

Automatic Unit Test Generation via Mining Gigabytes of Dynamic Traces

Suresh Thummalapenta¹, Jonathan de Halleux², Nikolai Tillmann², Scott Wadsworth³

¹Department of Computer Science, North Carolina State University, Raleigh

²Microsoft Research, One Microsoft Way, Redmond

³Microsoft Corporation, One Microsoft Way, Redmond

¹sthumma@ncsu.edu, ²{jhalleux, nikolait}@microsoft.com, ³bwadswor@microsoft.com

ABSTRACT

An important aspect of software maintenance is to ensure that the changes made in the new version of software do not introduce any unwanted side effects. Regression testing is a testing methodology that aims at exposing such unwanted side effects introduced in the new version of software. In this paper, we present a novel scalable approach that automatically generates regression tests without requiring any manual efforts. Our approach generates parameterized unit tests via mining dynamic traces recorded during program executions. Our approach next uses dynamic symbolic execution to generate regression tests that achieve a high coverage of the code under test. In our evaluations, we show that our approach records ≈ 1.5 GB of dynamic traces and generates $\approx 500,000$ regression tests on ten .NET 2.0 framework base class libraries. These numbers show that our approach is scalable and can be used in practice to deal with large real-world applications.

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 [19] 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

```
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.

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.

Recent advancements in software testing introduced Parameterized Unit Tests (PUT) [26], 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 [3, 9, 14, 15], 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 2 provides more details on how dynamic symbolic execution generates conventional unit tests from PUTs. In our approach, we use Pex [25] 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 [13, 31], evolutionary [11, 27], and random [4, 12, 18]. In our previous work [24], 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 [10].

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 ≈ 1.5 GB of dynamic traces and generates $\approx 500,000$ regression tests on ten .NET 2.0 framework base class libraries. These numbers show that our approach is scalable and can be used in practice to deal with large 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 framework base class libraries. From these PUTs, our approach generated 501,799 regression tests

that achieved a high code coverage of the ten .NET framework base class libraries.

The rest of the paper is structured as follows: Section 2 presents background on a DSE-based approach. Section 3 describes key aspects of our approach. Section 4 presents our evaluation results. Section ?? discusses threats to validity. Section ?? discusses limitations of our approach and future work. Section 5 presents related work. Finally, Section ?? concludes.

2. BACKGROUND

We next provide details of two major concepts used in the rest of the paper: dynamic symbolic execution and dynamic code coverage.

2.1 Dynamic Symbolic Execution

In our approach, we use Pex as an example state-of-the-art dynamic symbolic execution tool. Pex [25] is an automatic unit-test-generation tool developed by Microsoft. Pex accepts PUTs as input and generates conventional unit tests that can achieve high structural coverage of the code under test. Initially, Pex executes the code under test with random inputs. While executing the code under test, Pex collects constraints on inputs from predicates in branching statements. Pex next solves collected constraints to generate new test inputs that guide future executions along new paths. Pex includes various optimization techniques such as reducing the size of the formula before giving it over constraint solver.

2.2 Dynamic Code Coverage

In this paper, we use Pex generated reports for measuring coverage. These coverage reports are called dynamic, because Pex knows only about the code that was already executed. As Pex is not aware of the code that is not yet executed, dynamic code coverage cannot give absolute values for the coverage. The primary reason for using dynamic code coverage in our paper is that C# allows generate code during run time. Therefore, it is often not possible to find out how much code exists beforehand.

3. 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.

3.1 Capture Phase

In the capture phase, our approach records dynamic traces from program executions. The capture phase uses a profiler

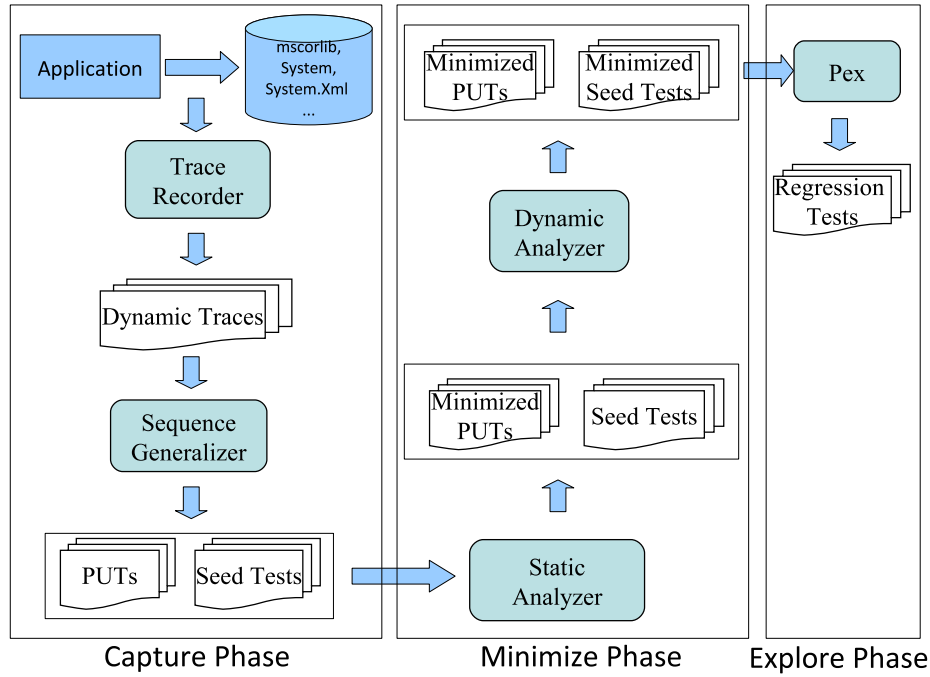


Figure 3: A high-level overview of our approach

```

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.

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

PUT:

```

00: public static void F1(string VAL1, int VAL2, out int OUT1)
01:   TagRegex tagex = new TagRegex();
02:   Match mc = ((Regex)tagex).Match(VAL1, VAL2);
03:   Capture cap = (Capture) mc;
04:   OUT1 = cap.Index;
05: }

```

Seed Test:

```

06: public static void T1() {
07:   int index;
08:   F1("<%@ Page..\u000a", 108, out index);
09: }

```

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

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 3.3). 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.

3.2 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

```

00: void PUT1(int arg1, int arg2, int arg3) {
01:     if (arg1 > 0)
02:         Console.WriteLine("arg1 > 0");
03:     else
04:         Console.WriteLine("arg1 <= 0");
05:     if (arg2 > 0)
06:         Console.WriteLine("arg2 > 0");
07:     else
08:         Console.WriteLine("arg2 <= 0");
09:     for (int c = 1; c <= arg3; c++) {
10:         Console.WriteLine("loop")
11:     }
12: }

13: public void SeedTest1() {
14:     PUT1(1, 1, 1);
15: }

16: void PUT2(int arg1, int arg2, int arg3) {
17:     if (arg1 > 0)
18:         Console.WriteLine("arg1 > 0");
19:     else
20:         Console.WriteLine("arg1 <= 0");
21:     if (arg2 > 0)
22:         Console.WriteLine("arg2 > 0");
23:     else
24:         Console.WriteLine("arg2 <= 0");
25:     for (int c = 2; c <= arg3; c++) {
26:         Console.WriteLine("loop")
27:     }
28: }

29: public void SeedTest2() {
30:     PUT2(1, 10, 1);
31: }

32: public void SeedTest3() {
33:     PUT1(5, 8, 2);
34: }

```

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

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 Language¹ 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

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

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.

3.3 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. [10], 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 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.

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.

Regression Test 1:

```

00: [PexRaisedException(typeof(ArgumentNullException))]
01: public static void F102() {
02:     int i = default(int);
03:     F1((string)null, 0, out i);
04: }

```

Regression Test 2:

```

00: [PexRaisedException(typeof(ArgumentOutOfRangeException))]
01: public static void F110() {
02:     int i = default(int);
03:     F1("", 1, out i);
04: }

```

Regression Test 3:

```

00: public static void F103() {
01:     int i = default(int);
02:     F1("0\0\0\0\0\0\0<\u013b\0", 7, out i);
03:     PexAssert.AreEqual<int>(0, i);
04: }

```

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

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. Based on the amount of time available for generating tests, tests can be generated in further iterations.

4. EVALUATIONS

We conducted three evaluations to show the effectiveness of our approach in generating regression tests that achieve a high coverage of the code under test. Our empirical results show that our approach is scalable and can automatically generate tests for large real-world applications without any manual efforts. In our evaluations, we use ten .NET 2.0 framework base class libraries as subject applications. We next describe the research questions addressed in our evaluation and present our evaluation results.

4.1 Research Questions

We address the following three research questions in our evaluations.

- RQ1: Can our approach handle large real-world applications in automatically generating regression tests that achieve a high coverage of the code under test?
- RQ2: Do seed tests help achieve higher coverage of the code under test than without using seed tests?

.NET libraries	LOC	# public classes	# public methods
mscorlib	185K	1440	17800
System			
System.Windows.Forms			
System.Drawing			
System.Xml	150K	686	9920
System.Web.RegularExpressions			
System.Configuration			
System.Data	196K	648	11550
System.Web			
System.Transactions			
TOTAL			

Table 1: Ten .NET framework base class libraries used in our evaluations

Machine Configuration	# of machines	# of processes
Xeon 2 CPU @ 2.50 GHz, 8 cores 16 GB RAM	1	7
Quad core 2 CPU @ 1.90 GHz, 8 cores 8 GB RAM	2	7
Intel Xeon CPU @2.40 GHz, 2 cores 1 GB RAM	6	1

Table 2: Three categories of machine configurations used in our evaluations.

- RQ3: Can more machine power help generate new regression tests that can achieve more coverage of the code under test?

4.2 Subject Applications

We used ten .NET 2.0 framework base class libraries² as subject applications in our evaluations. We selected these libraries for two main reasons. First, these libraries are the core libraries of .NET framework and are used by thousands of developers and millions of end users. Therefore, these libraries can represent real-world applications. Second, these libraries are extensively tested over several years and are suitable for generating regression tests. Table 1 shows the ten libraries used in our evaluations and their characteristics such as the number of classes and methods. The table shows that these libraries include 0 LOC with 0 classes and 0 methods.

4.3 Evaluation Setup

In our approach, we used nine machines that can be classified into three configuration categories. On each machine, we launched multiple Pex processes. The number of processes launched on a machine is based on the configuration of the machine. For example, on an eight core machine, we launched seven Pex processes. Each Pex process was exploring one class (including multiple PUTs) at a time. Table 2 shows all three configuration categories. The table also shows the number of machines of each configuration and the number of Pex processes launched on each machine.

As we used .NET framework base class libraries in our evaluations, the generated tests may invoke method calls that can result in the corruption of the machine. Therefore, while executing the code during exploration of PUTs or while running generated tests, we created a sand-box with

²<http://msdn.microsoft.com/en-us/library/ms229335.aspx>

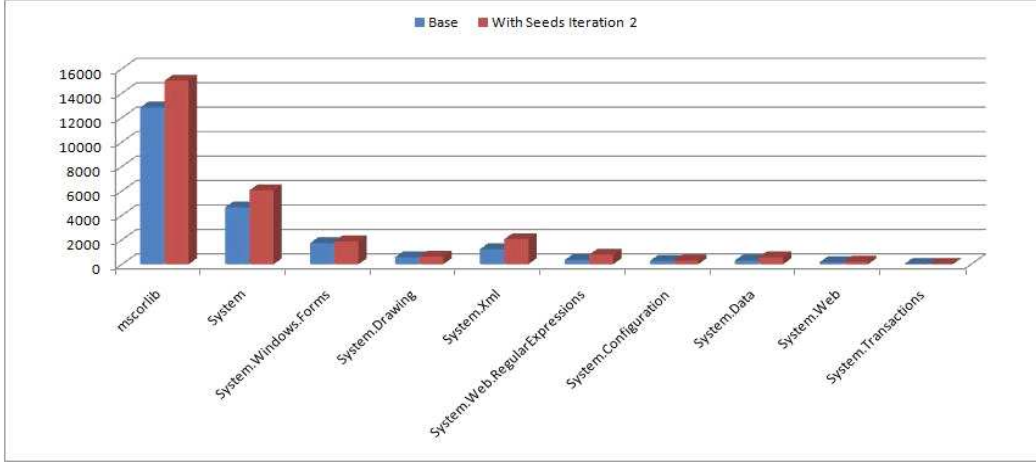


Figure 8: Comparison of base coverage and coverage achieved by regression tests of Mode 4 (WithSeeds Iteration 2).

the “Internet” security permission. This permission represents the default policy permission set for the content from an unknown origin. This permission blocks all operations that involve environment interactions such as file creations or registry accesses by throwing `SecurityException`. Since we use sand-box in our evaluations, the reported coverage is lower than the actual coverage that can be achieved by our generated regression tests.

To address our research questions, we first created a base line in terms of the code coverage achieved by the seed tests, referred to as *base coverage*. In our evaluations, we use block coverage [1] as a coverage criteria. We report our coverage in terms of the number of blocks covered in the code under test. The reason for not able to present the coverage as a percentage of the number of blocks to the total number of blocks is that we do not know how many blocks in the code under test are actually reachable.

We next generated regression tests in four different modes. In Mode 1 (referred to as *WithoutSeeds Iteration 1*), we generated regression tests without using seed tests for one iteration. In Mode 2 (referred to as *WithoutSeeds Iteration 2*), we generated regression tests without using seed tests for two iterations. The regression tests generated in Mode 2 are a super set of the regression tests generated in Mode 1. In Mode 3 (referred to as *WithSeeds Iteration 1*), we generated regression tests with using seed tests for one iteration. Finally, in Mode 4 (referred to as *WithSeeds Iteration 2*), we generated regression tests with using seed tests for two iterations. Modes 1 and 3 took one and half day for generating tests, whereas Modes 2 and 4 took three days since these modes correspond to Iteration 2.

4.4 RQ1: Generated Regression Tests

We next address the first research question of whether our approach can handle large real-world applications in automatically generating regression tests. This research question helps to show that our approach can be used in practice and can address scalability issues in generating regression tests for large applications. We first present the statistics after each phase in our approach and next present the number of regression tests generated in each mode.

In the capture phase, our approach recorded ≈ 1.50 GB

C# source code (including 433,809 traces) of dynamic traces for the ten libraries. The average trace length includes 21 method calls and the maximum trace length includes 52 method calls. As our capture phase transforms each dynamic trace into a PUT and a seed test, the capture phase resulted in 433,809 PUTs and 433,809 seed tests.

In the minimize phase, our approach uses static analysis to filter out duplicate PUTs. Our static analysis took 45 minutes and resulted in 68,575 unique PUTs. Our approach uses dynamic analysis to filter our duplicate seed tests. Our dynamic analysis took 5 hours and resulted in 128,185 unique seed tests. These results show that there are a large number of duplicate PUTs and seed tests, and show the significance of our minimize phase. We next measured the block coverage achieved by these 128,185 unique seed tests in the code under test and used this coverage as *base coverage*. These tests covered 22,111 blocks in the code under test.

Table 3 shows the number of regression tests generated in each mode along with the number of covered blocks. The table also shows the percentage of increase in the number of blocks compared to the base coverage. As shown in results, in Mode 4 (WithSeeds Iteration 2), our approach achieved 24.3% higher coverage than the base coverage. Figure 8 shows detailed comparison of coverages between base coverage and Mode 4 for each library. In the figure, x-axis represent the library and y-axis represent the number of covered blocks. As we use seed tests during our exploration, the coverage achieved is either the same or higher than the base coverage. However, our approach has achieved significant higher coverages than base coverage for libraries `mscorlib` and `System`. The primary reason is that most of the classes in these libraries are stateless and do not require environment interactions. The results show that our approach can handle large real-world applications and can generate large number of regression tests that achieve a high coverage of the code under test.

4.5 RQ2: Using Seed Tests

We next address the second research question of whether seed tests help achieve higher code coverage compared to without using seed tests. To address this question, we compare the coverages achieved by generated tests in Modes 2

Mode	# of Tests	# of covered blocks	% of increase from base
WithoutSeeds Iteration 1	248,306	21,920	0%
WithoutSeeds Iteration 2	412,928	23,176	4.8%
WithSeeds Iteration 1	376,367	26,939	21.8%
WithSeeds Iteration 2	501,799	27,485	24.3%

Table 3: Generated regression tests.

(WithoutSeeds Iteration 2) and 4 (WithSeeds Iteration 2) along with the base coverage. Figure 9 shows the comparison results for each library. As shown in the figure, Mode 4 has always achieved higher coverage than Mode 2. On average, Mode 4 achieved 18.6% higher coverage than Mode 2. These results show that seed tests are helpful in achieving higher code coverages.

4.6 RQ3: Using More Machine Power

We next address the third research question of whether more machine power helps to achieve more coverage. This research question helps to show that additional coverage can be achieved in further iterations of our approach. To address this question, we compare coverages achieved in Mode 1 (WithoutSeeds Iteration 1) with Mode 2 (WithoutSeeds Iteration 2), and Mode 3 (WithSeeds Iteration 2) with Mode 4 (WithSeeds Iteration 2).

Figure 10 shows the comparison results of Mode 1 with Mode 2 for all libraries. On average, Mode 2 achieved 5.73% higher coverage than Mode 1. This result shows that our approach can achieve additional coverage in further iterations. However, the coverage from Mode 1 to Mode 2 is not doubled. The primary reason is that it gets harder to cover new blocks in further iterations.

Figure 11 shows the comparison results of Mode 3 with Mode 4. On average, Mode 4 achieved 2.0% higher coverage than Mode 3. As shown, the increase in coverage from Mode 3 to Mode 4 is less than the increase in the coverage from Mode 1 to Mode 2. This difference is due to seed tests that help achieve higher coverage during Mode 3, leaving more harder blocks to be covered in Mode 4. In summary, the results show that further iterations can help generate new regression tests that can achieve more coverage.

4.7 Real Defects

5. RELATED WORK

Our approach is closely related to two major research areas: regression testing and method-call sequence generation.

Regression testing. There exist approaches [6, 16, 20] that use a capture-and-replay strategy for generating regression tests. In the capture phase, these approaches monitor the methods called during execution and uses these method calls in the replay phase to generate unit tests. Our approach also uses a strategy similar to the capture-and-replay strategy, where we capture dynamic traces during program execution and use those traces for generating regression tests. However, unlike existing approaches that replay exactly the same captured behavior, our approach replays beyond the captured behavior by using dynamic symbolic execution in generating new regression tests.

Another existing approach, called Orstra [30], augments an existing test suite with additional assertions to detect regression faults. To add these additional assertions, Orstra

executes given test suite and collects the return values and receiver object states after the execution of methods under test. Orstra generates additional assertions based on the collected return values or receiver object states. Our approach also uses a similar strategy for generating assertions in the regression tests. However, unlike Orstra that requires an existing test suite as input, our approach does not require any inputs.

Another category of existing approaches [5, 8, 21] in regression testing primarily target at using regression tests for effectively exposing the behavioral differences between two versions of a software. For example, these approaches target at selecting those regression tests that are relevant to portions of source code changed between the two versions of software. However, all these approaches require an existing regression test suite, which is the primary objective of our current approach.

Method-call sequence generation. To test object-oriented programs, existing test-generation approaches [4, 12, 17, 31] accept a class under test and generate sequences of method calls randomly. These approaches generate random values for arguments of those method calls. Another set of approaches [11] replaces concrete values for method arguments with symbolic values and exploits dynamic symbolic execution techniques [3, 9, 14, 15] to regenerate concrete values based on branching conditions in the method under test. However, all these approaches cannot handle multiple classes and their methods due to a large search space of possible sequences.

Randoop [18] is a random testing approach that uses an incremental approach for constructing method-call sequences. Randoop randomly selects a method call and finds arguments required for these method calls. Randoop uses previously constructed method-call sequences to generate arguments for the newly selected method call. Unlike a pure random approach, Randoop incorporates feedback obtained from previously constructed method-call sequences while generating new test inputs. As soon as a method-call sequence is constructed, Randoop executes the sequence and verifies whether the sequence violates any contracts and filters. Randoop cannot effectively generate sequences that achieve high coverage of the code under test as Randoop still relies on random techniques [24].

Our approach is also related to another category of approaches based on mining source code [1, 7, 22–24, 29]. These approaches statically analyze code bases and uses mining algorithms such as frequent itemset mining [28] or association rule mining [2] for extracting frequent patterns. These frequent patterns are treated as programming rules in either assisting programmers while writing code or for detecting violations as deviations from these patterns. Unlike these existing approaches, our approach mines dynamic traces recorded during program executions and uses those traces for generating regression tests. Our previous work [24] also mines method-call sequences from existing code bases. Our previous work uses these method-call sequences to assist random or DSE-based approaches. Our new approach is significantly different from our previous work in two major aspects. First, our new approach is a complete approach for automatically generating regression tests from dynamic traces, whereas, our previous work mines method-call sequences to assist random or DSE-based approaches. Second, our new approach uses dynamic traces, which are more precise compared to

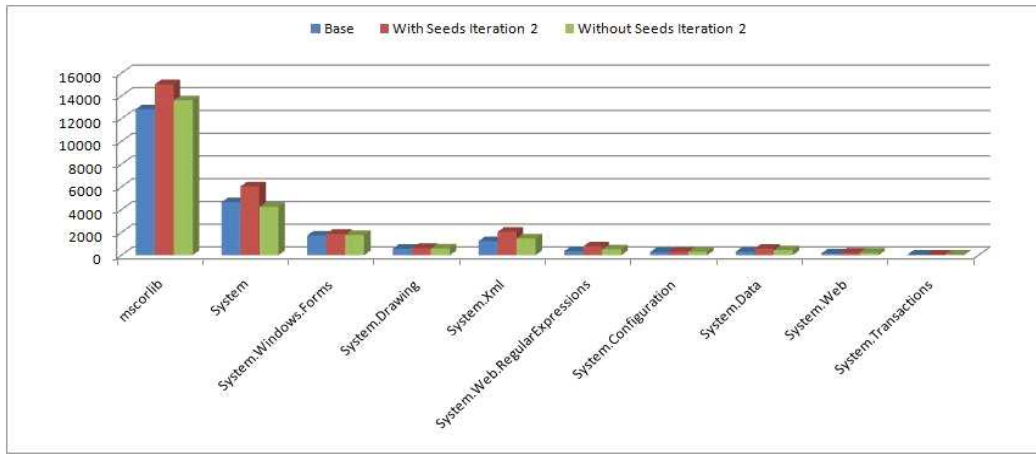


Figure 9: Comparison of code coverages achieved by base, Modes 2 (WithoutSeeds Iteration 2) and 4 (With-seeds Iteration 2).

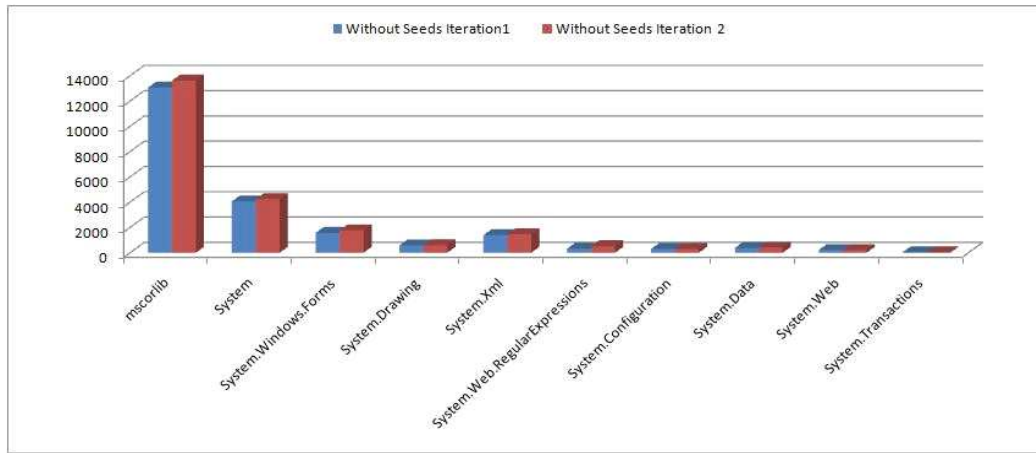


Figure 10: Comparison of code coverages achieved by Modes 1 (WithoutSeeds Iteration 1) and 2 (Without-Seeds Iteration 2).

the static traces used in our previous work.

6. CONCLUSION

Regression testing is an important and yet challenging problem in the field of software maintenance. The quality of regression tests is based on the test scenarios (in the form of method-call sequences) used in the tests. Unlike existing approaches that generate test scenarios randomly, in this paper, we proposed a novel scalable approach that generates test scenarios via mining dynamic traces recorded during program execution. In our approach, we transform these generated test scenarios into PUTs and use dynamic symbolic execution to generate regression tests that achieve a high coverage of the code under test. In our evaluations, we show that our approach recorded ≈ 1.5 GB of dynamic traces and generated $\approx 500,000$ regression tests on ten .NET 2.0 framework base class libraries. These numbers show that our approach is highly scalable and can be used in practice to deal with large real-world applications. In future work, we plan to generate new test scenarios via combining generated test scenarios using evolutionary strategies.

7. REFERENCES

- [1] M. Acharya, T. Xie, and J. Xu. Mining Interface Specifications for Generating Checkable Robustness Properties. In *Proc. ISSRE*, pages 311–320, 2006.
- [2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proc. VLDB*, pages 487–499, 1994.
- [3] L. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Softw. Eng.*, 2(3):215–222, 1976.
- [4] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Softw. Pract. Exper.*, 34(11):1025–1050, 2004.
- [5] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Trans. Softw. Eng.*, 17(9):900–910, 1991.
- [6] S. Elbaum, H. N. Chin, M. B. Dwyer, and J. Dokulil. Carving differential unit test cases from system test cases. In *Proc. FSE*, pages 253–264, 2006.
- [7] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *Proc.*

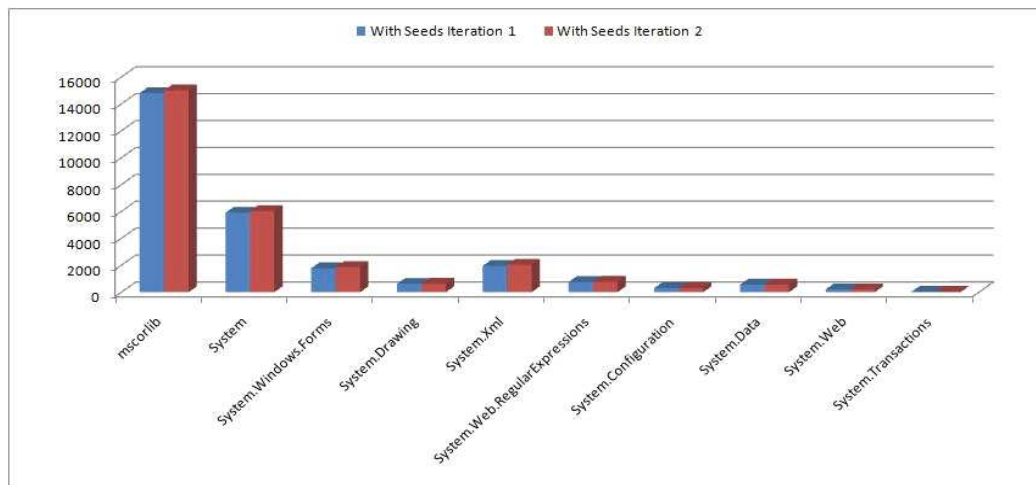


Figure 11: Comparison of code coverages achieved by Modes 3 (WithSeeds Iteration 1) and 4 (Withseeds Iteration 2).

- SOSP*, pages 57–72, 2001.
- [8] R. B. Evans and A. Savoia. Differential testing: a new approach to change detection. In *Proc. ESEC/FSE*, pages 549–552, 2007.
 - [9] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. PLDI*, pages 213–223, 2005.
 - [10] P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing. In *Proc. NDSS*, 2008.
 - [11] 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.
 - [12] Parasoft. Jtest manuals version 5.1. Online manual, 2006. <http://www.parasoft.com>.
 - [13] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proc. TACAS*, pages 553–568, 2003.
 - [14] J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.
 - [15] S. Koushik, M. Darko, and A. Gul. CUTE: a concolic unit testing engine for C. In *Proc. ESEC/FSE*, pages 263–272, 2005.
 - [16] A. Orso and B. Kennedy. Selective capture and replay of program executions. *SIGSOFT Softw. Eng. Notes*, 30(4):1–7, 2005.
 - [17] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *Proc. ECOOP*, pages 504–527, 2005.
 - [18] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proc. ICSE*, pages 75–84, 2007.
 - [19] D. S. Rosenblum and E. J. Weyuker. Predicting the cost-effectiveness of regression testing strategies. *SIGSOFT Softw. Eng. Notes*, 21(6):118–126, 1996.
 - [20] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automatic test factoring for Java. In *Proc. ASE*, pages 114–123, 2005.
 - [21] K. Taneja and T. Xie. DiffGen: Automated regression unit-test generation. In *Proc. ASE*, pages 407–410, 2008.
 - [22] S. Thummalapenta and T. Xie. PARSEWeb: A programmer assistant for reusing open source code on the web. In *Proc. ASE*, pages 204–213, 2007.
 - [23] S. Thummalapenta and T. Xie. Mining exception-handling rules as sequence association rules. In *Proc. ICSE*, pages 496–506, 2009.
 - [24] 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.
 - [25] N. Tillmann and J. de Halleux. Pex white box test generation for .NET. In *Proc. TAP*, pages 134–153, 2008.
 - [26] N. Tillmann and W. Schulte. Parameterized Unit Tests. In *Proc. ESEC/FSE*, pages 253–262, 2005.
 - [27] P. Tonella. Evolutionary testing of classes. In *Proc. ISSA*, pages 119–128, 2004.
 - [28] J. Wang and J. Han. BIDE: Efficient mining of frequent closed sequences. In *Proc. ICDE*, pages 79 – 88, 2004.
 - [29] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *Proc. ESEC/FSE*, pages 35–44, 2007.
 - [30] T. Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *Proc. ECOOP*, pages 380–403, 2006.
 - [31] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *Proc. ASE*, pages 196–205, 2004.