Retrofitting Unit Tests for Parameterized Unit Testing

Suresh Thummalapenta[1], Madhuri R. Marri[1], Tao Xie[1],
Nikolai Tillmann[2], and Jonathan de Halleux[2]

[1] Department of Computer Science, North Carolina State University, Raleigh, USA
{sthumma,mrmarri,txie}@ncsu.edu
[2] Microsoft Research, One Microsoft Way, Redmond, USA
{nikolait, jhalleux}@microsoft.com

Abstract. Recent advances in software testing introduced *parameterized unit tests* (PUT), which accept parameters, unlike conventional unit tests (CUT), which do not accept parameters. PUTs are more beneficial than CUTs with regards to fault-detection capability, since PUTs help describe the behaviors of methods under test for all test arguments. In general, existing applications often include manually written CUTs. With the existence of these CUTs, natural questions that arise are whether these CUTs can be retrofitted as PUTs to leverage the benefits of PUTs, and what are the cost and benefits involved in retrofitting CUTs as PUTs. To address these questions, in this paper, we conduct an empirical study to investigate whether existing CUTs can be retrofitted as PUTs with feasible effort and achieve the benefits of PUTs in terms of additional fault-detection capability and code coverage. We also propose a methodology, called *test generalization*, that helps in systematically retrofitting existing CUTs as PUTs. Our results on three real-world open-source applications ($\approx$ 4.6 KLOC) show that the retrofitted PUTs detect 19 new defects that are not detected by existing CUTs, and also increase branch coverage by 4% on average (with maximum increase of 52% for one class under test and 10% for one application under analysis) with feasible effort.[1]

1 Introduction

Unit tests are widely adopted in software industry for ensuring high quality of production code. Unit testing helps detect defects at an early stage, reducing the effort required in fixing those defects. Recent advances in unit testing introduced parameterized unit tests (PUT) [23], which accept parameters, unlike conventional unit tests (CUT), which do not accept parameters. Existing state-of-the-art test-generation approaches such as Dynamic Symbolic Execution (DSE) [15, 10, 20, 22] can be used in combination with PUTs to automatically generate CUTs by instantiating the parameters. In particular, DSE systematically explores the code under test exercised by a PUT and generates CUTs that achieve high structural coverage such as branch coverage of the code under test. Section 2 presents more details on how CUTs can be generated from PUTs via DSE.

In general, PUTs are more beneficial than CUTs. The primary reason is that PUTs help describe the behaviors of methods under test for all test arguments. With PUTs, test data can be automatically generated using DSE-based approaches, thereby helping

[1] The first and second authors have made equal contributions.

D. Giannakopoulou and F. Orejas (Eds.): FASE 2011, LNCS 6603, pp. 294–309, 2011.

address the following two issues with CUTs. First, developers may not be able to write test data (in CUTs) 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 [24]. These redundant unit tests increase only the testing time and do not increase the fault-detection capability. Consider the three CUTs shown in Figure 1 for testing the `Push` method of an integer stack class `IntStack`. These three CUTs 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 (`Push`) that can be detected by passing a negative value as an 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 with respect to CUT1, since `IntStack` has the same behavior for all non-negative integers passed as arguments to `Push`. Since test data is automatically generated by DSE-based approaches that tend to exercise all feasible paths in the methods under test, the *fault-detection capability* of PUTs is often higher than that of CUTs. Furthermore, a single PUT can represent multiple CUTs, thereby reducing the *size of test code* and improve the maintainability of the 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.

In general, existing applications often include manually written CUTs [9]. With the existence of these CUTs, natural questions that arise are whether these CUTs can be retrofitted as PUTs to leverage the benefits of PUTs, and what are the cost and benefits involved in retrofitting CUTs as PUTs. Here, cost includes the effort required in retrofitting CUTs as PUTs, and benefits include the additional fault-detection capability and code coverage achieved via retrofitting. However, to the best of our knowledge, there exists no empirical study that shows cost and benefits involved in retrofitting existing CUTs as PUTs. To address this issue, in this paper, we conduct an empirical study to investigate whether existing CUTs can be retrofitted as PUTs with feasible effort and such retrofitting achieves benefits in terms of fault-detection capability and code coverage. We also propose a methodology, called *test generalization*, that includes a systematic procedure for manually retrofitting CUTs as PUTs.

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

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

```
01:public void PUT(int[] elem) {
02:  IntStack stk = new IntStack();
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

In particular, our empirical study helps address the following two fundamental questions. First, is it cost-effective to retrofit existing CUTs as PUTs (using test generalization) with regards to the benefits of test generalization? Second, can developers other than the original developers who wrote the code under test (who do not have sufficient knowledge of the code under test) use our methodology to retrofit CUTs as PUTs? Such other developers could be those who take over legacy applications or those who try to augment the test suites for the code not written by them. The primary reason for investigating the second question is that, in general, developers who wrote code under test may not face challenges in writing test oracles (in PUTs) that need to describe the expected behavior for all test arguments; however, these other developers often do not have sufficient knowledge of code under test and may face challenges in writing test oracles in PUTs. Therefore, in this paper, we study whether our test-generalization methodology could help these other developers in addressing the challenge of test-oracle generalization (generalizing test oracles in existing CUTs). In particular, to address this issue, the first and second authors (of this paper) who do not have sufficient knowledge of our applications under analysis follow our methodology to retrofit existing CUTs as PUTs. Our results show that test generalization helps these other developers in achieving additional benefits in terms of fault-detection capability and code coverage with feasible effort.

In summary, this paper makes the following major contributions:

- The first empirical study that investigates cost and benefits involved in retrofitting existing CUTs as PUTs for leveraging the benefits of PUTs.
- A methodology, called *test generalization*, that helps developers to write PUTs with feasible effort by leveraging existing CUTs.
- Our empirical results on three real-world applications ($\approx$ 4.6 KLOC) show that test generalization helps detect 19 new defects that are not detected by existing CUTs, showing the benefits in terms of fault-detection capability with feasible effort. A few of these defects are complex to be detected using manually written CUTs. Furthermore, test generalization increases branch coverage by 4% on average (with a maximum increase of 52% for one class under test and 10% for one application under analysis).

2 Background

We use Pex [4] as an example state-of-the-art DSE-based test generation tool for generating CUTs using PUTs. Pex 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 code coverage of the code under test. Since these generated CUTs are targeted for some common testing frameworks such as NUnit [7], it is possible to debug and analyze failing CUTs. 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 applied on industrial code bases and detected serious new defects in a software component, which had already been extensively tested previously [22].

3 Test Generalization Methodology

We next present our test generalization methodology that assists developers in achieving test generalization. Although we explain our methodology using Pex, our methodology is independent of Pex and can be used with other DSE-based test generation tools [20]. Our methodology 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 more 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 a 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 in the PUTs. 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.

Algorithm 1 Test Generalization

```
Require: CUTs for an MT M
Ensure: PUTs
Set PUTs = ϕ, gAllCUTs = ϕ
for all c ∈ CUTs do
    if gAllCUTs.Contains(c) then
        Continue
    end if
    Set p = ϕ, gCUTs = ϕ, break = false
    p = Parameterize(c)
    p = GeneralizeTestOracle(c, p)
    gCUTs = GenerateCUTs(p)
    repeat
        while !Execute(gCUTs) do
            if LegalValueIssue(gCUTs) then
                p = AddAssumption(p)
            else
                ReportDefect()
                Continue
            end if
        end while
        if !Cov(M, gCUTs) ⊇ Cov(M, c) then
            if NPTypeParam(p) then
                p = AddFactoryMethod(p)
            end if
            if EnviInteractionIssue(M) then
                p = AddMockObj(p)
            end if
        else
            break = true
        end if
    until break
    PUTs.Add(p), gAllCUTs.Add(gCUTs)
end for
return PUTs
```

We next provide an overview of how a developer generalizes existing CUTs to PUTs by using our methodology to satisfy the preceding requirements and then explain each step in detail using illustrative examples from the NUnit framework [7].

3.1 Overview

Our test generalization algorithm includes five major steps: (S1) *Parameterize*, (S2) *Generalize Test Oracle*, (S3) *Add Assumption*, (S4) *Add Factory Method*, and (S5) *Add*

Mock Object. In our methodology, Steps S1 and S2 are mandatory, whereas Steps S3, S4, and S5 are optional and are used when R1 or R2 is not satisfied. Indeed, recent work [21, 16] (as discussed in subsequent sections) could help further alleviate effort required in Steps S3, S4, and S5. We next explain our methodology in detail.

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 (referred to as *gCUTS*) 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 [21], and (2) the MT includes interactions with external environments [16]. 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 [16] (Line 24), respectively, to assist Pex. More details on these two steps are described in subsequent sections.

```
//MSS=MemorySettingsStorage
00:public class SettingsGroup{
01: MSS storage; ...
02: public SettingsGroup(MSS storage){
03:  this.storage = storage; }
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 NUnit with the `SaveSetting` method under test

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

Fig. 4. A CUT to test the `SaveSetting` method

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 an existing CUT that is already generated by a previously

generalized PUT, the developer checks whether the existing CUT to be generalized belongs to already generated CUTs (referred to as *gAllCUTs*) (Lines $3-5$). If so, the developer ignores the existing CUT; otherwise, the developer generalizes the existing CUT. We next illustrate each step of our methodology using an MT and a CUT from the NUnit framework shown in Figures 3 and 4, respectively.

3.2 Example

MT 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 for the setting values of only 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[2] 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.

S1 - Parameterize. For the CUT shown in Figure 4, the developer promotes the `string` "`Tom`" and the `int 5` as 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 (`tg`) 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.

S2 - Generalize Test Oracle. 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

[2] We use NCover (`http://www.ncover.com/`) to measure branch coverage. NCover uses .NET byte code instead of source code for measuring branch coverage.

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 a 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 [6].

S3 - Add Assumption. A challenge faced during test generalization is that Pex or any DSE-based approach 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 set 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 each parameter with the tag `PexAssumeUnderTest`[3], 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. Recently, there is a growing interest towards a new methodology, called *Code Contracts* [5], where developers can explicitly describe assumptions of the code under test. We expect that effort required for Step S3 can be further reduced when the code under test includes contracts.

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 [21]. 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`. Recent techniques in object-oriented testing [21, 13] could

```
//PAUT: PexAssumeUnderTest
00:[PexMethod]
01:public void TestSave([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;}
```

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

[3] `PexAssumeUnderTest` is a custom attribute provided by Pex, shown as "PAUT" for simplicity in Figures 5, 6, and 9.

Table 1.

(a) Names. (b) Characteristics of subject applications. (c) Existing CUTs.

Subject Applications	Downloads	Code Under Test					Existing Test Code		
		#C	#M	#KLOC	Avg.CC	Max.CC	#C	#CUTs	#KLOC
NUnit	193,563	9	87	1.4	1.48	14.0	9	49	0.9
DSA	3239	27	259	2.4	2.09	16.0	20	337	2.5
QuickGraph	7969	56	463	6.2	1.79	16.0	9	21	1.2

help reduce effort required for this step. However, since Pex does not include these techniques yet, the developer can assist Pex by writing method-call sequences inside factory methods, supported by Pex. Figure 6 shows an example factory method for the `MemorySettingsStorage` class.

S5 - Add Mock Object. Pex (or any other existing DSE-based approaches) also faces challenges in handling PUTs or MT that interacts with an external environment such as a file system. To address this challenge related to the interactions with the environment, developers write mock objects for assisting Pex [16]. These mock objects help test features in isolation especially when PUTs or MT interact with environments such as a file system. Recent work [16] on mock objects can further help reduce effort in writing mock objects.

Generalized PUT. Figure 5 shows the final PUT after the developer follows our methodology. 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 two constant setting values are 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 additional advantage of test generalization in reducing the test code substantially 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`, `string`, or other non-primitive object types. Furthermore, the CUT used for generalization achieved branch coverage of 10%, whereas the CUTs generated from the generalized PUT achieved branch coverage of 90%, showing the benefits achieved through our test generalization methodology.

4 Empirical Study

We conducted an empirical study using three real-world applications to show the benefits of retrofitting CUTs as PUTs. In our empirical study, we show the cost and benefits

of PUTs over existing CUTs using *three metrics*: branch coverage, the number of detected defects, and the time taken for test generalization. In particular, we address the following three research questions in our empirical study:

- **RQ1: Branch Coverage.** How much higher percentage of *branch coverage* is achieved by retrofitted PUTs compared to existing CUTs? Since PUTs are a generalized form of CUTs, this research question helps 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 checking branches for those test assertions.
- **RQ2: Defect Detection.** How many new *defects* (that are not detected by existing CUTs) are detected by PUTs and vice-versa? This research question helps address whether PUTs have higher fault-detection capabilities compared to CUTs.
- **RQ3: Generalization Effort.** How much effort is required for generalizing CUTs to PUTs? This research question helps 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 our 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`.

4.1 Subject Applications

We use three popular open source applications (as shown by their download counts in their hosting web sites) in our study: NUnit [7], DSA [11], and Quickgraph [12]. Table 1(a) shows the names of three subject applications. 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 1(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 in January 2011). Column "Code Under Test" shows details of the code under test (of the application) in terms of the number of classes ("#C"), number of methods ("#M"), number of lines of code ("#KLOC"), and the average and maximum cyclomatic complexity ("Avg.CC" and "Max.CC", respectively) of the code under test. Similarly, Table 1(c) shows the statistics of existing CUTs for these subject applications.

4.2 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 were PhD (fourth year) and master (second year) students, respectively, with the same experience of two years with PUTs and Pex at the time of conducting the study. Before joining their graduate program, the authors had three and five years of programming experience, respectively, in software industry. Each of the authors conducted test generalization for half of CUTs across all the three subjects. The authors do not have prior knowledge of the

Table 2. Branch coverage achieved by the existing CUTs, CUTs + RTs, and CUTs generated by Pex using the retrofitted PUTs

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%

subject applications and conducted the study as third-party testers. We expect that our test-generalization results could 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 application 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 the 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. Among all three subject applications, the authors retrofitted 407 CUTs (4.6 KLOC) as 224 PUTs (4.0 KLOC). The authors also measured the time taken for generalizing all CUTs to compute the generalization effort for addressing RQ3. The measured time includes the amount of time taken for performing all steps described in our methodology and also applying Pex to generate CUTs from PUTs. The authors wrote 10 factory methods and 1 mock object during test generalization. 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 [18]. The authors used the default timeout parameter of 180 seconds. The rationale behind using the default timeout is that running Randoop for longer time often generates a large number of tests that are difficult to be compiled. 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[4]. To address RQ2 and RQ3, the authors measured the number of failing tests and computed the code metrics (LOC) using the CLOC[5] 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).

4.3 RQ1: Branch Coverage

We next describe our empirical results for addressing RQ1. Table 2 shows the branch coverage achieved by executing the existing CUTs, CUTs + RTs, and the CUTs generated by Pex using the retrofitted PUTs. The values in brackets (#) for CUTs + RTs

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

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

indicate the number of RTs, i.e., the tests generated by Randoop. Column "Overall Inc." shows the overall increase in the branch coverage from the existing CUTs to the retrofitted 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 only a little higher coverage than existing CUTs.

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

Fig. 7. `RemoveSetting` method whose coverage is increased by 60% due to test generalization

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).

4.4 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[6]. On the other hand, RTs

```
//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));}
```

Fig. 8. Existing CUT to test the `Remove` method of `Heap`

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

Fig. 9. A generalized PUT of the CUT shown in Fig. 8

[6] 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.

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 the 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 methodology, 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 are passed as arguments to methods invoked in RTs. In our methodology, we address this issue of illegal test data by adding assumptions to PUTs in Step S1. This issue of illegal test data in RTs shows the significance of Step S1 in our methodology.

To further show the significance of generalized PUTs, we applied Pex on these applications without using these PUTs and by 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 under test. 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 faced by 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 8 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 existing test suite) that exercise the behavior of the method. However, from our generalized PUT shown in Figure 9, 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 the 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 8 since the unit test assigns the heap with 5 elements (Statement 2) and therefore the defect-exposing scenario of heap size ≤ 4 is not 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. Furthermore, as described earlier, it is also difficult to write new CUTs (manually) that test corner cases as exercised by CUTs generated from PUTs.

4.5 RQ3: Generalization Effort

We next address RQ3 regarding the manual effort required for the generalization of CUTs to PUTs. The first two authors conducted comparable amount of generalization by equally splitting the existing CUTs of all three subject applications for generalization. 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: the expertise with PUTs and the Pex tool, prior knowledge of the subject applications, number of CUTs and the number of generalized PUTs, and the complexity of a CUT or a generalized PUT. Although the authors have experience with PUTs and using Pex, the authors do not have the prior knowledge of these subject applications and conducted the study as third-party testers. Therefore, we expect that the developers of these subject applications, despite 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.

5 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 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 and only two human subjects involved in the study. 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 additional human subjects. The results in our study can also vary based on other factors such as test-generation capability of Pex.

6 Related Work

Pex [22] accepts PUTs and uses dynamic symbolic execution to generate test inputs. Although we use the Pex terminology in describing our generalization procedure, our

procedure is independent of Pex and can be applied with other testing tools that accept unit tests with parameters such as JUnitFactory [1] for Java testing. Other existing tools such as Parasoft Jtest [2] and CodeProAnalytiX [3] adopt the design-by-contract approach [17] 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. [19] 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 [8, 18, 14] that 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 [18] 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 existing sequence-generation strategy faces challenges.

In our previous work [16], 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.

7 Conclusion

Recent advances in software testing introduced parameterized unit tests (PUTs) [23], which are a generalized form of conventional unit tests (CUTs). With PUTs, developers do not need to provide test data (for PUTs), which are generated automatically using state-of-the-art test-generation approaches such as dynamic symbolic execution. Since many existing applications often include manually written CUTs, in this paper, we present an empirical study to investigate whether existing CUTs can be retrofitted as PUTs to leverage the benefits of PUTs. We also proposed a methodology, called test generalization, for systematically retrofitting CUTs as PUTs. Our empirical results

show that test generalization helped detect 19 new defects and also helped achieve additional branch coverage of the code under test. In future work, we plan to automate our methodology to further reduce the manual effort required for test generalization. Furthermore, given the results of our current study, we plan to conduct further empirical study to compare the cost and benefits involved in writing PUTs directly, and writing CUTs first and generalizing those CUTs as PUTs using our methodology.

Acknowledgments

This work is supported in part by NSF grants CCF-0725190, CCF-0845272, CCF-0915400, CNS-0958235, ARO grant W911NF-08-1-0443, and ARO grant W911NF-08-1-0105 managed by NCSU SOSI.

References

1. Agitar JUnit Factory (2008), http://www.agitar.com/developers/junit_factory.html
2. Parasoft Jtest (2008), http://www.parasoft.com/jsp/products/home.jsp?product=Jtest
3. CodePro AnalytiX (2009), http://www.eclipse-plugins.info/eclipse/plugin_details.jsp?id=943
4. Pex - automated white box testing for .NET (2009), http://research.microsoft.com/Pex/
5. Code Contracts (2010), http://research.microsoft.com/en-us/projects/contracts/
6. Pex Documentation (2010), http://research.microsoft.com/Pex/documentation.aspx
7. Cansdale, J., Feldman, G., Poole, C., Two, M.: NUnit (2002), http://nunit.com/index.php
8. Csallner, C., Smaragdakis, Y.: JCrasher: an automatic robustness tester for Java. Softw. Pract. Exper. 34(11) (2004)
9. Daniel, B., Jagannath, V., Dig, D., Marinov, D.: ReAssert: Suggesting repairs for broken unit tests. In: Proc. ASE, pp. 433–444 (2009)
10. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed Automated Random Testing. In: Proc. PLDI, pp. 213–223 (2005)
11. Granville, B., Tongo, L.D.: Data structures and algorithms (2006), http://dsa.codeplex.com/
12. de Halleux, J.: Quickgraph, graph data structures and algorithms for .NET (2006), http://quickgraph.codeplex.com/
13. Jaygarl, H., Kim, S., Xie, T., Chang, C.K.: OCAT: Object capture-based automated testing. In: Proc. ISSTA, pp. 159–170 (2010)
14. Khurshid, S., Pasareanu, C.S., Visser, W.: Generalized symbolic execution for model checking and testing. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 553–568. Springer, Heidelberg (2003)
15. King, J.C.: Symbolic execution and program testing. Communications of the ACM 19(7), 385–394 (1976)

16. Marri, M.R., Xie, T., Tillmann, N., de Halleux, J., Schulte, W.: An empirical study of testing file-system-dependent software with mock objects. In: Proc. AST, Business and Industry Case Studies, pp. 149–153 (2009)
17. Meyer, B.: Object-Oriented Software Construction. Prentice Hall PTR, Englewood Cliffs (2000)
18. Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: Proc. ICSE. pp. 75–84 (2007)
19. Saff, D., Boshernitsan, M., Ernst, M.D.: Theories in practice: Easy-to-write specifications that catch bugs. Tech. Rep. MIT-CSAIL-TR-2008-002, MIT Computer Science and Artificial Intelligence Laboratory (2008), http://www.cs.washington.edu/homes/mernst/pubs/testing-theories-tr002-abstract.html
20. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: Proc. ESEC/FSE, pp. 263–272 (2005)
21. Thummalapenta, S., Xie, T., Tillmann, N., de Halleux, P., Schulte, W.: MSeqGen: Object-oriented unit-test generation via mining source code. In: Proc. ESEC/FSE, pp. 193–202 (2009)
22. Tillmann, N., de Halleux, J.: Pex–white box test generation for .NET. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 134–153. Springer, Heidelberg (2008)
23. Tillmann, N., Schulte, W.: Parameterized Unit Tests. In: Proc. ESEC/FSE, pp. 253–262 (2005)
24. Xie, T., Marinov, D., Notkin, D.: Rostra: A framework for detecting redundant object-oriented unit tests. In: Proc. ASE, pp. 196–205 (2004)
25. Xie, T., Tillmann, N., de Halleux, P., Schulte, W.: Mutation analysis of parameterized unit tests. In: Proc. Mutation, pp. 177–181 (2009)