metrics (LOC) using CLOC[7] tool, respectively. The authors did not compare the execution time of CUTs for all three categories, since the time taken for executing CUTs of all categories is negligible ($<$ 20 sec).

C. RQ1: Branch Coverage

We next describe our empirical results for addressing RQ1. Table II shows the branch coverage achieved by executing the existing CUTs, CUTs + RTs, and the CUTs generated by Pex using the generalized PUTs. The values in brackets (#) for CUTs + RTs indicate the number of RTs, i.e., the tests generated by Randoop (using a default timeout of 180 sec). Column "Overall Inc." shows the overall increase in the branch coverage from the existing CUTs to the generalized PUTs. Column "Max. Inc." shows the maximum increase for a class or namespace in the respective subject applications.

Column "Overall Inc." shows that the branch coverage is increased by 10%, 1%, and 2% for NUnit, DSA, and QuickGraph, respectively. Furthermore, Column "Max Inc." shows that the maximum branch coverage for a class or a namespace is increased by 52%, 1%, and 11% for NUnit, DSA, and QuickGraph, respectively. One major reason for not achieving an increase in the coverage for DSA is that the existing CUTs already achieved high branch coverage and PUTs help achieve a little higher coverage than existing CUTs.

To show that the increase in the branch coverage achieved by PUTs is not trivial to achieve, we compare the results of PUTs with CUTs + RTs. The increase in the branch coverage achieved by CUTs + RTs compared to CUTs alone is 0%, 0%, and 1% for NUnit, DSA, and QuickGraph, respectively. This comparison shows that the improvement in the branch coverage achieved by PUTs is not trivial to achieve, since the branches that are not covered by the existing CUTs are generally quite difficult to cover (as shown in the results of CUTs + RTs).

D. RQ2: Defects

To address RQ2, we identify the number of defects detected by PUTs. We did not find any failing CUTs among existing CUTs of the subject applications. Therefore, we consider the defects detected by failing tests among the CUTs generated from PUTs as new defects not detected by existing CUTs. In addition to the defects detected by PUTs, we also inspect the failing tests among the RTs to compare the fault-detection capabilities of PUTs and RTs.

In summary, our PUTs found 15 new defects in DSA and 4 new defects in NUnit. After our inspection, we reported the failing tests on their hosting websites[8]. On the other hand, RTs include 90, 25, and 738 failing tests for DSA, NUnit, and QuickGraph, respectively. Since RTs are generated automatically using Randoop, RTs do not include test oracles. Therefore, an RT is considered as a failing test, if the execution of RT results in an uncaught exception being thrown. In our inspection of these failing tests in RTs, we found that only 18 failing tests for DSA are related to 4 real defects in DSA, since the same defect is detected by multiple failing tests. These 4 defects are also detected by our PUTs. The remaining failing tests are due to two major issues. First, exceptions raised by RTs are expected. In our approach, we address this issue by adding annotations to PUTs regarding expected exceptions. We add these additional annotations based on expected exceptions in CUTs. Second, illegal test data such as `null` values passed as arguments to methods invoked in RTs. In our approach, we address this issue of illegal test data by adding assumptions to PUTs in Step S1. This issue of illegal test data in RTs show the significance of Step S1 in our approach.

To further show the significance of generalized PUTs, we applied Pex on these applications without these PUTs and by

[6]http://www.ncover.com/

[7]http://cloc.sourceforge.net/

[8]Reported bugs can be found at the DSA CodePlex website with defect IDs from 8846 to 8858 and the NUnit SourceForge website with defect IDs 2872749, 2872752, and 2872753.

```
//To test Remove item not present
01: public void RemoveCUT() {
02:  Heap<int> actual = new Heap<int> {2, 78, 1, 0, 56};
03:  Assert.IsFalse(actual.Remove(99));
04: }
```

Fig. 9. Existing CUT to test the `Remove` method of `Heap`.

```
01: public void RemoveItemPUT (
            [PAUT]List<int> input, int item) {
02:  Heap<int> actual = new Heap<int> (input);
03:  if (input.Contains(item)) {
04:     ..... }
05:  else {
06:    PexAssert.IsFalse(actual.Remove(randomPick));
07:    PexAssert.AreEqual(input.Count, actual.Count);
08:    CollectionAssert.AreEquivalent(actual, input);}
09: }
```

Fig. 10. A generalized PUT of the CUT shown in Figure 9.

using *PexWizard*. PexWizard is a tool provided with Pex and this tool automatically generates PUTs (without test oracles) for each public method in the application. We found that the generated CUTs include 23, 170, and 17 failing tests for DSA, NUnit, and QuickGraph, respectively. However, similar to Randoop, only 2 tests are related to 2 real defects (also detected by our generalized PUTs) in DSA, and the remaining failing tests are due to the preceding two issues described for Randoop.

We next explain an example defect detected in the `Heap` class of the DSA application by CUTs generated from generalized PUTs. The details of remaining defects can be found at our project website. The `Heap` class is a heap implementation in the `DataStructure` namespace. This class includes methods to add, remove, and heapify the elements in the heap. The `Remove` method of the class takes an item to be removed as a parameter and returns `true` when the item to be removed is in the heap, and returns `false` otherwise. Figure 9 shows the existing CUT that checks whether the `Remove` method returns `false` when an item that is not in the heap is passed as the parameter. On execution, this CUT passed; exposing no defect in the code under test and there are no other CUTs (in the test suite) that exercise the behavior of the method. However, from our generalized PUT shown in Figure 10, a few of the generated CUTs failed, exposing a defect in the `Remove` method. The test data for the failing tests had the following common characteristics: the heap size is less than 4 (the `input` parameter of the PUT is of size less than 4), the item to be removed is 0 (the `item` parameter of the PUT), and the item 0 was not already added to the heap (the generated value for `input` did not contain the item 0).

When we inspected the causes of the failing tests, we found that in the constructor of the `Heap` class, a default array of size 4 (of type `int`) is created to store the items. In C#, an integer array is by default assigned values zero to the elements of the array. Therefore, there is always an item 0 in the heap unless an input list of size greater than or equal to 4 is passed as parameter. Therefore, on calling the `Remove` method to remove the item 0, even when there is no such item in the heap, the method returns `true` indicating that the item has been successfully removed and causing the assertion statement to fail (Statement 6 of the PUT). However, this defect was not detected by the CUT shown in Figure 9 since the unit test assigns the heap with 5 elements (Statement 2) and

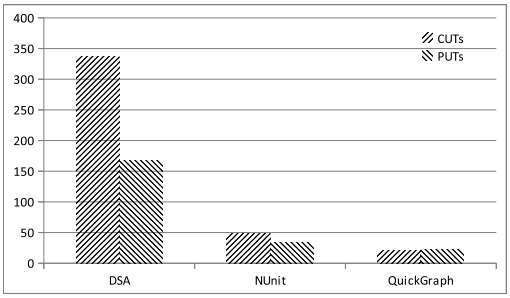

Fig. 11. Comparison of the number of CUTs and PUTs

therefore the defect-exposing scenario of heap size < 4 cannot be exercised. These 19 new defects that were not detected by the existing CUTs show that PUTs are an effective means for rigorous testing of the code under test.

E. RQ3: Test-code-size Reduction

We next address RQ3 of whether test generalization can help reduce the test-code size. We use two metrics to address this research question. First, we compare the number of CUTs and the number of PUTs. Second, we compare the LOC of CUTs and PUTs. The reason for the second metric is that a low number of PUTs with a high number of LOC does not help in reducing the test-code size.

Figure 11 shows the comparison of the number of CUTs with PUTs for all subject applications. The x-axis shows the subject application and y-axis shows the number of CUTs or PUTs. In total, we generalized 407 CUTs to 224 PUTs that achieved higher branch coverage than CUTs and also detected new defects that are not detected by the CUTs. The figure shows the reduction in the number of tests for the subjects DSA and NUnit, since often multiple CUTs are generalized to a single PUT. For example, Figures 1 and 2 show two CUTs and the corresponding single generalized PUT, respectively.

We observed that reduction in the number of unit tests can help in better management of test code. For example, consider the PUT shown in Figure 10. This PUT replaces three CUTs and detects a new defect (described in Section IV-D) in the DSA application. A fix for that defect is to add a new constructor (for the `Heap` class) that accepts size of the heap as a parameter. However, such fix requires changing three CUTs. Although refactoring tools can be used to automatically replace the old constructor with the new constructor in all three CUTs, the developer still needs to provide test data for the new parameter in those three CUTs. Instead, with test generalization, it is sufficient to change one PUT, since the PUT represents all three CUTs, and the developer does not need to provide test data. In our study, we also observed an exceptional case, where a CUT is generalized to more than one PUT. Section V discusses more about this exceptional case.

Figure 13 shows the results of comparing the LOC of CUTs and PUTs. For DSA and QuickGraph, the LOC of PUTs is less than the LOC of CUTs, whereas for NUnit, the LOC of PUTs is slightly more than the LOC of CUTs. Section V describes why the LOC of PUTs for NUnit is slightly more than the LOC of CUTs. As shown in Figure 11, for QuickGraph, although there is no reduction in the number of CUTs, the LOC of

```
//PAUT = PexAssumeUnderTest
01:public void AddFirstTest([PAUT]SinglyLinkedList<int> sll,
          [PAUT]int[] ne) {
02:    PexAssume.IsTrue(ne.Length > 1);
03:    PexAssume.IsTrue(sll.Count == 0);
04:    for (int i = 0; i < ne.Length; i++)
05:        sll.AddFirst(ne[i]);
06:    PexAssert.AreEqual(ne[ne.Length - 1], sll.Head.Value);
07:    PexAssert.AreEqual(ne[0], sll.Tail.Value);
08:    PexAssert.AreEqual(ne.Length, sll.Count);}
```

Fig. 12. PUT for the `AddFirst` method under test.

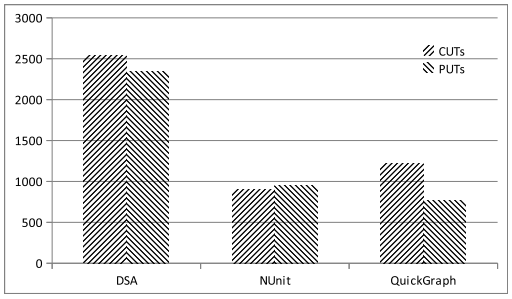

Fig. 13. Comparison of Lines of Code of CUTs and PUTs.

PUTs is reduced by 37%, showing the significance of test generalization. For DSA, the reduction in the LOC (7.8%) is not as significant as the reduction in the number of tests (50.2%). We identify that among new LOC written for PUTs, many statements are related to the additional `using` statements or new annotations that do not affect the effort in maintaining test code. Therefore, our results show that test generalization can help reduce the number of tests (and LOC) and thereby reduce the efforts in maintaining test code.

F. RQ4: Generalization Effort

We next address RQ4 regarding the manual effort required for the generalization of CUTs to PUTs. Both authors equally split the existing CUTs of all three subject applications. The cumulative effort of both the authors in conducting the study is 2.8, 13.8, and 1.5 hours for subject applications NUnit, DSA, and QuickGraph, respectively. Our measured timings are primarily dependent on four factors: expertise with PUTs and Pex tool, knowledge of the subject applications, number of CUTs and the number of transformed PUTs, and the complexity of a CUT or a transformed PUT. Although the authors have experience with PUTs and using Pex, the authors do not have the knowledge of these subject applications. Therefore, we expect that the developers of these subject applications, although unfamiliar with PUTs or Pex, may take similar amount of effort. Overall, our results show that the effort of test generalization is worthwhile considering the benefits that can be gained through generalization.

V. Discussion and Future work

In our results related to test-code maintenance, there is a scenario where the number of PUTs is more than the number of CUTs. The reason is that it is sometimes difficult to generalize test oracles in a few cases. For example, consider the following CUT:

```
public void Canonicalize() {
  PexAssert.AreEqual(@"C:/folder1/file.tmp",
     PathUtils.Canonicalize(@"C:/folder1/./folder2/
     ../file.tmp")); }
```

The `Canonicalize` method in `PathUtils` accepts a `string` parameter and uses a complex procedure to transform the input into a standard form. It is easy to identify the expected output for concrete strings such as `C:/folder1/.../folder2/.../file.tmp`. However, when the CUT is generalized with a parameter for the input string, it is challenging to identify the expected output. Although a developer can provide an alternative implementation for the `Canonicalize` method, the amount of required effort could be higher than the effort required to write the implementation of the actual method under test. To address this issue in our empirical study, we split the CUT into multiple PUTs during test generalization. For those PUTs with difficulties in generalizing test oracles, we do not generalize the test assertions, and instead replace assertions with statements that print outputs suitable for manual review. These PUTs can still help in detecting defects related to exceptions such as `null` pointer exceptions.

For NUnit, the LOC of PUTs is more than the LOC of CUTs. The reason is that to generalize 2 CUTs to PUTs, the respective test objectives required *specialized* test data, such as that a tree structure of keys was required to test the `ClearTestKey` method of the `NUnitRegistry` class. In order to create such specialized test data, we created the required test setup that increased the LOC of PUTs for NUnit.

VI. Related Work

Pex [2] accepts PUTs and uses dynamic symbolic execution to generate test inputs. Although we use Pex terminology in describing our procedure, our procedure is independent of Pex and can be applied with other testing tools that accept unit tests with parameters such as JUnitFactory [19] for Java testing. Other existing tools such as Parasoft Jtest [20] and CodeProAnalytiX [21] adopt the design-by-contract approach [22] and allow developers to specify method preconditions, postconditions, and class invariants for the unit under test and carry out symbolic execution or random testing to generate test inputs. More recently, Saff et al. [23] propose theory-based testing and generalize six Java applications to show that the proposed theory-based testing is more effective compared to traditional example-based testing. A theory is a partial specification of a program behavior and is a generic form of unit tests where assertions should hold for all inputs that satisfy the assumptions specified in the unit tests. A theory is similar to a PUT and Saff et al.'s approach uses these defined theories and applies the constraint solving mechanism based on path coverage to generate test inputs similar to Pex. In contrast to our study, their study does not provide a systematic procedure of writing generalized PUTs or show empirical evidence of benefits of PUTs as shown in our study.

There are existing approaches [3], [5], [24] that can automatically generate required method-call sequences that achieve different object states. However, in practice, each approach has its own limitations. For example, Pacheco et al.'s approach [5] generates method-call sequences randomly by incorporating

feedback from already generated method-call sequences. However, such a random approach can still face challenges in generating desirable method-call sequences, since often there is little chance of generating required sequences at random. In our test generalization, we manually write factory methods to assist Pex in generating desirable object states for non-primitive data types, when Pex's demand-driven strategy faces challenges.

In our previous work [15], we presented an empirical study to analyze the use of parameterized mock objects in unit testing with PUTs. We showed that using a mock object can ease the process of unit testing and identified challenges faced in testing code when there are multiple APIs that need to be mocked. In our current study, we also use mock objects in our testing with PUTs. However, our previous study showed the benefits of mock objects in unit testing, while our current study shows the use of mock objects to help achieve test generalization. In our other previous work with PUTs [25], we propose mutation analysis to help developers in identifying likely locations in PUTs that can be improved to make more general PUTs. In contrast, our current study suggests a systematic procedure of retrofitting CUTs for parameterized unit testing.

VII. Threats to validity

The threats to external validity primarily include the degree to which the subject programs, defects, and CUTs are representative of true practice. The subject applications used in our empirical study range from small-scale to medium-scale applications that are widely used in the software industry as shown by their number of downloads. We tried to alleviate the threats related to detected defects by inspecting the source code and by reporting the defects to the developers of the application under test. These threats could further be reduced by conducting more studies with wider types of subjects in our future work. The threats to internal validity are due to manual process involved in generalizing CUTs to PUTs. Our study results can be biased based on our experience and knowledge of the subject applications. These threats can be reduced by conducting more case studies with more subject applications and other human subjects. The results in our study can also vary based on other factors such as the effectiveness of generalized PUTs and test-generation capability of Pex.

VIII. Conclusion

Recent advances in software testing introduced parameterized unit tests (PUTs) [7], which are a generalized form of conventional unit tests (CUTs). With PUTs, developers do not need to provide test data (in PUTs), which are generated automatically using a test-generation approach. Although PUTs are more beneficial than CUTs, PUTs are still not widely adopted in software industry. In this paper, we proposed an approach for leveraging the benefits of PUTs and increasing the fault-detection capability of existing CUTs with low effort via test generalization. Our empirical results show that test generalization helped reduce 407 CUTs (4.6 KLOC) to 224 PUTs (4.0 KLOC). Along with achieving higher branch coverage (a maximum increase of 52% for one class under test and 10% for one application under analysis), test generalization helped detect 19 new defects not detected by existing CUTs. In future work, we plan to automate our approach to further reduce the manual effort required for test generalization.

References

[1] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proc. PLDI*, 2005, pp. 213–223.

[2] N. Tillmann and J. de Halleux, "Pex - white box test generation for .NET," in *Proc. TAP*, 2008, pp. 134–153.

[3] C. Csallner and Y. Smaragdakis, "JCrasher: an automatic robustness tester for Java," *Softw. Pract. Exper.*, vol. 34, no. 11, 2004.

[4] C. Pacheco and M. D. Ernst, "Eclat: Automatic generation and classification of test inputs," in *Proc. ECOOP*, 2005, pp. 504–527.

[5] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proc. ICSE*, 2007, pp. 75–84.

[6] M. d'Amorim, C. Pacheco, T. Xie, D. Marinov, and M. D. Ernst, "An empirical comparison of automated generation and classification techniques for object-oriented unit testing," in *Proc. ASE*, 2006, pp. 59–68.

[7] N. Tillmann and W. Schulte, "Parameterized Unit Tests," in *Proc. ESEC/FSE*, 2005, pp. 253–262.

[8] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[9] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *Proc. ESEC/FSE*, 2005, pp. 263–272.

[10] J. Henkel and A. Diwan, "Discovering algebraic specifications from java classes," in *Proc. ECOOP*, 2003, pp. 431–456.

[11] "Pex - automated white box testing for .NET," 2009, http://research.microsoft.com/Pex/.

[12] B. Daniel, V. Jagannath, D. Dig, and D. Marinov, "ReAssert: Suggesting repairs for broken unit tests," in *Proc. ASE*, 2009, pp. 433–444.

[13] C. P. Jamie Cansdale, Gary Feldman and M. C. Two, "NUnit," 2002, http://nunit.com/index.php.

[14] S. Thummalapenta, T. Xie, N. Tillmann, P. de Halleux, and W. Schulte, "MSeqGen: Object-oriented unit-test generation via mining source code," in *Proc. ESEC/FSE*, 2009, pp. 193–202.

[15] M. R. Marri, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "An empirical study of testing file-system-dependent software with mock objects," in *Proc. AST, Business and Industry Case Studies*, 2009, pp. 149–153.

[16] "Pex Documentation," 2010, http://research.microsoft.com/Pex/documentation.aspx.

[17] Granville and L. D. Tongo, "Data structures and algorithms," 2006, http://dsa.codeplex.com/.

[18] J. de Halleux, "Quickgraph, graph data structures and algorithms for .NET," 2006, http://quickgraph.codeplex.com/.

[19] "Agitar JUnit Factory," 2008, http://www.agitar.com/developers/junit_factory.html.

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

[21] "CodePro AnalytiX," 2009, http://www.eclipse-plugins.info/eclipse/plugin_details.jsp?id=943.

[22] B. Meyer, *Object-Oriented Software Construction*. Prentice Hall PTR, 2000.

[23] D. Saff, M. Boshernitsan, and M. D. Ernst, "Theories in practice: Easy-to-write specifications that catch bugs," MIT Computer Science and Artificial Intelligence Laboratory, Tech. Rep. MIT-CSAIL-TR-2008-002, 2008, http://www.cs.washington.edu/homes/mernst/pubs/testing-theories-tr002-abstract.html.

[24] S. Khurshid, C. S. Pasareanu, and W. Visser, "Generalized symbolic execution for model checking and testing," in *Proc. TACAS*, 2003, pp. 553–568.

[25] T. Xie, N. Tillmann, P. de Halleux, and W. Schulte, "Mutation analysis of parameterized unit tests," in *Proc. Mutation*, 2009, pp. 177–181.