A Characteristic Study of Parameterized Unit Tests in .NET Open Source Projects

Wing Lam
University of Illinois at Urbana-Champaign, USA
winglam2@illinois.edu

Siwakorn Srisakaokul
University of Illinois at Urbana-Champaign, USA
srisaka2@illinois.edu

Blake Bassett
University of Illinois at Urbana-Champaign, USA
rbasset2@illinois.edu

Peyman Mahdian
University of Illinois at Urbana-Champaign, USA
mahdian2@illinois.edu

Tao Xie
University of Illinois at Urbana-Champaign, USA
taoxie@illinois.edu

Pratap Lakshman
Microsoft, India
pratapl@microsoft.com

Jonathan de Halleux
Microsoft Research, USA
jhalleux@microsoft.com

Abstract

In the past decade, parameterized unit testing has emerged as a promising method to specify program behaviors under test in the form of unit tests. Developers can write parameterized unit tests (PUTs), unit-test methods with parameters, in contrast to conventional unit tests, without parameters. The use of PUTs can enable powerful test generation tools such as Pex to have strong test oracles to check against, beyond just uncaught runtime exceptions. In addition, PUTs have been popularly supported by various unit testing frameworks for .NET and the JUnit framework for Java. However, there exists no study to offer insights on how PUTs are written by developers in either proprietary or open source development practices, posing barriers for various stakeholders to bring PUTs to widely adopted practices in software industry. To fill this gap, we first present categorization results of the Microsoft MSDN Pex Forum posts (contributed primarily by industrial practitioners) related to PUTs. We then use the categorization results to guide the design of the first characteristic study of PUTs in .NET open source projects. We study hundreds of PUTs that open source developers wrote for these open source projects. Our study findings provide valuable insights for various stakeholders such as current or prospective PUT writers (e.g., developers), PUT framework designers, test-generation tool vendors, testing researchers, and testing educators.

2012 ACM Subject Classification Software and its engineering → Software testing and debugging

Keywords and phrases Parameterized unit testing, automated test generation, unit testing

32nd European Conference on Object-Oriented Programming (ECOOP 2018).
Editor: Todd Millstein; Article No. 5; pp. 5:1–5:28
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.5

Acknowledgements This work was supported in part by National Science Foundation under grants no. CCF-1409423, CNS-1513939, and CNS1564274.

1 Introduction

With advances in test generation research such as dynamic symbolic execution [23, 35], powerful test generation tools are now at the fingertips of software developers. For example, Pex [37, 39], a state-of-the-art tool based on dynamic symbolic execution, has been shipped as *IntelliTest* [32, 26] in Microsoft Visual Studio 2015 and 2017, benefiting numerous developers in software industry. Such test generation tools allow developers to automatically generate input values for the code under test, comprehensively covering various program behaviors and consequently achieving high code coverage. These tools help alleviate the burden of extensive manual software testing, especially on test generation.

Although such tools provide powerful support for automatic test generation, when they are applied directly to the code under test, only a predefined limited set of properties can be checked. These predefined properties serve as test oracles for these automatically generated input values, and violating these predefined properties leads to various runtime exceptions, such as null dereferencing or division by zero. Despite being valuable, these predefined properties are *weak test oracles*, which do not aim for checking functional correctness but focus on robustness of the code under test.

To supply strong test oracles for automatically generated input values, developers can write formal specifications such as code contracts [25, 30, 16] in the form of preconditions, postconditions, and object invariants in the code under test. However, just like writing other types of formal specifications, writing code contracts, especially postconditions, can be challenging. According to a study on code contracts [34], 68% of code contracts are preconditions while only 26% of them are postconditions (the remaining 6% are object invariants). Section 2 shows an example of a method under test whose postconditions are difficult to write.

In the past decade, parameterized unit testing [40, 38] has emerged as a practical alternative to specify program behaviors under test in the form of unit tests. Developers can write parameterized unit tests (PUTs), unit-test methods with parameters, in contrast to conventional unit tests (CUTs), without parameters. Then developers can apply an automatic test generation tool such as Pex to automatically generate input values for a PUT's parameters. Note that algebraic specifications [24] can be naturally written in the form of PUTs but PUTs are not limited to being used to specify algebraic specifications.

Since parameterized unit testing was first proposed in 2005 [40], PUTs have been popularly supported by various unit testing frameworks for .NET along with recent versions of the JUnit framework (as parameterized tests [9] and theories [33, 13]). However, there exists no study to offer insights on how PUTs are written by developers in development practices of either proprietary or open source software, posing barriers for various stakeholders to bring PUTs to widely adopted practices in software industry. Example stakeholders are current or prospective PUT writers (e.g., developers), PUT framework designers, test-generation tool vendors, testing researchers, and testing educators.

To address the lack of studies on PUTs, we first conduct a categorization of 93 Microsoft MSDN Pex Forum posts [31] (contributed primarily by industrial practitioners) related to parameterized unit tests. We then use the categorization results to guide the design of the first characteristic study of PUTs in .NET open source projects (with a focus on PUTs written using the Pex framework, given that Pex is one of the most widely used test generation tools in industry [39]). Our findings from the categorization results of the forum posts show the following top three PUT-related categories that developers are most concerned with:

1. "Assumption/Assertion/Attribute usage" problems, which involve the discussion of using certain PUT assumptions, assertions, and attributes to address the issues faced by developers, are the most popular category of posts (occupying 23 of the 93 posts).
2. "Non-primitive parameters/object creation" problems, which involve the discussion of generating objects for PUTs with parameters of a non-primitive type, are the second most popular category of posts (occupying 17 of the 93 posts).
3. "PUT concept/guideline" problems, which involve the discussion of the PUT concept and general guidelines for writing good PUTs, are the third most popular category of posts (occupying 11 of the 93 posts).

Upon further investigation into these top PUT-related categories, we find that developers in general are concerned with when and what assumptions, assertions, and attributes they should use when they are writing PUTs. We find that a significant number of forum posts are directly related to how developers should replace hard-coded method sequences with non-primitive parameters of PUTs. We also find that developers often question what patterns their PUTs should be written in. Using our categorization and investigation results, we formulate three research questions and answer these questions using 11 open-source projects, which contain 741 PUTs.

In particular, we investigate the following three research questions and attain corresponding findings:

1. **What are the extents and the types of assumptions, assertions, and attributes being used in PUTs?** We present a wide range of assumption, assertion, and attribute types used by developers as shown in Tables 3a, 3b, and 5, and tool vendors or researchers can incorporate this data with their tools to better infer assumptions, assertions, and attributes to assist developers. For example, tool vendors or researchers who care about the most commonly used assumptions should focus on `PexAssumeUnderTest` or `PexAssumeNotNull`, since these two are the most commonly used assumptions. Lastly, based on the studied PUTs, we find that increasing the default value of attributes as suggested by tools such as Pex rarely contributes to increased code coverage. Tool vendors or researchers should aim to improve the quality of the attribute recommendations provided by their tools, if any are provided at all.
2. **How often can hard-coded method sequences in PUTs be replaced with non-primitive parameters and how useful is it to do so?** There are a significant number of receiver objects in the PUTs (written by developers) that could be promoted to non-primitive parameters, and a significant number of existing non-primitive parameters that lack factory methods (i.e., methods manually written to help the tools generate desirable object states for non-primitive parameters). It is worthwhile for tool researchers or vendors to provide effective tool support to assist developers to promote these receiver objects (resulted from hard-coded method sequences), e.g., inferring assumptions for a non-primitive parameter promoted from hard-coded method sequences. Additionally, once hard-coded method sequences are promoted to non-primitive parameters, developers can also use assistance in writing more factory methods for such parameters.
3. **What are common design patterns and bad code smells of PUTs?** By understanding how developers write PUTs, testing educators can teach developers appropriate

ways to improve PUTs. For example, developers should consider splitting PUTs with multiple conditional statements into separate PUTs each covering a case of the conditional statements. Doing so makes the PUTs easier to understand and eases the effort to diagnose the reason for test failures. Tool vendors and researchers can also incorporate this data with their tools to check the style of PUTs for suggesting how the PUTs can be improved. For example, checking whether a PUT contains conditionals, contains hard-coded test data, and contains duplicate test code, etc. often accurately identifies a PUT that can be improved.

In summary, this paper makes the following major contributions:

- The categorization of the Microsoft MSDN Pex Forum posts (contributed primarily by industrial practitioners) related to PUTs.
- The first characteristic study of PUTs in open source projects, with a focus on hundreds of real-world PUTs, producing study findings that provide valuable insights for various stakeholders.
- A collection of real-world open-source projects equipped with developer-written PUTs and a suite of tools for analyzing PUTs (both are used for our study and are released on our project website [10]). These PUTs and analysis tools can be used by the community to conduct future empirical studies or to evaluate enhancements to automated test generation tools.

The work in this paper is part of the efforts of our industry-academia team (including university/industrial testing researchers and tool vendors) for bringing parameterized unit testing to broad industrial practices of software development. To understand how automatic test generation tools interact with PUTs, we specifically study PUTs written with the Pex framework. Besides the Pex framework, other .NET frameworks such as NUnit also support PUTs. In recent years, PUTs are also increasingly adopted among Java developers, partly due to the inclusion of parameterized test [9] and theories [33, 13] in the JUnit framework. However, unlike the Pex framework, these other frameworks lack powerful test generation tools such as Pex to support automatic generation of tests with high code coverage, and part of our study with PUTs, specifically the part described in Section 5, does investigate the code coverage of the input values automatically generated from PUTs.

The remainder of this paper is organized as follows. Section 2 presents an example of parameterized unit testing. Section 3 discusses the categorization of Pex forum posts that motivates our study. Section 4 discusses the setup of our study. Section 5 presents our study findings and discusses the implications to stakeholders. Section 6 discusses threats to validity of our study. Section 7 presents our related work, and Section 8 concludes the paper.

2 Background

Consider the method under test from the open source project of NUnit Console [6] in Figure 1. One way to supply strong test oracles for automatically generated input values is to write preconditions and postconditions for this method under test. It is relatively easy to specify preconditions for the method as `(sn != null) && (sv != null)` but it is actually quite challenging to specify comprehensive postconditions to capture this method's intended behaviors. The reason is that this method's intended behaviors depend on the behaviors of all the method calls inside the `SaveSetting` method. In order to write postconditions for `SaveSetting`, we would need to know the postconditions of the other method calls in

```
1  public class SettingsGroup {
2    private Hashtable storage = new Hashtable();
3    public event SettingsEventHandler Changed;
4    public void SaveSetting(string sn, object sv) {
5      object ov = GetSetting(settingName);
6      //Avoid change if there is no real change
7      if(ov != null) {
8        if((ov is string && sv is string && (string)ov == (string)sv) ||
9           (os is int && sv is int && (int)ov == (int)sv) ||
10          (os is bool && sv is bool && (bool)ov == (bool)sv) ||
11          (os is Enum && sv is Enum && ov.Equals(sv)))
12         return;
13     }
14     storage[settingName] = settingValue;
15     if(Changed != null)
16       Changed(this, new SettingsEventArgs(sn));
17   }
18 }
```

Figure 1 `SaveSetting` method under test from the `SettingsGroup` class of NUnit Console [6].

`SaveSetting` (e.g., `GetSetting`) as well. In addition, the postconditions can be very long since there are many conditional statements with complex conditions (e.g., Lines 8-11). If a method contains loops, its postcondition may be even more difficult to write, since we would need to know the loop invariants and the postconditions may need to contain quantifiers. Thus, there is a need for a practical method to specify program behaviors under test in the form of unit tests. Specifying program behaviors in the form of unit tests can be easier since we do not need to specify all the intended behaviors of the method under test as a single logical formula. Instead, we can write test code to specify the intended behaviors of the method under test for a specific scenario (e.g., interacting with other specific methods). For example, a real-world conventional unit test (CUT) written by the NUnit developers is shown in Figure 2. The CUT in this figure checks that after we save a setting by calling the `SaveSetting` method, we should be able to retrieve the same setting by calling the `GetSetting` method. Despite seemingly comprehensive, the CUT in Figure 2 is insufficient, since it is unable to cover Lines 8-12 of the method in Figure 1. Figure 3 shows an additional CUT that developers can write to cover Lines 8-12; this additional CUT checks that saving the same setting twice does not invoke the `Changed` event handler twice. These two CUTs' corresponding, and more powerful, PUT is shown in Figure 4.

The beginning of the PUT (Lines 3-5) include `PexAssume` statements that serve as assumptions for the three PUT parameters. During test generation, Pex filters out all the generated input values (for the PUT parameters) that violate the specified assumptions. These assumptions are needed to specify the state of `SettingsGroup` that one may want to test. For example, according to Lines 2-3 in Figure 2, `sg` initially does not have `"X"` and `"NAME"` set. Thus, we need to add `PexAssume.IsNull(st.Getting(sn))` (Line 5) to force Pex to generate only an object of `SettingsGroup` that satisfies the same condition as Lines 2-3 in Figure 2. Otherwise, without such assumptions, the input values generated by Pex may largely be of no interest to the developers. The `PexAssert` statements in Lines 7 and 10 are used as the assertions to be verified when running the generated input values. More specifically, the assumption on Line 5 and the assertion on Line 7 in the PUT correspond to Lines 2-3 and Lines 6-7, respectively, in the CUT from Figure 2. Lines 8-9 in the PUT then cover the case of calling the `SaveSetting` method twice with the same parameters as accomplished in the CUT shown in Figure 3. Note that writing the PUT allows the test to be more general as variable `sn` can be any arbitrary string, better than hard-coding it to be only `"X"` or `"NAME"` (as done in the CUTs).

A PUT is annotated with the `[PexMethod]` attribute, and is sometimes attached with

```
public void SaveAndLoadSettings() {
  Assert.IsNull(sg.GetSetting("X"));
  Assert.IsNull(sg.GetSetting("NAME"));
  sg.SaveSetting("X", 5);
  sg.SaveSetting("NAME", "Charlie");
  Assert.AreEqual(5, sg.GetSetting("X"));
  Assert.AreEqual("Charlie", sg.GetSetting("NAME"));
}
```

Figure 2 A real-world CUT for the method in Figure 1.

```
public void SaveSettingsWhenSettingIsAlreadyInitialized() {
  Assert.IsNull(sg.GetSetting("X"));
  sg.SaveSetting("X", 5);
  sg.SaveSetting("X", 5);
  // Below assert that Changed only got invoked once in SaveSetting
  ...
}
```

Figure 3 An additional CUT for the method in Figure 1 to cover the lines that the CUT in Figure 2 does not cover.

```
[PexMethod(MaxRuns = 200)]
public void TestSave1(SettingsGroup sg, string sn, object sv) {
  PexAssume.IsTrue(sg != null && sg.Changed != null);
  PexAssume.IsTrue(sn != null && sv != null);
  PexAssume.IsNull(sg.GetSetting(sn));
  sg.SaveSetting(sn, sv);
  PexAssert.AreEqual(sv, sg.GetSetting(sn));
  sg.SaveSetting(sn, sv);
  // Below assert that Changed only got invoked once in SaveSetting
  ...
}
```

Figure 4 The PUT corresponding to the CUTs in Figures 2 and 3.

optional attributes to provide configuration options for automatic test generation tools. An example attribute is `[PexMethod(MaxRuns = 200)]` as shown in Figure 4. The `MaxRuns` attribute along with the attribute value of `200` indicates that Pex can take a maximum of 200 runs/iterations during Pex's path exploration phase for test generation. Since the default value of `MaxRuns` is 1000, setting the value of `MaxRuns` to be just 200 decreases the time that Pex may take to generate input values. Note that doing so may also cause Pex to generate fewer input values.

3 Categorization of Forum Posts

This section presents our categorization results of the Microsoft MSDN Pex Forum posts [31] related to parameterized unit tests. As of January 10th, 2018, the forum includes 1,436 posts asked by Pex users around the world. These users are primarily industrial practitioners. To select the forum posts related to parameterized unit tests, we search the forum with each of the keywords "parameterized", "PUT", and "unit test". Searching the forum with these three keywords returns 14, 18, and 243 posts, respectively. We manually inspect each of these returned posts to select only posts that are actually related to parameterized unit tests. Finally among the returned posts, we identify 93 posts as those related to parameterized unit tests. Then we categorize these 93 posts into 8 major categories and one miscellaneous category, as shown in Table 1. The categorization details of the 93 posts can be found on our project website [10]. We next describe each of these categories and the number of posts falling into each category.

The posts falling into the top 1 category "assumption/assertion/attribute usage" (25% of the posts) involve discussion of using certain PUT assumptions, assertions, and attributes

Table 1 Categorization results of the Microsoft MSDN Pex Forum posts related to parameterized unit tests.

Category	**#Posts**
Assumption/Assertion/Attribute usage	25% (23/93)
Non-primitive parameters/object creation	18% (17/93)
PUT concept/guideline	12% (11/93)
Test generation	11% (10/93)
PUT/CUT relationship	9% (8/93)
Testing interface/generic class/abstract class	6% (6/93)
Code contracts	5% (5/93)
Mocking	5% (5/93)
Miscellaneous	9% (8/93)
Total	**100% (93/93)**

to address the issues faced by PUT users. The posts falling into the second most popular category "non-primitive parameters/object creation" (18% of the posts) involve discussion of generating objects for PUTs with non-primitive-type parameters, one of the two major issues [42] for Pex to generate input values for PUTs. The posts falling into category "PUT concept/guideline" (12% of the posts) involve discussion of the PUT concept and general guideline for writing good PUTs. The posts falling into category "test generation" (11% of the posts) involve discussion of Pex's test generation for PUTs. The posts falling into category "PUT/CUT relationship" (9% of the posts) involve discussion of co-existence of both CUTs and PUTs for the code under test. The posts falling into category "testing interface/generic class/abstract class" (6% of the posts) involve discussion of writing PUTs for interfaces, generic classes, or abstract classes. The posts falling into category "code contracts" (5% of the posts) involve discussion of writing PUTs for code under test equipped with code contracts [25, 30, 16]. The posts falling into category "mocking" (5% of the posts) involve discussion of writing mock models together with PUTs. The miscellaneous category (9% of the posts) includes those other posts that cannot be classified into one of the 8 major categories.

We use the posts from the top 3 major categories to guide our study design described in the rest of the paper, specifically with research questions RQ1-RQ3 listed in Section 5. In particular, our study focuses on quantitative aspects of assumption, assertion, and attribute usage (top 1 category) in RQ1, non-primitive parameters/object creation (top 2 category) in RQ2, and PUT concept/guideline (top 3 category) in RQ3.

4 Study Setup

This section describes our process for collecting subjects (e.g., open source projects containing PUTs) and the tools that we develop to collect and process data from the subjects. The details of these subjects and our tools can be found on our project website [10].

4.1 Subject-collection Procedure

The subject-collection procedure (including subject sanitization) is a multi-stage process. At a coarse granularity, this process involves (1) comprehensive and extensive subject collection from searchable online source code repositories, (2) deduplication of subjects obtained multiple times from different repositories, and (3) verification of developer-written parameterized unit tests (e.g., filtering out subjects containing only automatically-generated parameterized test stubs).

For comprehensive collection of subjects, we query a set of widely known code search services. The used query is "`PexMethod Assert`", requiring both "`PexMethod`" and "`Assert`" to appear in the source file of the search results. The two code search services that return non-empty results based on our search criteria are GitHub [4] and SearchCode [12]. For each code search service, we first search with our query, and then we extract the source code repositories containing the files in the search results. When a particular repository is available from multiple search services, we extract the version of the repository from the search service that has the most recent commit. Lastly, we manually verify that each of our source code repositories has at least one PUT with one or more parameters and one or more assertions.

4.2 Analysis Tools

We develop a set of tools to collect metrics from the subjects. We use Roslyn [5], the .NET Compiler Platform, to build our tools. These tools parse C# source files to produce an abstract syntax tree, which is traversed to collect information and statistics of interest. More specifically, the analysis tools statically analyze the C# source code in the .cs files of each subject. The outputs of the tools include but are not limited to the following: PUTs, PUTs with `if` statements, results in Tables 3 and 6, the number of assumption and assertion clauses, and attributes of the subjects' PUTs. In general, the results that we present in the remainder of the paper are collected either directly with the analysis tools released on our website [10], manual investigation conducted by the authors, or a combination of the two (e.g., using the PUTs with `if` statements to manually categorize the number of PUTs that have unnecessary `if` statements).

4.3 Collected Subjects

In total, we study 77 subjects and retain only the subjects that contain at least 10 PUTs and are not used for university courses or academic research (e.g., creating PUTs to experiment with Pex's capability of achieving high code coverage). This comprehensive list of subjects that we study can be found on our project website [10].

Table 2 shows the information on the subjects that contain at least 10 PUTs. We count a test method as a PUT if the test method is annotated with attribute "`PexMethod`" and has at least one parameter. Our detailed study for research questions focuses on subjects with at least 10 PUTs because a subject with fewer PUTs often includes occasional tryouts of PUTs instead of serious use of them for testing the functionalities of the open source project. Column 1 shows the name of each subject, and Columns 2-3 shows the number of PUTs and CUTs in each subject. Columns 4-6 show the number of the lines of production source code, PUTs and CUTs, respectively, in each subject. Columns 7-8 shows the percentage of statements covered in the project under test by the PUTs on which Pex is applied and by the CUTs of the subject. Column 9 shows the version of Pex a subject's PUTs were written with. If a subject contains PUTs written with multiple versions of Pex, the most recent version of Pex used to write the subject's PUTs is shown. Altogether, we identify 11 subjects with at least 10 PUTs, and these subjects contain a total of 741 PUTs. When we examine the profiles of the contributors to the subjects, we find that all but one of the subjects have contributors who work in industry. The remaining one subject, PurelyFunctionalDataStructures, referred to as PFDS in our tables, is developed by a graduate student imitating the algorithms in a

Table 2 Subjects collected for our study.

Subject Name	#Methods PUT	#Methods CUT	#LOC Source	#LOC PUT	#LOC CUT	Code Cov. PUT	Code Cov. CUT	Pex Version
Atom	240	297	127916	3570	3983	N/A	N/A	0.20.41218.2
BBCode	17	22	1576	188	219	84%	69%	0.94.0.0
ConcurrentList	23	57	315	243	645	51%	75%	0.94.0.0
Functional-dotnet	41	87	14002	355	1666	N/A	N/A	0.15.40714.1
Henoch	63	149	4793	142	2816	N/A	N/A	0.94.0.0
OpenMheg	45	6	21809	382	100	N/A	N/A	0.6.30728.0
PFDS	10	2	1818	120	34	50%	12%	0.93.0.0
QuickGraph	205	123	38530	1478	2186	5%	50%	0.94.0.0
SerialProtocol	34	0	7603	269	0	49%	0%	0.94.0.0
Shweet	12	42	2481	295	703	N/A	N/A	0.91.50418.0
Utilities-net	51	0	3224	475	0	26%	0%	0.94.0.0
Total	**741**	**785**	**223158**	**7496**	**12352**	-	-	-
Average	**67**	**71**	**22174**	**681**	**1123**	**44%**	**34%**	-

data structure textbook. The table shows the percentage of statements covered for only 5 out of 11 subjects because we have difficulties compiling the other subjects (e.g., a subject misses some dependencies). Part of our future work is to debug the remaining subjects so that we can compile them. More details about the subjects (e.g., the contributors of the subjects, the number of public methods in the subjects) can be found on our project website [10].

5 Study Results

Our study is based on forum posts asked by Pex users around the world as detailed in Sections 5.1 to 5.3. Our study findings aim to benefit various stakeholders such as current or prospective PUT writers (e.g., developers), PUT framework designers, test-generation tool vendors, testing researchers, and testing educators. In particular, our study intends to address the following three main research questions:

- **RQ1**: What are the extents and the types of assumptions, assertions, and attributes being used in PUTs?
  - We address RQ1 because addressing it can help understand developers' current practice of writing assumptions, assertions, and attributes in PUTs, and better inform stakeholders future directions on providing effective tool support or training on writing assumptions, assertions, and attributes in PUTs.
- **RQ2**: How often can hard-coded method sequences in PUTs be replaced with non-primitive parameters and how useful is it to do so?
  - We address RQ2 because addressing it can help understand the extent of writing sufficiently general PUTs (e.g., promoting an object produced by a method sequence hard-coded in a PUT to a non-primitive parameter of the PUT) to fully leverage automatic test generation tools.
- **RQ3**: What are common design patterns and bad code smells of PUTs?
  - We address RQ3 because addressing it can help understand how developers are currently writing PUTs and identify better ways to write good PUTs.

5.1 RQ1. Assumptions, Assertions, and Attributes

Table 3

(a) Different types of assumptions in subjects.

PexAssume Type	#	#NC
PexAssumeUnderTest	273	273
PexAssumeNotNull	211	211
IsTrue	158	2
AreNotEqual	73	0
EnumIsDefined	22	0
AreDistinct	13	0
AreDistinctValues	13	0
IsNotNull	10	10
IsFalse	9	0
AreEqual	9	0
TrueForAll	7	2
IsNotNullOrEmpty	4	4
Fail	4	0
InRange	3	0
AreElementsNotNull	1	1
Total	**810**	**503**
Null Check Percentage	**62% (503/810)**	

(b) Different types of assertions in subjects.

PexAssert Type	#	#NC
AreEqual	355	0
IsTrue	199	2
IsFalse	75	3
Inconclusive	43	0
IsNotNull	26	26
Equal	21	1
TrueForAll	19	0
That	17	0
AreElementsEqual	16	0
IsNull	9	9
AreNotEqual	5	0
Fail	5	0
Throws	5	0
AreBehaviorsEqual	4	0
ImpliesIsTrue	3	0
FALSE	3	0
TRUE	3	0
Empty	2	0
Implies	2	0
Contains	1	0
DoesNotContain	1	0
ReachEventually	1	0
Total	**815**	**41**
Null Check Percentage	**5% (41/815)**	

To understand developers' practices of writing assumptions, assertions, and attributes in PUTs, we study our subjects' common types of assumptions, assertions, and attributes. Our study helps provide relevant insights to the posts from the Assumption/Assertion/Attribute usage category described in Section 3. For example, the original poster of the forum post titled "New to Unit Testing" questions what type of assertions she/he should use. Another forum post titled "Do I use NUnit Assert or PexAssert inside my PUTs?" reveals that the original poster does not understand when and what assumptions to use.

5.1.1 Assumption Usage

As shown in Table 3a, `PexAssumeUnderTest` is the most common type of assumption, used 273 times in our subjects. `PexAssumeUnderTest` marks parameters as non-null and to be that precise type. The second most common type of assumption, `PexAssumeNotNull`, is used 211 times. Similar to `PexAssumeUnderTest`, `PexAssumeNotNull` marks parameters as non-null except that it does not require their types to be precise. Both `PexAssumeUnderTest` and `PexAssumeNotNull` are specified as attributes of parameters, but they are essentially a conve-

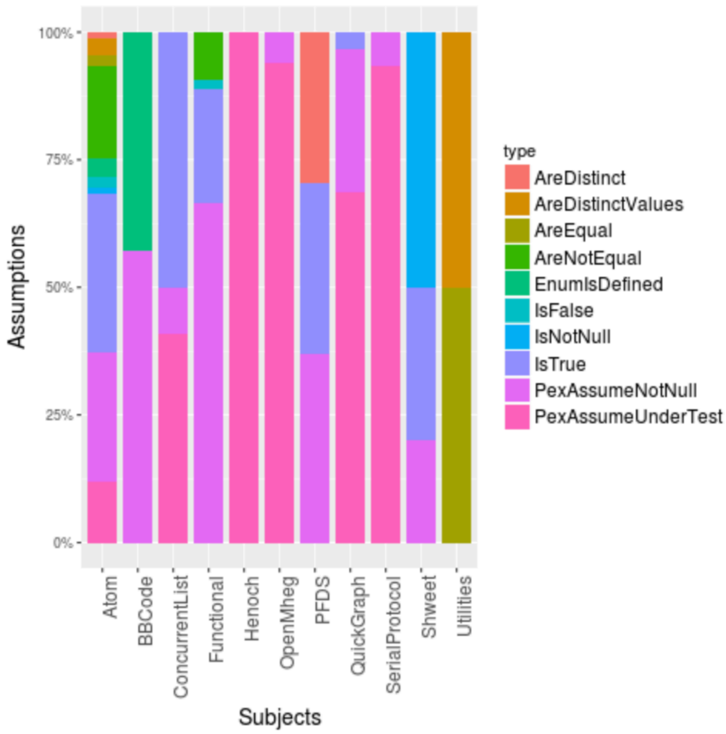

Figure 5 Assumption-type distribution for each of our subjects.

nient alternative to specifying assumptions (e.g., the use of attribute `PexAssumeNotNull` on a parameter `X` is the same as `PexAssume.IsNotNull(X)`). Since PUTs are commonly written to test the behavior of non-null objects as the class under test or use non-null objects as arguments to a method under test, it is reasonable that the common assumption types used by developers are ones that mark parameters as non-null. Figure 5 shows that the combination of `PexAssumeUnderTest`, `PexAssumeNotNull`, and `IsNotNull`, which are for nullness checking, appears the most in all of our subjects. Note that Figure 5 contains only the top 10 commonly used assumption types in our subjects. Furthermore, according to the last row of Tables 3a and 3b, developers perform null checks much more frequently for assumptions than assertions. Our findings about the frequency of assumption types and assertion types that check whether objects are null are similar to the findings of a previous study [34] on how frequently preconditions and postconditions in code contracts are used to check whether objects are null. Similar to code contracts, we find that 62% of assumptions perform null checks while the study on code contracts finds that 77% (1079/1356) of preconditions perform null checks. Our study also finds that 5% of assertions perform null checks while the study on code contracts finds that 43% (165/380) of postconditions perform null checks. Since assertions are validated at the end of a PUT and it is less often that code before the assertions manipulates or produces a null object, it is reasonable that assumptions check for null much more frequently than assertions do. For assumption and assertion types such as `TrueForAll`, developers' low number of uses may be due to the unawareness of such types' existence. `TrueForAll` checks whether a predicate holds over a collection of elements. In our subjects, we find cases such as the one in Figure 6 where a collection is iterated over to check whether a predicate is true for all of its elements; instead, developers could have used the

```
1  [PexMethod]
2  public void GetEnumerator_WhenMatrixConvertedToEnumerable_IteratesOverAllElements<T>(
3    [PexAssumeNotNull]ObjectMatrix<T> matrix ) {
4    System.Collections.IEnumerable enumerable = matrix;
5    foreach(var item in enumerable.Cast<T>())
6    {
7      Assert.IsTrue( matrix.Contains( item ) );
8    }
9  }
```

Figure 6 PUT (in Atom [1]) that could benefit from Pex's `TrueForAll` assertion.

`TrueForAll` assumption or assertion. More specifically, the developers of the method in Figure 6 could have replaced Lines 5-8 with `PexAssert.TrueForAll(enumerable.Cast<T>(), item => matrix.Contains(item))`. It is important to note that in versions of Pex after 0.94.0.0, certain assumption and assertion types were removed (e.g., `TrueForAll`). However, as shown in Table 2, none of our subjects used versions of Pex after 0.94.0.0.

5.1.2 Assertion Usage

According to Figure 7, in all of the subjects except OpenMheg, the PUTs usually contain assertions for nullness or equality checking. Instead, OpenMheg's assertions are mainly `Assert.Inconclusive`. `Assert.Inconclusive` is used to indicate that a test is still incomplete. From our inspection of the PUTs with `Assert.Inconclusive` in OpenMheg, we find that developers write `Assert.Inconclusive("this test has to be reviewed")` in the PUTs. When we investigate the contents of these PUTs, we find that the developers indeed use these assertions to keep track of which tests are still incomplete. One example of OpenMheg's PUT that contains `Assert.Inconclusive` is shown in Figure 8. The example is one of many PUTs in OpenMheg that create a new object but then do nothing with the object and contain no other assertions but `Assert.Inconclusive`. When we ignore all PUTs of OpenMheg that contain only `Assert.Inconclusive`, we find that the remaining assertions are similar to our other subjects in that most of them are for nullness or equality checking.

As shown in Table 4, the PFDS subject has the highest number of assume clauses per PUT method. Upon closer investigation of PFDS's assume clauses, we find that these clauses are necessary because PUTs in PFDS test various data structures and the developers of PFDS have to specify assumptions for all of its PUTs to guide Pex to generate data-structure inputs that are not null and contain some elements. When we examine the assume clauses in Atom, the subject with the second highest number of assume clauses per PUT method, we also find similar cases. On the other hand, the Shweet subject has the highest number of assert clauses per PUT method. Shweet's high number of assert clauses per PUT method can be attributed to the fact that the subject has multiple PUTs each of which contains around 8 assertions. The reason why some of Shweet's PUTs each have around 8 assertions is that the subject's PUTs test a web service, and the service returns 8 values every time it is triggered. Therefore, multiple of Shweet's PUTs assert for whether these 8 values are correctly returned or not.

5.1.3 Attribute Usage

To investigate developers' practices of configuring Pex via PUT attributes, we study the number and settings of attributes, as configuration options for running Pex, written by developers in PUTs. Our findings from the forum posts related to attributes suggest that developers are often confused on what attributes to use or how they should configure attributes. More specifically, 5 out of 23 of the Assumption/Assertion/Attribute usage forum

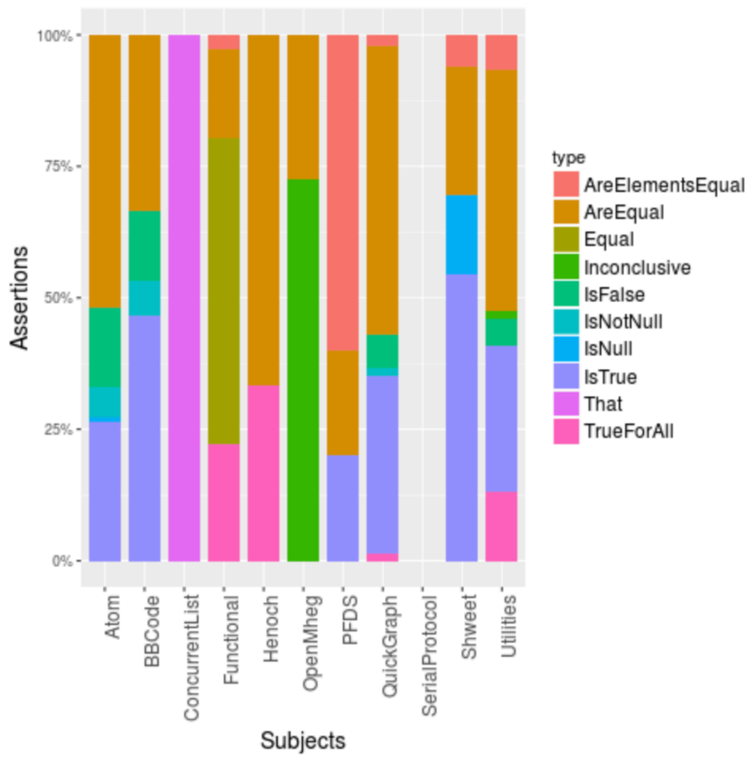

Figure 7 Assertion-type distribution for each of our subjects.

```
[PexMethod]
public Content Constructor03(GenericContentRef genericContentRef) {
  Content target = new Content(genericContentRef);
  Assert.Inconclusive("this test has to be reviewed");
  return target;
}
```

Figure 8 PUT (in OpenMheg [7]) that contains `Assert.Inconclusive`.

posts involve an answer recommending the use of a particular attribute or configuring an attribute in a specific way. For example, a post titled "the test state was: path bounds exceeded - infinite loop" discusses how developers should set the `MaxBranches` attribute of Pex. The setting of `MaxBranches` controls the maximum number of branches taken by Pex along a single execution path.

The fourth column of Table 4 shows the average number of attributes added per PUT. The results show that developers add only 1 attribute for every 3-4 PUTs. Table 5 shows the number of attributes added for our subjects. Common attributes that developers add are `MaxRuns`, `MaxConstraintSolverTime`, and `MaxBranches`. The setting of `MaxRuns` controls the maximum number of runs before Pex terminates. Developers commonly set this attribute to be 100 runs when the default value is 1,000. Upon our inspection, most of the PUTs that use this attribute test methods related to inserting objects into a data structure. By setting the value of this attribute, developers make Pex terminate faster. In fact, 14 out of 18 attributes used in QuickGraph are `MaxRuns`.

`MaxConstraintSolverTime` is another type of attribute that some projects contain. The attribute controls the constraint solver's timeout value during Pex's exploration. By default,

Table 4 Number of PexAssume clauses, PexAssert clauses, and Pex Attributes per PUT.

Subject Name	# of Assume Cl. / PUT		# of Assert Cl. / PUT		# of Attrs / PUT	
Atom	1.72	(412/240)	1.71	(411/240)	0.07	(16/240)
BBCode	1.71	(29/ 17)	1.47	(25/ 17)	2.18	(37/ 17)
ConcurrentList	0.96	(22/ 23)	0.74	(17/ 23)	0.26	(6/ 23)
Functional-dotnet	1.39	(57/ 41)	1.24	(51/ 41)	0.17	(7/ 41)
Henoch	0.78	(49/ 63)	0.05	(3/ 63)	0.38	(24/ 63)
OpenMheg	0.76	(34/ 45)	1.29	(58/ 45)	0.00	(0/ 45)
PFDS	2.70	(27/ 10)	1.10	(11/ 10)	0.00	(0/ 10)
QuickGraph	0.91	(186/205)	0.85	(175/205)	0.10	(21/205)
SerialProtocol	0.44	(15/ 34)	0.00	(0/ 34)	0.00	(0/ 34)
Shweet	1.00	(12/ 12)	3.42	(41/ 12)	0.33	(4/ 12)
Utilities-net	0.18	(9/ 51)	1.37	(70/ 51)	0.00	(0/ 51)
Average	**1.14**		**1.20**		**0.32**	

Table 5 Different types of Pex attributes in our subjects' PUTs.

Pex Attribute Type	#
MaxBranches	36
MaxRuns	18
MaxConstraintSolverTime	12
MaxConditions	8
MaxRunsWIthoutNewTests	6
MaxStack	5
Timeout	4
MaxExecutionTreeNodes	4
MaxWorkingSet	4
MaxConstraintSolverMemory	4
Total	**101**

`MaxConstraintSolverTime` is set to 10 seconds. Similar to `MaxRuns`, we find that developers often set the value to be lower than the default value so that Pex would finish sooner. For example, BBCode contains PUTs with `MaxConstraintSolverTime` set to 5 seconds, and Atom contains PUTs with `MaxConstraintSolverTime` set to 2 seconds.

In contrast to `MaxRuns`, we find that developers commonly set the value of `MaxBranches` to be higher than the default value. A common value set by developers is 20,000 when the default value is 10,000. When we study these PUTs, we find that the code tested by these PUTs all has loops, and the developers' intention when using this attribute is to increase the number of loop iterations allowed by Pex. For example, ConcurrentList contains several PUTs with `MaxBranches = 20000` set. When we run Pex without this attribute, Pex suggests to set `MaxBranches` to 20000. However, when we compare the code coverage with and without the attribute being set, we find that the code coverage does not increase with the attribute set. In fact, we find that when we manually unset all attributes of ConcurrentList, the code coverage does not change at all. The number of input values (generated by Pex) that exhibit a failed test result also does not change. Our findings indicate that increasing the default values of attributes often does not help increase the code coverage. In fact, for some of BB-

Code's PUTs, its developers set 9 different attributes all to the value of 1,000,000,000. Based on our estimation of running Pex on these PUTs, it would take approximately 2000 days for Pex to terminate. When we run Pex with a time limit of three hours on BBCode's PUTs with the developer-specified attributes, we notice that the coverage increases marginally by less than 1% compared to running Pex with the same time limit on BBCode's PUTs without any attributes.

5.1.4 Implications

With the wide range of assumption and assertion types used by developers as shown in Tables 3a and 3b, tool vendors or researchers can incorporate this data with their tools to better infer assumptions and assertions to assist developers. For example, tool vendors or researchers who care about the most commonly used assumption types should focus on `PexAssumeUnderTest` or `PexAssumeNotNull`, since these two are the most commonly used assumption types. Lastly, based on our subjects' PUTs, we find that increasing the default value of attributes as suggested by tools such as Pex rarely contributes to increased code coverage. Tool vendors or researchers should aim to improve the quality of the attribute recommendations provided by their tools, if any are provided at all.

5.2 RQ2. Non-primitive Parameters

Typically developers are expected to avoid hard-coding a method sequence in a PUT to produce an object used for testing the method under test. Instead, developers are expected to promote such objects to a non-primitive parameter of the PUT. In this way, the PUT can be made more general, to capture the intended behavior and enable an automatic test generation tool such as Pex to generate objects of various states for the non-primitive parameter. We find that 4 out of 17 answers from our non-primitive parameters/object creation category of forum posts described in Section 3 are directly related to how developers should replace hard-coded method sequences with non-primitive parameters. For example, in a forum post titled "Can Pex Generate a List<T> for my PUT", one of the answers to the question is that the developer should write a PUT that takes `List` as a non-primitive parameter instead of hard-coding a specific method sequence for producing a `List` object. Doing so enables Pex to generate non-empty, non-null objects of that list. Since many of our forum posts are related to how developers should replace hard-coded method sequences with non-primitive parameters, we decide to study how frequently developers write PUTs with non-primitive parameters and how often hard-coded method sequences in these PUTs could be replaced with non-primitive parameters. More details about the forum posts specifically related to this research question can be found on our project website [10].

5.2.1 Non-primitive Parameter Usage

As shown in Table 6, our result indicates that developers on average write non-primitive parameters 59.0% of the time for the PUTs in our subjects. In other words, for every 10 parameters used by developers, 5-6 of those parameters are non-primitive. However, developers write factory methods for only 17.9% of the non-primitive parameters used in our subjects' PUTs. The lack of non-primitive parameters and factory methods for such parameters inhibits test generation tools such as Pex from generating high-quality input values. For example, Figure 9 depicts 1 out of 16 PUTs that tests the `BinaryHeap` data structure in the QuickGraph subject. Promoting the object that it is testing (`BinaryHeap`)

Table 6 Statistics for factory methods and non-primitive parameters of our subjects. Average is calculated by dividing the sum of the two relevant columns (e.g., 59.0% is from the sum of Column 3 / the sum of Column 2).

Subject Name	**Total Params**	**Non-prim Params**	**Non-prim / Params**	**Non-prim Params w/ Factory**	**w/ Factory / Non-prim Params**
Atom	456	290	63.6%	66	22.8%
BBCode	33	9	27.3%	0	0.0%
ConcurrentList	16	0	0.0%	0	-
Functional-dotnet	50	5	10.0%	2	40.0%
Henoch	54	48	88.9%	0	0.0%
OpenMheg	75	55	73.3%	0	0.0%
PFDS	10	10	100.0%	0	0.0%
QuickGraph	125	111	88.8%	21	18.9%
SerialProtocol	51	21	41.2%	12	57.1%
Shweet	21	1	4.8%	0	0.0%
Utilities-net	66	15	22.7%	0	0.0%
Average			**59.0%**		**17.9%**

to a non-primitive parameter enables Pex to use factory methods such as the one depicted in Figure 10 to generate high-quality input values. Without promoting the `BinaryHeap` object to a parameter and using a factory method such as the one in Figure 10, the input values generated by Pex with the 16 PUTs can cover only 13% of the code blocks in the `BinaryHeap` class as opposed to 80% when the `BinaryHeap` object is promoted and a factory method is provided for it. When developers do not promote non-primitive objects to a non-primitive parameter or provide factory methods for it, the effectiveness of their tests really depends on the values that the developers use to initialize the objects in their tests. For example, if developers do not promote the `BinaryHeap` object to a parameter or provide factory methods for it, then depending on the values that the developers would use to initialize the `BinaryHeap` object, the code blocks covered by the 16 PUTs could actually range from 13% to 80% (the same as that achieved by promoting the `BinaryHeap` object to a parameter and providing a factory method for it). Promoting the `BinaryHeap` object to a parameter and providing factory methods for it not only enable tools such as Pex to generate objects of `BinaryHeap` that the developers may not have thought of themselves, but also alleviate the burden of developers to choose the right values for their tests to properly exercise the code under test. It is important to note that if we just promote the `BinaryHeap` object in the 16 PUTs but do not provide a factory method for it, the percentage of code blocks covered by the PUTs is 52%. Our findings here suggest that to enable tools such as Pex to generate input values that cover the most code, it is desirable to promote non-primitive objects in PUTs to non-primitive parameters and provide factory methods for such parameters. However, even if no factory methods are provided, simply promoting non-primitive objects in PUTs may already increase the code coverage achieved by the input values generated by tools such as Pex.

```
[PexMethod(MaxRuns = 100)]
[PexAllowedExceptionFromTypeUnderTest(typeof(InvalidOperationException))]
public void InsertAndRemoveMinimum<TPriority, TValue>(
  [PexAssumeUnderTest]BinaryHeap<TPriority, TValue> target,
  [PexAssumeNotNull] KeyValuePair<TPriority, TValue>[] kvs)
{
  var count = target.Count;
  foreach (var kv in kvs)
    target.Add(kv.Key, kv.Value);
  TPriority minimum = default(TPriority);
  for (int i = 0; i < kvs.Length; ++i)
  {
    if (i == 0)
      minimum = target.RemoveMinimum().Key;
    else
    {
      var m = target.RemoveMinimum().Key;
      Assert.IsTrue(target.PriorityComparison(minimum, m) <= 0);
      minimum = m;
    }
    AssertInvariant(target);
  }
  Assert.AreEqual(0, target.Count);
}
```

Figure 9 `InsertAndRemoveMinimum` PUT from the `BinaryHeapTest` class of QuickGraph [11].

```
[PexFactoryMethod(typeof(BinaryHeap<int, int>))]
public static BinaryHeap<int, int> Create(int capacity)
{
  var heap = new BinaryHeap<int, int>(capacity, (i, j) => i.CompareTo(j));
  return heap;
}
```

Figure 10 Factory method for the `BinaryHeapTest` class of QuickGraph [11].

5.2.2 Promoting Receiver Object

To determine how often developers could have replaced a hard-coded method sequence with a non-primitive parameter, we manually inspect each PUT to determine the number of them that could have had their receiver objects be replaced with a non-primitive parameter. We consider an object of a PUT to be a receiver object if the object directly or indirectly affects the PUT's assertions. The detailed results of our manual inspection effort can be found on our project website [10] under "PUT Patterns". As shown in Table 7, 95.7% (709/741) of the PUTs in our subjects have at least one receiver object. However, we find that 49.2% (349/709) of these PUTs with receiver objects do not have a parameter for the receiver objects, and 89.4% (312/349) of them can actually be modified so that all receiver objects in the PUT are promoted to PUT parameters. As shown in Table 8, we categorize the 349 PUTs whose receiver objects could be promoted into the following four different categories. (1) In 47.9% (167/349) of the PUTs, we can easily promote their receiver objects into a non-primitive parameter (e.g., removing the object creation line and adding a parameter). (2) In 41.5% (145/349) of the PUTs, their receiver objects are static (which cannot be instantiated). (3) In 9.7% (34/349) of the PUTs, they are testing their receiver objects' constructors. (4) In 1.6% (3/349) of the PUTs, they are testing multiple receiver objects with shared variables (e.g., testing the `equals` method of an object).

Of the PUTs belonging to the first category shown in Table 8, 33.0% (55/167) of them test specific object states. Figure 11 shows an example of a PUT that tests a specific object state. The developers of this PUT could have promoted `_list` and `element` to parameters and updated `index` accordingly before the assertion in Line 9. Figure 12 depicts a more general version of the PUT in Figure 11. Notice how the initial contents of the list and the element being added to the list are hard-coded in Figure 11 but not in Figure 12.

Table 7 Statistics of PUTs with receiver objects (ROs).

Subject Name	**# of PUTs w/ ROs**	**# of PUTs w/o promoted ROs**	**# of PUTs whose ROs should be promoted**
Atom	90.4% (217/240)	59.4% (129/217)	98.4% (127/129)
BBCode	88.2% (15/ 17)	100.0% (15/ 15)	100.0% (15/ 15)
ConcurrentList	100.0% (23/ 23)	56.5% (13/ 23)	100.0% (13/ 13)
Functional-dotnet	85.4% (35/ 41)	91.4% (32/ 35)	100.0% (32/ 32)
Henoch	100.0% (63/ 63)	25.4% (16/ 63)	43.8% (7/ 16)
OpenMheg	100.0% (45/ 45)	25.0% (11/ 45)	18.2% (2/ 11)
PFDS	100.0% (10/ 10)	100.0% (10/ 10)	100.0% (10/ 10)
QuickGraph	99.5% (204/205)	20.1% (41/204)	73.2% (30/ 41)
SerialProtocol	100.0% (34/ 34)	55.9% (19/ 34)	68.4% (13/ 19)
Shweet	100.0% (12/ 12)	100.0% (12/ 12)	100.0% (12/ 12)
Utilities-net	100.0% (51/ 51)	100.0% (51/ 51)	100.0% (51/ 51)
Total	**95.7% (709/741)**	**49.2% (349/709)**	**89.4% (312/349)**

```
[PexMethod]
public void GetItem(int index) {
  IList<int> _list = new ConcurrentList<int>();
  PexAssume.IsTrue(index >= 0);
  const int element = 5;
  for (int i = 0; i < index; i++)
    _list.Add(0);
  _list.Add(element);
  Assert.That(_list[index], Is.EqualTo(element));
}
```

Figure 11 PUT testing a specific object state in ConcurrentList [2].

```
[PexMethod]
public void GetItem_Promoted(int index, IList<int> _list, int element) {
  int size = _list.Count;
  PexAssume.IsTrue(index >= 0);
  for(int i = 0; i < index; i++)
    _list.Add(0);
  _list.Add(element);
  index += size;
  Assert.That(_list[index], Is.EqualTo(element));
}
```

Figure 12 Version of the PUT in Figure 11 with receiver object promoted.

Upon further investigation, we find that the 145 PUTs in the second category shown in Table 8 can and should actually be promoted by making the class under test not be static. On the other hand, the PUTs that test their receiver objects' constructors have no need to be improved by promotion. Lastly, the PUTs that test multiple receiver objects are best left not promoted. In the end we find that the 167 PUTs in the first category (their receiver objects can be easily promoted) and the 145 PUTs in the second category (their receiver objects are static) are PUTs whose receiver objects could be promoted and they should actually be promoted. These two categories of PUTs form the total of 89.4% (312/394) of the PUTs that could be promoted and should be promoted. Promoting these objects enables test generation tools such as Pex to use factory methods to generate different states of the receiver objects (beyond specific hard-coded ones) for the PUTs.

Based on our promotion experiences, often the time, after we promote receiver objects (resulted from hard-coded method sequences) to non-primitive parameters of PUTs, we need to add assumptions to constrain the non-primitive parameters so that test generation tools

Table 8 Categorization results of the PUTs whose receiver objects could be promoted.

Category	**#PUTs**
(1) Their receiver objects can be easily promoted	167 (47.9%)
(2) Their receiver objects are static	145 (41.5%)
(3) Testing their receiver objects' constructors	34 (9.7%)
(4) Testing multiple receiver objects with shared variables	3 (0.9%)
Total	**349**

```
[TestMethod]
public void GetItem_CUT()
{
  GetItem_Promoted(0, null, 5);
}
```

Figure 13 Example of a CUT generated from the PUT in Figure 12.

will not generate input values that are of no interest to developers. For example, for the `GetItem_Promoted` PUT in Figure 12, one of the input values generated by Pex with this PUT can be found in Figure 13. Although the value of `index` (0) from the `GetItem_CUT` in Figure 13 is reasonable for both the `GetItem` and `GetItem_Promoted` PUTs and the value of `element` (5) is reasonable for the `GetItem_Promoted` PUT, the additional value of `_list` (null) is unreasonable. The value is unreasonable because the `GetItem` PUT is expected to test adding various elements to `_list` but it is not expected to test the case when `_list` is null. However, due to our promotion of `_list`'s hard-coded method sequence to a non-primitive parameter, input values generated from `GetItem_Promoted` would actually test such a case. In order for developers to prevent such nonsensical input values from being generated, the developers would have to add the assumption of `PexAssume.IsNotNull(_list)` before Line 3 of `GetItem_Promoted`. Such assumption writing can be time-consuming: essentially promoting hard-coded method sequences to be non-primitive parameters and adding assumptions to these parameters are going from specifying "how" (to generate specific object states) to specifying "what" (specific object states need to be generated).

5.2.3 Implications

There are a significant number of receiver objects (in the PUTs written by developers) that could be promoted to non-primitive parameters, and a significant number of existing non-primitive parameters that lack factory methods. It is worthwhile for tool researchers or vendors to provide effective tool support to assist developers to promote these receiver objects (resulted from hard-coded method sequences), e.g., inferring assumptions for a non-primitive parameter promoted from hard-coded method sequences. Additionally, once hard-coded method sequences are promoted to non-primitive parameters, developers can also use assistance in writing effective factory methods for such parameters.

5.3 RQ3. PUT Design Patterns and Bad Smells

Our categorization of forum posts as described in Section 3 shows that 5 out of 11 of the PUT concept/guideline posts discuss patterns in which PUTs should be written in. For example, two of the posts titled "Assertions in PUT" and "PUT with PEX" involve answers informing the original poster that assertions are typically necessary for PUTs. One such forum post contains the following response: "You should write Asserts, in order to ensure

```
[PexMethod]
public void Clear<T>([PexAssumeUnderTest]ConcurrentList<T> target) {
  target.Clear();
}
```

Figure 14 PUT (in ConcurrentList [2]) that should be improved with assertions.

Table 9 Categorization results of bad smells in PUTs

Category	**#PUTs**
(1) Code duplication	55
(2) Unnecessary conditional statement	39
(3) Hard-coded test data	37
Total	**131**

that the Function (TestInvoice in this case) really does what it is intended to do". To better understand how developers write PUTs, we manually inspect all of the PUTs in our subjects to see what the common design patterns and bad smells are. The detailed results of our manual inspection effort can be found on our project website [10] under "PUT Patterns".

5.3.1 PUT Design Patterns

We find that the majority of the PUTs are written in the following patterns: "AAA" (Triple-A) and Parameterized Stub. Triple-A is a well-known design pattern for writing unit tests [8]. These tests are organized into three sections: setting up the code under test (Arrange), exercising the code under test (Act), and verifying the behavior of the code under test (Assert). On the other hand, a Parameterized Stub test is used to test the code under test that already contains many assertions (e.g., code equipped with code contracts [25, 30, 16]). In general, Parameterized Stub tests are easy to write and understand, since the test body is short and contains only a few method calls to the code under test. In our subjects, we find that 34.6% (270/741) and 32.1% (251/741) of the PUTs to exhibit the Triple-A and Parameterized Stub test pattern, respectively. Of the 251 PUTs that exhibit the Parameterized Stub pattern, we find that 74.5% (187/251) of them are PUTs that should be improved with assertions, given that the code under test itself does not contain any code-contract assertions or any other type of assertions. For example, the PUT in Figure 14 contains only a single statement to test the robustness of the `Clear` method, which by itself does not contain any assertions. Developers of this PUT should at least add an assertion such as `Assert.That(target.Count, Is.EqualTo(0));` to the end of the PUT to ensure that once `Clear` is invoked, then the number of elements in a `ConcurrentList` object will be 0.

Similar to the bad smells typically found in conventional unit tests [29], we consider the following three categories of bad smells in our PUTs: (1) code duplication, (2) unnecessary conditional statement, and (3) hard-coded test data. These three categories of bad smells can cause tests to be difficult to understand and maintain. Table 9 shows the number of PUTs containing each category of bad smells. Our analysis tools as described in Section 4.2 assist our manual inspections of the PUTs by listing the PUTs that contain conditional statements or hard-coded test data (as arbitrary strings). Using these lists of PUTs, we then manually inspect each of these PUTs to determine whether it really has bad code smells. To determine code duplication, we manually compare every PUT with every other PUT of the same class. Next, we discuss each of the categories in detail.

```
1  [PexMethod]
2  public void GetItem(int index)
3  {
4    PexAssume.IsTrue(index >= 0);
5    const int element = 5;
6    for (int i = 0; i < index; i++)
7    {
8      _list.Add(0);
9    }
10   _list.Add(element);
11   Assert.That(_list[index], Is.EqualTo(element));
12 }
```

Figure 15 PUT (from the `ConcurrentListHandWrittenTests` class of ConcurrentList [2]) that contains many lines of test-code duplication with another PUT named `SetItem` from the same class.

Table 10 Categorization results of why conditional statements exist in PUTs.

Category	#PUTs
(1) Testing particular cases	16
(2) Forcing Pex to explore particular cases	9
(3) Testing different cases according to boolean conditions	9
(4) Unnecessary `if` statements	5
Total	**39**

5.3.2 Code Duplication in PUTs

Similar to conventional unit tests, PUTs also contain the bad smell of test-code duplication. Test-code duplication is a poor practice because it increases the cost of maintaining tests. Duplication often arises when developers clone tests and do not put enough thought into how to reuse test logic intelligently. As the number of tests increases, it is important that the developers either factor out commonly used sequences of statements into helper methods that can be reused by various tests, or in the case of PUTs, consider merging the PUTs and using assumptions/attributes to ensure that the specific cases being tested previously are still tested. In our subjects' PUTs, we find that 7.4% (55/741) of them contain test-code duplication. In other words, for 55 of our subjects' PUTs, there exist another PUT (in the same subject) that contains a significant amount of duplicate test code. One example of such PUT is shown in Figure 15. The PUT in this example is from the `ConcurrentListHandWrittenTests` class of ConcurrentList [2] and is almost identical to another PUT named `SetItem` in the same class. More specifically, the only lines that differ between the two PUTs are Lines 6 and 10. For Line 6 the loop terminating condition is set to `i <= index` as opposed to `i < index`. For Line 10, instead of adding an element with the `Add` method, the line is `_list[index] = element;`. In .NET, the use of brackets and an index value to add elements to a collection is enabled by Indexers [14]. Since the intention of the two PUTs is to test whether setting and getting an element from a list of arbitrary size correctly set and get the correct element, the two differences in Lines 6 and 10 between the two PUTs actually do not matter. Instead of duplicating so many lines of test code, the developers of these two PUTs should just delete one of them. Doing so will not only help decrease the cost for developers to maintain the tests, but also to speed up the testing time, since there will be fewer tests that cover the same parts of the code under test. Developers can also make use of existing tools for detecting code clones [18, 19] to automatically help detect code duplication in PUTs.

```
IList<int> _list = new ConcurrentList<int>();
[PexMethod(MaxBranches = 20000)]
public void Clear(int count)
{
  var numClears = 100;
  var results = new List<int>(numClears * 2);
  var numCpus = Environment.ProcessorCount;
  var sw = Stopwatch.StartNew();
  using (SaneParallel.For(0, numCpus, x =>
  {
    for (var i = 0; i < count; i++)
      _list.Add(i);
  }))
  {
    for (var i = 0; i < numClears; i++)
    {
      Thread.Sleep(100);
      results.Add(_list.Count);
      _list.Clear();
      results.Add(_list.Count);
    }
  }
  sw.Stop();
  for (var i = 0; i < numClears; i++)
    Console.WriteLine("Before/After Clear #{0}: {1}/{2}", i, results[i << 1], results[(i << 1) + 1]);
  Console.WriteLine("ClearParallelSane took {0}ms", sw.ElapsedMilliseconds);
  _list.Clear();
  Assert.That(_list.Count, Is.EqualTo(0));
}
```

Figure 16 PUT with hard-coded test data in the `SaneParallelTests` class of ConcurrentList [2].

5.3.3 Unnecessary Conditional Statements in PUTs

Typically developers are expected not to write any conditional statements in their tests, because tests should be simple, linear sequences of statements. When a test has multiple execution paths, one cannot be sure exactly how the test will execute in a specific case. In our subjects, 7.0% (52/741) of the PUTs contain at least one conditional branch. To understand why developers write PUTs with conditionals, we study whether the conditionals in these PUTs are necessary and if they are not, why the developers write such conditionals in their PUTs. We find that 25% (13/52) of the PUTs contain conditional statements that could not be removed. These PUTs are typically testing the interactions of two or more operations of the code under test (e.g., adding and removing from a data structure). The remaining 75.0% (39/52) of the PUTs with conditionals can have their conditionals removed or each of these PUTs should be split into two or more PUTs. Table 10 shows the reasons for why the conditionals of such PUTs should be removed and the number of PUTs for each of the reasons. The PUTs in the first and second categories should replace their conditionals with `PexAssume()` statements to force Pex to explore and test particular cases. The PUTs in the third category should be each split into multiple PUTs each of which tests a different case of the conditional. For the PUTs created from the third category, developers can use `PexAssume()` statements in the new PUTs to filter out inputs that do not satisfy the boolean conditions of the case that the new PUTs are responsible for. The PUTs in the last category contain conditionals that can be removed with a slight modification to the test (e.g., some conditionals in a loop can be removed by amending the loop and/or adding code before the loop). The automatic detection and fixing of unnecessary conditional statements in PUTs would be a valuable and challenging line of future work due to the following. There are various reasons for why a PUT may have conditionals as shown in Table 10, and depending on the reason why a PUT may have conditionals, the fix for removing the conditionals, if removal is possible, can be quite different.

5.3.4 Hard-coded Test Data in PUTs

Another bad smell that we identify in our subjects' PUTs is hard-coded test data. This smell can be problematic for three main reasons. (1) Tests are more difficult to understand. A developer debugging the tests would need to look at the hard-coded data and deduce how each value is related to another and how these values affect the code under test. (2) Tests are more likely to be flaky [28, 22, 15]. A common reason for tests to be flaky is the reliance on external dependencies such as databases, file system, and global variables. Hard-coded data in these tests often lead to multiple tests modifying the same external dependency and these modifications could cause these tests to fail unexpectedly. (3) Hard-coded test data prevent automatic test generation tools such as Pex from generating high-quality input values. In our subjects' PUTs, we find that 5.0% (37/741) of them use hard-coded test data. One example of such PUT is shown in Figure 16. In this example, the developers are testing the `Clear` method of the `ConcurrentList` object (`_list`). The PUT adds an arbitrary number of elements to the `_list` object, clears the list, and records the number of elements in the list. The process of adding and clearing the list repeats 100 times as decided by `numClears` on Line 5. As far as we can tell, the developers arbitrarily choose the value of 100 for `numClears` on Line 5. When we parameterize the `numClears` variable and add an assumption that the variable should be between 1 and 1073741823 (to prevent `ArgumentOutOfRangeException`), we find that the input values generated by Pex for the `numClears` variable to be 1 and 2. These two values exercise the same lines of the `Clear` method just as the value of 100 would. The important point here is that contrary to the developers' arbitrarily chosen value of 100, Pex is able to systematically find that using just the values of 1 and 2 would already sufficiently test the `Clear` method. That is, as we manually confirm, even if the developers devote more computation time to testing the `Clear` method by setting `numClears` to 100, they would not cover any additional code or find any additional test failures. Therefore, the developers of this PUT should not hard code the test data, and instead they should parameterize the `numClears` variable. Doing so would enable automatic test generation tools such as Pex to generate high-quality input values that sufficiently test the code under test. Developers can also make use of existing program analysis tools [41] to automatically detect whether certain hard-coded test data may exist between multiple PUTs.

5.3.5 Implications

By understanding how developers write PUTs, testing educators can suggest ways to improve PUTs. For example, developers should consider splitting PUTs with multiple conditional statements into separate PUTs each covering a case of the conditional statements. Doing so makes the developer's PUTs easier to understand and eases the effort to diagnose the reason for test failures. Tool vendors and researchers can incorporate this data with their tools to check the style of PUTs for better suggestions on how the PUTs can be improved. For example, checking whether a PUT is a Parameterized Stub, contains conditionals, contains hard-coded test data, and contains duplicate test code often correctly identifies a PUT that can be improved.

6 Threats to Validity

There are various threats to validity in our study. We broadly divide the main threats into internal and external validity.

6.1 Internal Validity

Threats to internal validity are concerned with the validity of our study procedure. Due to the complexity of software, faults in our analysis tools could have affected our results. However, our analysis tools are tested with a suite of unit tests, and samples of the results are manually verified. Results from our manual analyses are confirmed by at least two of the authors. Furthermore, we rely on various other tools for our study, such as dotCover [3] to measure the code coverage of the input values generated by Pex. These tools could have faults as well and consequently such faults could have affected our results.

6.2 External Validity

There are two main threats to external validity in our study.

1. We use the categorization of the Microsoft MSDN Pex Forum posts [31] to determine the issues surrounding parameterized unit testing. These forum posts enable us and the research community to access the issues of developers objectively and quantitatively, but the issues identified from the posts may not be representative of all the issues that developers encounter.
2. Our findings may not apply to subjects other than those that we study, especially since we are able to find only 11 subjects matching the criteria defined in Section 4. Furthermore, we primarily focus on projects using PUTs in the context of automated test generation, so our findings from such subjects may not generalize to situations outside of this setting (e.g., general usage of Theories [33] in Java). In addition, our analyses focus specifically on subjects that contain PUTs written using the Pex framework, and the API differences or idiosyncrasies of other frameworks may impact the applicability of our findings. All of our subjects are written in C#, but vary widely in their application domains and project sizes. Finally, all of our subjects are open source software, and therefore our findings may not generalize to proprietary software.

7 Related Work

To the best of our knowledge, our characteristic study is the first on parameterized unit testing in open source projects. In contrast, previous work focuses on proposing new techniques for parameterized unit testing and does not provide any insight on the practices of parameterized unit testing. For example, Xie et al. [43] propose a technique for assessing the quality of PUTs using mutation testing. Thummalapenta et al. [36] propose manual retrofitting of CUTs to PUTs, and show that new faults are detected and coverage is increased after such manual retrofitting is conducted. Fraser et al. [21] propose a technique for generating PUTs starting from concrete test inputs and results.

Our work is related to previous work on studying developer-written formal specifications such as code contracts [16]. Schiller et al. [34] conduct case studies on the use of code contracts in open source projects in C#. They analyze 90 projects using code contracts and categorize their use of various types of specifications, such as null checks, bound checks, and emptiness checks. They find that checks for nullity and emptiness are the most common types of specifications. Similarly we find that the most common types of PUT assumptions are also used for nullness specification. However, the most common types of PUT assertions are used for equality checking instead of null and emptiness.

Estler et al. [20] study code contract usage in 21 open source projects using JML [27] in Java, Design By Contract in Eiffel [30], and code contracts [16] in C#. Their study also includes an analysis of the change in code contracts over time, relative to the change in the specified source code. Their findings agree with Schiller's on the majority use of nullness code contracts. Furthermore, Chalin [17] studies code contract usage in over 80 Eiffel projects. They show that programmers using Eiffel tend to write more assertions than programmers using any other languages do.

8 Conclusion

To fill the gap of lacking studies of PUTs in development practices of either proprietary or open source software, we have presented categorization results of the Microsoft MSDN Pex Forum posts (contributed primarily by industrial practitioners) related to PUTs. We then use the categorization results to guide the design of the first characteristic study of parameterized unit testing in open source projects. Our study involves hundreds of PUTs that open source developers write for various open source projects.

Our study findings provide the following valuable insights for various stakeholders such as current or prospective PUT writers (e.g., developers), PUT framework designers, test-generation tool vendors, testing researchers, and testing educators.

1. We have studied the extents and types of assumptions, assertions, and attributes being used in PUTs. Our study has identified assumption and assertion types that tool vendors or researchers can incorporate with their tools to better infer assumptions and assertions to assist developers. For example, tool vendors or researchers who care about the most commonly used assumption types should focus on `PexAssumeUnderTest` or `PexAssumeNotNull`, since these two are the most commonly used assumption types. We have also found that increasing the default value of attributes as suggested by tools such as Pex rarely contributes to increased code coverage. Tool vendors or researchers should aim to improve the quality of the attribute recommendations provided by their tools, if any are provided at all.
2. We have studied how often hard-coded method sequences in PUTs can be replaced with non-primitive parameters and how useful it is for developers to do so. Our study has found that there are a significant number of receiver objects in the PUTs written by developers that could be promoted to non-primitive parameters, and a significant number of existing non-primitive parameters that lack factory methods. Tool researchers or vendors should provide effective tool support to assist developers to promote these receiver objects (resulted from hard-coded method sequences), e.g., inferring assumptions for a non-primitive parameter promoted from hard-coded method sequences. Additionally, once hard-coded method sequences are promoted to non-primitive parameters, developers can also use assistance in writing effective factory methods for such parameters.
3. We have studied the common design patterns and bad smells in PUTs, and have found that there are a number of patterns that often correctly identify a PUT that can be improved. More specifically, checking whether a PUT is a Parameterized Stub, contains conditionals, contains hard-coded test data, and contains duplicate test code often correctly identifies a PUT that can be improved. Tool vendors and researchers can incorporate this data with their tools to check the style of PUTs for better suggestions on how these PUTs can be improved.

The study is part of our ongoing industry-academia team efforts for bringing parameterized unit testing to broad industrial practices of software development.

References

1 Atom. URL: https://github.com/tivtag/Atom.

2 ConcurrentList. URL: https://github.com/damageboy/ConcurrentList.

3 dotCover. URL: https://www.jetbrains.com/dotcover.

4 GitHub code search. URL: https://github.com/search.

5 The .NET compiler platform Roslyn. URL: https://github.com/dotnet/roslyn.

6 NUnit Console. URL: https://github.com/nunit/nunit-console.

7 OpenMheg. URL: https://github.com/orryverducci/openmheg.

8 Parameterized Test Patterns for Microsoft Pex. URL: http://citeseerx.ist.psu.edu/viewdoc/download?rep=rep1&type=pdf&doi=10.1.1.216.282.

9 Parameterized tests in JUnit. URL: https://github.com/junit-team/junit/wiki/Parameterized-tests.

10 PUT study project web. URL: https://sites.google.com/site/putstudy.

11 QuickGraph. URL: https://github.com/tathanhdinh/QuickGraph.

12 SearchCode code search. URL: https://searchcode.com.

13 Theories in JUnit. URL: https://github.com/junit-team/junit/wiki/Theories.

14 Using Indexers (C# Programming Guide). URL: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/indexers/using-indexers.

15 Stephan Arlt, Tobias Morciniec, Andreas Podelski, and Silke Wagner. If A fails, can B still succeed? Inferring dependencies between test results in automotive system testing. In *ICST 2015: Proceedings of the 8th International Conference on Software Testing, Verification and Validation*, pages 1–10, Graz, Austria, April 2015.

16 Michael Barnett, Manuel Fähndrich, Peli de Halleux, Francesco Logozzo, and Nikolai Tillmann. Exploiting the synergy between automated-test-generation and programming-by-contract. In *ICSE 2009: Proceedings of the 31st International Conference on Software Engineering*, pages 401–402, Vancouver, BC, Canada, May 2009.

17 Patrice Chalin. Are practitioners writing contracts? In *Rigorous Development of Complex Fault-Tolerant Systems*, pages 100–113. 2006.

18 Yingnong Dang, Dongmei Zhang, Song Ge, Chengyun Chu, Yingjun Qiu, and Tao Xie. XIAO: Tuning code clones at hands of engineers in practice. In *ACSAC 2012: Proceedings of 28th Annual Computer Security Applications Conference*, pages 369–378, Orlando, FL, USA, December 2012.

19 Yingnong Dang, Dongmei Zhang, Song Ge, Ray Huang, Chengyun Chu, and Tao Xie. Transferring code-clone detection and analysis to practice. In *ICSE 2017: Proceedings of the 39th International Conference on Software Engineering, Software Engineering in Practice (SEIP)*, pages 53–62, Buenos Aires, Argentina, May 2017.

20 H-Christian Estler, Carlo A Furia, Martin Nordio, Marco Piccioni, and Bertrand Meyer. Contracts in practice. In *FM 2014: Proceedings of the 19th International Symposium on Formal Methods*, pages 230–246. Singapore, May 2014.

21 Gordon Fraser and Andreas Zeller. Generating parameterized unit tests. In *ISSTA 2011: Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 364–374, Toronto, ON, Canada, July 2011.

22 Zebao Gao, Yalan Liang, Myra B. Cohen, Atif M. Memon, and Zhen Wang. Making system user interactive tests repeatable: When and what should we control? In *ICSE 2015: Proceedings of the 37th International Conference on Software Engineering*, pages 55–65, Florence, Italy, May 2015.

23 Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *PLDI 2005: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, Chicago, IL, USA, June 2005.

24 John V. Guttag and James J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, pages 27–52, 1978.

25 C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, pages 576–580, October 1969.

26 Pratap Lakshman. Visual Studio 2015 Build better software with Smart Unit Tests. *MSDN Magazine*.

27 Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report TR 98-06i, Department of Computer Science, Iowa State University, June 1998.

28 Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *FSE 2014: Proceedings of the ACM SIGSOFT 22nd Symposium on the Foundations of Software Engineering*, pages 643–653, Hong Kong, November 2014.

29 Gerard Meszaros. *XUnit Test Patterns: Refactoring Test Code*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.

30 Bertrand Meyer. Applying "Design by Contract". *Computer*, pages 40–51, October 1992.

31 Microsoft. Pex MSDN discussion forum, April 2011. URL: `http://social.msdn.microsoft.com/Forums/en-US/pex`.

32 Microsoft. Generate unit tests for your code with IntelliTest, 2015. URL: `https://msdn.microsoft.com/library/dn823749`.

33 David Saff. Theory-infected: Or how I learned to stop worrying and love universal quantification. In *OOPSLA Companion: Proceedings of the Object-Oriented Programming Systems, Languages, and Applications*, pages 846–847, Montreal, QC, Canada, October 2007.

34 Todd W Schiller, Kellen Donohue, Forrest Coward, and Michael D Ernst. Case studies and tools for contract specifications. In *ICSE 2014: Proceedings of the 36th International Conference on Software Engineering*, pages 596–607, Hyderabad, India, June 2014.

35 Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *ESEC/FSE 2005: Proceedings of the 10th European Software Engineering Conference and the 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 263–272, Lisbon, Portugal, September 2005.

36 Suresh Thummalapenta, Madhuri R Marri, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. Retrofitting unit tests for parameterized unit testing. In *FASE 2011: Proceedings of the Fundamental Approaches to Software Engineering*, pages 294–309. Saarbrücken, Germany, March 2011.

37 Nikolai Tillmann and Jonathan De Halleux. Pex: White box test generation for .NET. In *TAP 2008: Proceedings of the 2nd International Conference on Tests And Proofs (TAP)*, pages 134–153, Prato, Italy, April 2008.

38 Nikolai Tillmann, Jonathan de Halleux, and Tao Xie. Parameterized unit testing: Theory and practice. In *ICSE 2010: Proceedings of the 32nd International Conference on Software Engineering*, pages 483–484, Cape Town, South Africa, May 2010.

39 Nikolai Tillmann, Jonathan de Halleux, and Tao Xie. Transferring an automated test generation tool to practice: From Pex to Fakes and Code Digger. In *ASE 2014: Proceedings of the 29th Annual International Conference on Automated Software Engineering*, pages 385–396, Västerøas, Sweden, September 2014.

40 Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests. In *ESEC/FSE 2005: Proceedings of the 10th European Software Engineering Conference and the 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 253–262, Lisbon, Portugal, September 2005.

41 Matias Waterloo, Suzette Person, and Sebastian Elbaum. Test analysis: Searching for faults in tests. In *ASE 2015: Proceedings of the 30th Annual International Conference on Automated Software Engineering*, pages 149–154, Lincoln, NE, USA, November 2015.

42 Xusheng Xiao, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. Precise identification of problems for structural test generation. In *ICSE 2011: Proceedings of the 33rd International Conference on Software Engineering*, pages 611–620, Waikiki, HI, USA, May 2011.

43 Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. Mutation analysis of parameterized unit tests. In *ICSTW 2009: Proceedings of the International Conference on Software Testing, Verification and Validation Workshops*, pages 177–181, Denver, CO, USA, April 2009.