Automatic Generation of Regression Tests via Mining Gigabytes of Dynamic Traces

Suresh Thummalapenta[1], Jonathan de Halleux[2], Nikolai Tillmann[2], Scott Wadsworth[3]
[1]Department of Computer Science, North Carolina State University, Raleigh
[2]Microsoft Research, One Microsoft Way, Redmond
[3]Microsoft Corporation, One Microsoft Way, Redmond
[1]sthumma@ncsu.edu, [2]{jhalleux, nikolait}@microsoft.com, [3]???

ABSTRACT

1. INTRODUCTION

Software maintenance is an important phase of software development life cycle. Software maintenance involves maintaining programs that evolve during their life time. One important aspect of software maintenance is to make sure that the changes made in the new version of software do not introduce any unwanted side effects in the existing functionality. Regression testing is a testing methodology that aims at exposing such unwanted side effects introduced in the new version of software. Rosenblum and Weyuker [11] describe that the majority of software maintenance costs is spent on regression testing.

A regression test is a unit test (also referred to as a conventional unit test) that is created on one version of software (often a stable version) and is executed on further versions of software to expose unwanted side effects. In general, a unit test includes three major components: test scenario, test data, and test assertions. Figure 1 shows an example unit test. In a unit test, test scenario refers to the method-call sequence shown in Statements 1, 2, and 3. Test data refers to the concrete values (such as 7 and 3 in Statements 2 and 3, respectively) passed as arguments to the method calls. Test assertions refer to assertions (Statement 4) that verify whether the actual behavior is the same as the expected behavior.

```
00:void AddTest() {
01:     HashSet set = new HashSet();
02:     set.Add(7);
03:     set.Add(3);
04:     Assert.IsTrue(set.Count == 2);
05:}
```

Figure 1: An example unit test.

```
00:void AddSpec(int x, int y) {
01:     HashSet set = new HashSet();
02:     set.Add(x);
03:     set.Add(y);
04:     Assert.AreEqual(x == y, set.Count == 1);
05:     Assert.AreEqual(x != y, set.Count == 2);
06:}
```

Figure 2: An example PUT.

Recent advancements in software testing introduced Parameterized Unit Tests (PUT) [14], which generalize conventional unit tests by accepting parameters. Figure 2 shows a PUT for the unit test shown in Figure 1, where concrete values in Statements 2 and 3 are replaced by the parameters x and y. An approach, called dynamic symbolic execution [1, 3, 8, 9], can be used to automatically generate a minimal set of conventional unit tests that achieve a high coverage of the code under test defined by PUT. Section **??** provides more details on how dynamic symbolic execution generates conventional unit tests from PUTs. In our approach, we use Pex [13] as an example state-of-the-art dynamic symbolic execution approach. A major advantage of PUTs compared to conventional unit tests is that test data is automatically generated based on the code under test. However, writing good PUTs can still be challenging since PUTs require test scenarios (method-call sequences) to exercise the code under test. Automatic generation of test scenarios is quite challenging due to a large search space of possible scenarios and valid scenarios are quite small. In literature, there exist three major categories of approaches that generate test scenarios in the form of method-call sequences: bounded-exhaustive [7,16], evolutionary [5,15], and random [2, 6, 10]. In our previous work [12], we show that these approaches are either not scalable or not effective in practice due to their random nature.

Our approach addresses the issue of test scenarios by automatically generating test scenarios from dynamic traces recorded during program execution. We use dynamic traces compared to static traces, since dynamic traces are more precise than static traces. These dynamic traces include two aspects: realistic scenarios of method-call sequences and concrete values passed as arguments to those method calls. Since recorded dynamic traces include both test scenarios and test data (concrete values passed as arguments to method calls in test scenarios), regression tests can directly be generated from the recorded dynamic traces. However, such regression tests exercise only happy paths such as paths that do not include error-handling code in the code under test. To address this issue, we first transform recorded dynamic traces into PUTs. We use concrete values in dynamic traces to generate conventional unit tests that are used as seed tests to increase the efficiency of dynamic symbolic execution while exploring PUTs [4].

Since the dynamic traces are recorded during program execution, we identify that many of recorded traces are duplicates. The reason for duplicates is that the same method-call sequences can get invoked multiple times. Consequently, we have many duplicate PUTs and seed tests. Exploration of such duplicate PUTs is redundant and can also lead to scalability issues. Therefore, we first filter out duplicate PUTs and seed tests by using static and dynamic analyses, respectively. We next explore the remaining PUTs to generate regression tests that can achieve a high coverage of the code under test. In our evaluations (and also in practice), we identify that even after minimization of PUTs and seed tests, the remaining number of PUTs can still be many and can take long time in exploring those PUTs. To address this issue, we develop a distributed setup that allows parallel exploration of PUTs. To infer test assertions, we execute the generated regression tests on a stable version of software and capture the return values of observer methods. These observer methods are pure methods that do not change the state of their receiver or arguments. For example, `Hashset.Count` shown in Statement 4 of Figure 1 is an observer method.

To the best of our knowledge, ours is the first scalable approach that automatically generates regression tests without requiring any manual efforts. In our evaluations, we show that our approach handles ≈ 1.5GB of dynamic traces and generates ≈ 500,000 regression tests on ten .NET base class libraries. These numbers show that our approach is scalable and can be used in practice to deal with real-world applications.

In summary, this paper makes the following major contributions:

- A technique to record dynamic traces during program execution and generate PUTs and seed unit tests from recorded dynamic traces.
- A technique to filter out duplicate PUTs and seed unit tests by using static and dynamic analyses, respectively.
- A distributed setup for the parallel exploration of PUTs to generate conventional unit tests.
- Three large-scale evaluations to show the effectiveness of our approach. In our approach, we recorded ≈1.5 GB (including 433,809) of dynamic traces related to ten .NET base class libraries. From these PUTs, our approach generated 501,799 regression tests that achieved a high code coverage of the ten .NET base class libraries.

The rest of the paper is structured as follows: Section **??** presents background on a DSE-based approach. Section **??** explains our approach with an example. Section 2 describes key aspects of our approach. Section **??** presents our evaluation results. Section **??** discusses threats to validity. Section **??** discusses limitations of our approach and future work. Section **??** presents related work. Finally, Section **??** concludes.

2. APPROACH

Figure 3 shows the high-level overview of our approach. Our approach includes three major phases: *capture*, *minimize*, and *explore*. In the capture phase, our approach records dynamic traces from program executions. Our approach next transforms these dynamic traces into PUTs and seed tests. Among recorded traces, we identify that there many duplicate traces since the same sequence of method calls can get invoked multiple times during program executions. Consequently, the generated PUTs and seed tests also include duplicates. In the minimize phase, we use a combination of static and dynamic analyses to filter out duplicate PUTs and seed tests, respectively. In the explore phase, we use Pex to explore PUTs to generate regression tests that achieve a high coverage of the code under test. We next explain each phase in detail.

```
01: TagRegex tagex = new TagRegex();
02: Match mc = ((Regex)tagex).Match(”<% Page..\u000a”,108);
03: Capture cap = (Capture) mc;
04: int indexval = cap.Index;
```

Figure 4: An example dynamic trace recorded by the capture phase.

PUT:

```
00: public static void $F_1$(string $VAL_1$, int $VAL_2$, out int $OUT_1$)
01:       TagRegex tagex = new TagRegex();
02:       Match mc = ((Regex)tagex).Match($VAL_1$, $VAL_2$);
03:       Capture cap = (Capture) mc;
04:       $OUT_1$ = cap.Index;
05: }
```

Seed Test:

```
06: public static void $T_1$() {
07:       int index;
08:       $F_1$(”<%@ Page..\u000a”, 108, out index);
09: }
```

Figure 5: A PUT and a seed test generated from the dynamic trace in Figure 4.

2.1 Capture Phase

In the capture phase, our approach records dynamic traces from program executions. The capture phase uses a profiler that records method calls invoked by the application during execution. The capture phase records both the method calls invoked and the concrete values passed as arguments to those method calls. Figure 4 shows an example dynamic trace recorded by the capture phase. Statement 2 shows the concrete value “`<% Page..\u000a`” passed as an argument for the `Match` method. Our recorded traces are complete and do not require any other primitive values or non-primitive objects. Our recorded traces include two kinds of methods: state-modifying and observer methods. A method is referred to as a state-modifying method if the method affects (i.e., writes) at least one field of its declaring class. A method is referred to as an observer method if the method does not affect any fields of its declaring class and also returns a non-void type.

Our approach next generates PUTs and seed tests from recorded traces. To generate PUTs, our approach identifies all constant values and promotes those constant values as parameters. Furthermore, our approach identifies return values of observer methods in the PUT and promotes those return values as `OUT` parameters for the PUT. In C#, these `OUT` parameters represent the return values of a method. Our approach next generates seed tests that includes all concrete values from the dynamic traces. Figure 5 shows a PUT and a seed test generated from the dynamic trace shown in Figure 4.

The generated PUT includes two parameters and one `OUT`

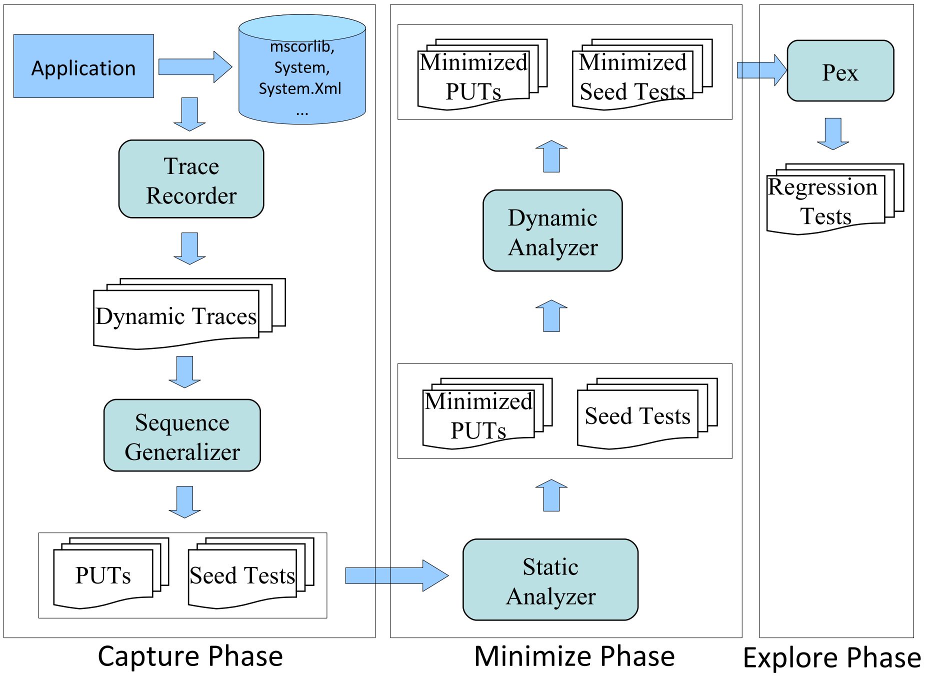

Figure 3: A high-level overview of our approach

parameter. The `OUT` parameter is the return value of the observer method `Capture.Index`. These `OUT` parameters are later used to generate test assertions in regression tests (Section 2.3). The figure also shows a seed test generated from the dynamic trace. The seed test includes concrete values of the dynamic trace and invokes the generated PUT with those concrete values.

2.2 Minimize Phase

Our approach records dynamic traces during actual program executions. As the same sequence of method calls can be invoked multiple times during program executions, we identify that there are many duplicates among recorded dynamic traces. Consequently, there are many duplicates among generated PUTs and seed tests. In the minimize phase, our approach filters out duplicate PUTs and seed tests. The primary reason for filtering out duplicates is that exploration of duplicate PUTs is redundant and can also lead to scalability issues while generating regression tests.

We first present our criteria for a duplicate PUT and a seed test and next explain how we filter out such duplicate PUTs and seed tests.

Duplicate PUT: We consider a PUT, say P_1, as a duplicate of another PUT, say P_2, if both P_1 and P_2 have the same sequence of method calls.

Duplicate Seed Test: We consider a seed test, say S_1, as a duplicate of another seed test, say S_2, if both S_1 and S_2 exercise the same execution path.

We use PUTs and seed tests shown in Figure 6 as illustrative examples. The figure shows two PUTs and three seed tests. Our approach uses static analysis to identify duplicate PUTs. For example, our approach compares the method bodies of `PUT1` and `PUT2` at the level of Microsoft Intermediate Language[1] instructions. Our approach ignores any primitive values related to local variables in the PUTs while comparing instructions. In this example, our approach considers `PUT2` as a duplicate of `PUT1`, although the local variable `c` in Statement 9 of `PUT1` is assigned a different value in `PUT2`. As `PUT2` is a duplicate of `PUT1`, our approach automatically replaces the `PUT2` method call in `SeedTest2` with `PUT1`.

After filtering out duplicate PUTs, our approach uses dynamic analysis for filtering out duplicate seed tests. To identify duplicate seed tests, our approach executes each seed test and monitors its execution path in the code under test. For example, `SeedTest1` follows the path "2 → 6 → 10" in `PUT1`. Our approach considers `SeedTest2` as a duplicate of `SeedTest1`, as `SeedTest2` also follows the same path "2 → 6 → 10" in `PUT1`. Consider another unit test `SeedTest3` shown in Figure 6. Our approach does not consider `SeedTest3` as a duplicate of `SeedTest1` as `SeedTest3` follows the path "2 → 6 → 10 → 10", since `SeedTest3` iterates the loop in Statement 9 two times.

2.3 Explore Phase

In the explore phase, our approach uses Pex to generate regression tests from PUTs. Although seed tests generated in the capture phase can be used as regression tests, those seed tests exercise only the happy paths such as paths that do not include error-handling code in the code under test. Therefore, these seed tests do not achieve a high coverage of the code under test.

To address this issue, we use Pex to explore generated PUTs. Inspired by Patrice et al. [4], we use seed tests to assist Pex during exploration of PUTs. Using seed tests increases the effectiveness of Pex or any other dynamic-symbolic-execution-based approach in two major ways. First, seed tests cover several paths in the code under test and Pex

[1] `http://msdn.microsoft.com/en-us/library/c5tkafs1(VS.71).aspx`

```
void PUT1(int arg1, int arg2, int arg3) {
    if (arg1 > 0)
        Console.WriteLine(”arg1 > 0”);
    else
        Console.WriteLine(”arg1 <= 0”);
    if (arg2 > 0)
        Console.WriteLine(”arg2 > 0”);
    else
        Console.WriteLine(”arg2 <= 0”);
    for (int c = 1; c <= arg3; c++) {
        Console.WriteLine(”loop”)
    }
}

public void SeedTest1() {
    PUT1(1, 1, 1);
}

void PUT2(int arg1, int arg2, int arg3) {
    if (arg1 > 0)
        Console.WriteLine(”arg1 > 0”);
    else
        Console.WriteLine(”arg1 <= 0”);
    if (arg2 > 0)
        Console.WriteLine(”arg2 > 0”);
    else
        Console.WriteLine(”arg2 <= 0”);
    for (int c = 2; c <= arg3; c++) {
        Console.WriteLine(”loop”)
    }
}

public void SeedTest2() {
    PUT2(1, 10, 1);
}

public void SeedTest3() {
    PUT1(5, 8, 2);
}
```

Figure 6: Two PUTs and associated seed tests generated by the capture phase.

can start exploration from these covered paths rather than starting exploration from the beginning. This would reduce the amount of time required in generating tests from PUTs. Second, seed tests can help cover certain paths that are hard to be covered without using those tests. For example, it is quite challenging for Pex or any other dynamic-symbolic-execution-based approach to generate concrete values for variables that require complex values such as IP addresses, URLs, doubles. In such scenarios, seed tests can help by providing desired concrete values to cover those paths.

Pex generated 86 regression tests for the PUT shown in Figure 5. Figure 7 shows three sample regression tests generated by Pex. In regression tests 1 and 2, Pex automatically annotated the unit tests with the expected exceptions `ArgumentNullException` and `ArgumentOutOfRangeException`, respectively. Since the PUT (Figure 5) includes an `OUT` parameter, Pex generated assertions in regression tests based on actual values captured while generating the test. These expected exceptions or assertions serve as test oracles in regression tests.

Regression Test 1:

```
[PexRaisedException(typeof(ArgumentNullException))]
public static void $F_{102}$() {
    int i = default(int);
    $F_1$ ((string)null, 0, out i);
}
```

Regression Test 2:

```
[PexRaisedException(typeof(ArgumentOutOfRangeException))]
public static void $F_{110}$() {
    int i = default(int);
    $F_1$(””, 1, out i);
}
```

Regression Test 3:

```
public static void $F_{103}$() {
    int i = default(int);
    $F_1$ (”\0\0\0\0\0\0\0<\u013b\0”, 7, out i);
    PexAssert.AreEqual<int>(0, i);
}
```

Figure 7: Regression tests generated by Pex by exploring the PUT shown in Figure 5.

Although Pex is effective in exploring PUTs and generating unit tests, we identify that Pex or any other dynamic-symbolic-execution-based approaches can lead to scalability issues when handling large number of PUTs. To address this issue, our approach uses a distributed setup where Pex can explore multiple PUTs in parallel. Our distributed setup allows to launch multiple Pex processes on several machines. Once started, our distributed setup is designed to run forever in iterations. The primary reason for such a setup is that it is hard to decide when to stop exploring a PUT. For example, loops in the code under test introduce infinite number of possible paths and it will take infinite amount of time to generate tests.

To address the preceding issue, we explore PUTs in iterations bounded by various parameters. For example, consider a timeout parameter that describes when to stop exploring a PUT. In the first iteration, we set three minutes for the timeout parameter. This timeout parameter indicates that we terminate exploration of a PUT after three minutes. In the first iteration, we explore all PUTs with these bounded parameters. In the second iteration, we double the values of these parameters. For example, we set six minutes for the timeout parameter in the second iteration. Doubling the parameters gives more time for Pex in exploring new paths in the code under test. To avoid Pex exploring the same paths that were explored in previous iterations, we maintain a pool of all generated tests. We use the tests in the pool generated by previous iterations as a seed for further iterations. For example, tests generated in Iteration 1 are used as seed tests in Iteration 2.

3. REFERENCES

[1] L. Clarke. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Trans. Softw. Eng.*, 2(3):215–222, 1976.

[2] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Softw. Pract. Exper.*, 34(11):1025–1050, 2004.

[3] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. PLDI*,

pages 213–223, 2005.

[4] P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing. In *Proc. NDSS*, 2008.

[5] K. Inkumsah and T. Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *Proc. ASE*, pages 297–306, 2008.

[6] Parasoft. Jtest manuals version 5.1. Online manual, 2006. `http://www.parasoft.com`.

[7] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proc. TACAS*, pages 553–568, 2003.

[8] J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.

[9] S. Koushik, M. Darko, and A. Gul. CUTE: a concolic unit testing engine for C. In *Proc. ESEC/FSE*, pages 263–272, 2005.

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

[11] D. S. Rosenblum and E. J. Weyuker. Predicting the cost-effectiveness of regression testing strategies. *SIGSOFT Softw. Eng. Notes*, 21(6):118–126, 1996.

[12] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Mseqgen: object-oriented unit-test generation via mining source code. In *Proc. ESEC/FSE*, pages 193–202, 2009.

[13] N. Tillmann and J. de Halleux. Pex white box test generation for .NET. In *Proc. TAP*, pages 134–153, 2008.

[14] N. Tillmann and W. Schulte. Parameterized Unit Tests. In *Proc. ESEC/FSE*, pages 253–262, 2005.

[15] P. Tonella. Evolutionary testing of classes. In *Proc. ISSTA*, pages 119–128, 2004.

[16] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *Proc. ASE*, pages 196–205, 2004.