DyGen: Automatic Generation of High-Coverage 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, NC
[2] Microsoft Research, One Microsoft Way, Redmond, WA
[3] Microsoft Corporation, One Microsoft Way, Redmond, WA
[1]{sthumma}@ncsu.edu, [2]{jhalleux, nikolait}@microsoft.com,
[3]bwadswor@microsoft.com

Abstract. Unit tests of object-oriented code exercise particular sequences of method calls. A key problem when automatically generating unit tests that achieve high structural code coverage is the selection of relevant method-call sequences, since the number of potentially relevant sequences explodes with the number of methods. To address this issue, we propose a novel approach, called *DyGen*, that generates tests via mining dynamic traces recorded during program executions. Typical program executions tend to exercise only *happy* paths that do not include error-handling code, and thus recorded traces often do not achieve high structural coverage. To increase coverage, DyGen transforms traces into parameterized unit tests (PUTs) and uses dynamic symbolic execution to generate new unit tests for the PUTs that can achieve high structural code coverage. In this paper, we show an application of DyGen by automatically generating regression tests on a given version of software. In our evaluations, we show that DyGen records ≈1.5 GB (size of corresponding C# source code) of dynamic traces and generates ≈500,000 regression tests, where each test exercises a unique path, on two core libraries of .NET 2.0 framework. The generated regression tests covered 27,485 basic blocks, which are 24.3% higher than the number of blocks covered by recorded dynamic traces.[4]

Key words: object-oriented unit testing, regression testing, dynamic symbolic execution

1 Introduction

Software testing is a common methodology used to detect defects in the code under test. A major objective of unit testing is to achieve high structural coverage of the code under test, since unit tests can only uncover defects in those portions of the code, which are executed by those tests. Automatic generation of unit tests that achieve high structural coverage of object-oriented code requires method-call sequences (in short as *sequences*). These sequences help cover `true` or `false` branches in a method under test by creating desired object states for its receiver or arguments. We next present an example for desired object state and explain how method-call sequences help achieve desired object states using an illustrative example shown in Figure 1a.

[4] The majority of the work was done during an internship at Microsoft Research.

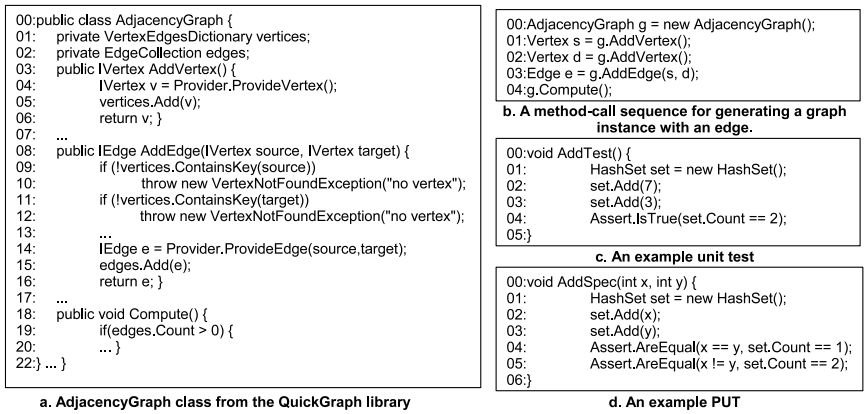


Fig. 1. Sample code examples

Figure 1a shows an `AdjacencyGraph` class from the QuickGraph[5] library. The graph includes vertices and edges that can be added using the methods `AddVertex` and `AddEdge`, respectively. To reach Statement 20 in the `Compute` method, a desired object state is that the graph object should include at least one edge. Figure 1b shows a sequence that generates the desired object state. It is quite challenging to generate these sequences automatically from the implementation of `AdjacencyGraph` due to a large number of possible sequences and only a few sequences are valid. In practice, sequences required for generating desired object states often include multiple classes leading to a large space of possible sequences that cannot be effectively handled by existing approaches [1][2][3][4][5][6][7] that are either random or based on class implementations.

To address preceding issues, we propose a novel approach, called *DyGen*, that generates sequences from dynamic traces recorded during (typical) program executions. We use *dynamic* traces as opposed to *static* traces, since dynamic traces are more precise than static traces. These recorded dynamic traces include two aspects: realistic scenarios expressed as sequences and concrete values passed as arguments to those method calls. Since dynamic traces include both sequences and concrete argument values, these traces can directly be transformed into unit tests. However, such a naive transformation results in a large number of *redundant* unit tests that often do not achieve high structural coverage due to *two* major issues. We next explain these two major issues of naive transformation and describe how DyGen addresses those issues.

First, since dynamic traces are recorded during program executions, we identify that many of the recorded traces are duplicates. The reason for duplicates is that the same sequence can get invoked multiple times. Therefore, a naive transformation results in a large number of redundant unit tests. To address this issue, DyGen uses a combination of static and dynamic analyses and filters out duplicate traces.

Second, unit tests generated with the naive transformation tend to exercise only *happy paths* (such as paths that do not include error-handling code in the code under test) and often do not achieve high structural coverage of the code under test. To address this issue, DyGen transforms recorded dynamic traces into Parameterized Unit

[5] `http://www.codeplex.com/quickgraph`

Tests (PUT) [8] rather than Conventional Unit Tests (CUT). PUTs are a recent advance in software testing and generalize CUTs by accepting parameters. Figure 1d shows a PUT for the CUT shown in Figure 1c, where concrete values in Statements 2 and 3 are replaced by the parameters `x` and `y`. DyGen uses Dynamic Symbolic Execution (DSE) [9][10][11][12] to automatically generate a small set of CUTs that achieve high coverage of the code under test defined by the PUT. Section 2 provides more details on how DSE generates CUTs from PUTs. DyGen uses Pex [13], a DSE-based approach for generating CUTs from PUTs. However, DyGen is not specific to Pex and can be used with any other test-input generation engine.

DyGen addresses *two* major challenges faced by existing DSE-based approaches in effectively generating CUTs from PUTs. First, DSE-based approaches face a challenge in generating concrete values for parameters that require complex values such as floating point values or URLs. To address this challenge, DyGen uses naive transformation on each trace to generate a CUT, which is effectively an instantiation of the corresponding PUT. DyGen uses this CUT to seed the exploration of the corresponding PUT, which DyGen generates as well. Using seed tests helps not only to address the preceding challenge in generating complex concrete values, but also helps in increasing the efficiency of DSE while exploring PUTs. Second, in our evaluations (and also in practice), we identify that even after minimization of duplicate traces, the number of generated PUTs and seed tests can still be large, and it would take a long time (days or months) to explore those PUTs with DSE on a single machine. To address this challenge, DyGen uses a distributed setup that allows parallel exploration of PUTs.

In this paper, we show an application of DyGen by automatically generating regression tests on a given version of software. Regression testing, an important aspect of software maintenance, helps ensure that changes made in new versions of software do not introduce any new defects, referred to as *regression defects*, relative to the baseline functionality. Rosenblum and Weyuker [14] describe that the majority of software maintenance costs is spent on regression testing. To transform generated CUTs into regression tests, DyGen infers test assertions based on the given version of software. More specifically, DyGen executes generated CUTs on the given version of software, captures the return values of method calls, and generates test assertions from these captured return values. These test assertions help detect regression defects by checking whether the new version of software also returns the same values.

In summary, this paper makes the following major contributions:

- A scalable approach for automatically generating regression tests (that achieve high structural coverage of the code under test) via mining dynamic traces from program executions and without requiring any manual efforts.
- A technique to filter out duplicate dynamic traces by using static and dynamic analyses, respectively.
- A distributed setup to address scalability issues via parallel exploration of PUTs to generate CUTs.
- Three large-scale evaluations to show the effectiveness of our DyGen approach. In our evaluations, we show that DyGen recorded $\approx$1.5 GB C# source code (including 433,809 traces) of dynamic traces from applications using two core libraries of the .NET framework. From these PUTs, DyGen eventually generated 501,799 regression tests, where each test exercises a unique path, that together covered 27,485

basic blocks, which represents an increase of 24.3% over the number of blocks covered by the originally recorded dynamic traces.

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 5 discusses limitations of our approach and future work. Section 6 presents related work. Finally, Section 7 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

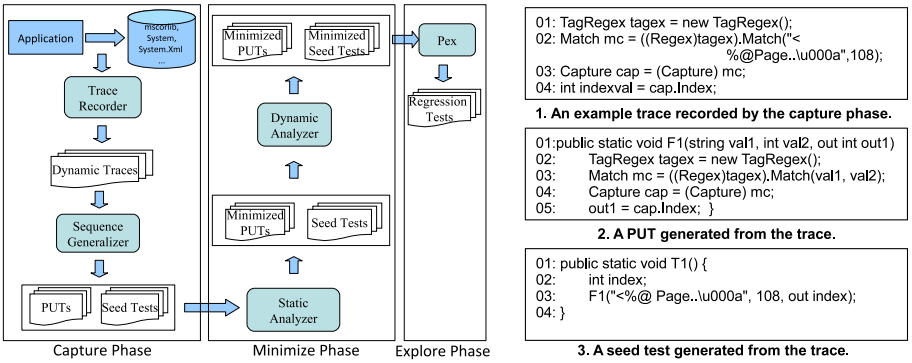
In our approach, we use Pex as an example state-of-the-art dynamic symbolic execution tool. Pex [13] is an automatic unit-test-generation tool developed by Microsoft Research. Pex accepts PUTs as input and generates CUTs that achieve high coverage of the code under test. Initially, Pex executes the code under test with arbitrary inputs. While executing the code under test, Pex collects constraints on inputs from predicates in branching statements along the exercised execution path. Pex next solves collected constraints to generate new inputs that guide future executions along new paths. Pex uses a constraint solver and theorem prover, called Z3 [15], to reason about collected constraints by faithfully encoding all constraints that arise in safe .NET programs. Z3 uses decision procedures for propositional logic, fixed sized bit-vectors, tuples, arrays, and quantifiers to reason about encoded constraints. Z3 approximates arithmetic constraints over floating point numbers by translating them to rational numbers. Pex also implements various optimization techniques to reduce the size of the formula that is given to Z3.

2.2 Dynamic Code Coverage

In this paper, we present dynamic code coverage information collected by Pex. As Pex performs code instrumentation dynamically at runtime, Pex only knows about the code that was already executed. In addition to code loaded from binaries on the disk, the .NET environment in which we perform our experiments allows the generation of additional code at runtime via *Reflection-Emit*.

3 Approach

Figure 2 shows the high-level overview of our DyGen approach. DyGen includes three major phases: *capture*, *minimize*, and *explore*. In the capture phase, DyGen records dynamic traces from (typical) program executions. DyGen next transforms these dynamic traces into PUTs and seed tests. Among recorded traces, we identify that there are 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. For example, in our evaluations, we found that 84% of PUTs and 70% of seed tests are classified as duplicates by our minimize phase. To address this issue,



(a) (b)

Fig. 2. (a) A high-level overview of DyGen. (b) A dynamic trace and generated PUT and CUT from the trace.

in the minimize phase, DyGen uses a combination of static and dynamic analyses to filter out duplicate PUTs and seed tests, respectively. In the explore phase, DyGen uses Pex to explore PUTs to generate regression tests that achieve high coverage of the code under test.

3.1 Capture Phase

In the capture phase, DyGen records dynamic traces from program executions. The capture phase uses a profiler that records method calls invoked by the program during execution. The capture phase records both the method calls invoked and the concrete values passed as arguments to those method calls. Figure 2b1 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.

DyGen uses a technique similar to Saff et al. [16] for transforming recorded traces into PUTs and seed tests. To generate PUTs, DyGen identifies all constant values and promotes those constant values as parameters. Furthermore, DyGen identifies return values of method calls 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. DyGen next generates seed tests that include all concrete values from the dynamic traces. Figures 2b2 and 2b3 show the PUT and the seed test, respectively, generated from the dynamic trace shown in Figure 2b1.

The generated PUT includes two parameters and one `out` parameter. The `out` parameter is the return value of the 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

In the minimize phase, DyGen filters out duplicate PUTs and seed tests. The primary reason for filtering out duplicates is that exploration of duplicate PUTs or execution of duplicate seed tests is redundant and can also lead to scalability issues while generating regression tests. We use PUTs and seed tests shown in Figure 3 as illustrative examples

```
00:Class A {
01:     public void foo(int arg1, int arg2, int arg3) {
02:             if (arg1 > 0)
03:                 Console.WriteLine("arg1 > 0");
04:             else
05:                 Console.WriteLine("arg1 <= 0");
06:             if (arg2 > 0)
07:                 Console.WriteLine("arg2 > 0");
08:             else
09:                 Console.WriteLine("arg2 <= 0");
10:             for (int c = 1; c <= arg3; c++)  {
11:                     Console.WriteLine("loop");
12:             }
13:     }
14:}
```

```
00:void PUT1(int arg1, int arg2, int arg3) {
01:     A a = new A();
02:     a.foo(arg1, arg2, arg3); }

03:public void SeedTest1() {
04:     PUT1(1, 1, 1); }

05:void PUT2(int arg1, int arg2, int arg3) {
06:     A a = new A();
07:     a.foo(arg1, arg2, arg3); }

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

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

Fig. 3. Two PUTs and associated seed tests generated by the capture phase.

to explain the minimize phase. The figure shows a method under test `foo`, two PUTs, and three seed tests. We use these examples primarily for explaining our minimize phase. Our actual PUTs are much more complex than these illustrative examples with an average PUT size of 21 method calls (Section 4.4). 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 Microsoft Intermediate Language (MSIL)[6] instructions.

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. This execution path refers to the path that starts from beginning of the PUT that is called by the seed test, and goes through all (transitive) method calls performed by the PUT.

DyGen uses static analysis to identify duplicate PUTs. Consider the method bodies of `PUT1` and `PUT2`. DyGen considers `PUT2` as a duplicate of `PUT1`, since both the PUTs include the same sequence of MSIL instructions. Since `PUT2` is a duplicate of `PUT1`, DyGen automatically replaces the `PUT2` method call in `SeedTest2` with `PUT1`.

After eliminating duplicate PUTs, DyGen uses dynamic analysis for filtering out duplicate seed tests. To identify duplicate seed tests, DyGen executes each seed test and monitors its execution path in the code under test. For example, `SeedTest1` follows the path "$3 \rightarrow 7 \rightarrow 11$" in the `foo` method. DyGen considers `SeedTest2` as a duplicate of `SeedTest1`, since `SeedTest2` also follows the same path "$3 \rightarrow 7 \rightarrow 11$" in the `foo` method. Consider another unit test `SeedTest3` shown in Figure 3. DyGen does not consider `SeedTest3` as a duplicate of `SeedTest1`, since `SeedTest3` follows the path "$3 \rightarrow 7 \rightarrow 11 \rightarrow 11$" (since `SeedTest3` iterates the loop in Statement 10 two times).

3.3 Explore Phase

In the explore phase, DyGen uses Pex to generate regression tests from PUTs. Although seed tests generated in the capture phase can be considered as regression tests, most seed tests tend to exercise common happy paths such as paths that do not include error-handling code in the code under test. In only a few rare scenarios, seed tests may exercise the paths related to error-handling code, if such scenarios happen during the recorded program executions. Therefore, these seed tests do not achieve high coverage of the corner cases and error handling of the code under test.

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

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

a. Regression Test 1

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

b. Regression Test 2

```
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: }
```

c. Regression Test 3

Fig. 4. Regression tests generated by Pex by exploring the PUT shown in Figure 2b2.

To address this issue, DyGen uses Pex to explore generated PUTs. Inspired by Patrice et al. [17], Pex can leverage seed inputs in the form of conventional unit tests. Using seed tests increases the effectiveness of Pex, and potentially any other DSE-based approaches, in two major ways. First, with seed tests, Pex executes those seed tests and internally builds an execution tree with nodes for all conditional control-flow statements executed along the paths exercised by the seed tests. Pex starts exploration from this pre-populated tree. In each subsequent iteration of the exploration, Pex tries to extend this tree as follows: a formula is constructed that represents the conjunction of the branch conditions of an already known path prefix, conjoined with the negation of a branch condition of a known suffix; the definitions of all derived values are expanded so that conditions only refer to the test inputs as variables. If the formula is satisfiable, and test inputs can be computed by the constraint solver, then by executing the PUT with those test inputs, Pex learns a new feasible path and extends the execution trees with nodes for the suffix of the new path. Without any seed tests, Pex starts exploration with an empty execution tree, and all nodes are discovered incrementally. Therefore, using seed tests significantly reduces the amount of time required in generating a variety of tests with potentially deep execution paths from PUTs. Second, seed tests can help cover reach certain paths that are hard to be covered without using those tests. For example, it is quite challenging for Pex or any other DSE-based approach to generate concrete values for variables that require complex values such as IP addresses, URLs, or floating point values. In such scenarios, seed tests can help provide desired concrete values to reach those paths.

Pex generated 86 regression tests for the PUT shown in Figure 2b2. Figure 4 shows three sample regression tests generated by Pex. In Regression tests 1 and 2, Pex automatically annotated the unit tests with expected exceptions `ArgumentNullException` and `ArgumentOutOfRangeException`, respectively. Since the PUT (Figure 2b2) includes an `out` parameter, Pex generated assertions in regression tests (such as Statement 3 in Regression test 3) based on actual values captured while generating the test. These expected exceptions or assertions serve as test oracles in regression tests.

When a PUT invokes code containing loops, an exhaustive exploration of all execution paths via DSE may not terminate. While Pex employs search strategies to achieve high code coverage quickly even in the presence of loops, Pex or any other DSE-based approaches may still take a long time (days or months) to explore PUTs with DSE on a single machine. To address this issue, DyGen uses an enhanced distributed setup originally proposed in our previous work [13]. Our distributed setup allows to launch multiple Pex processes on several machines. Once started, our distributed setup is designed to run forever in iterations. Each subsequent iterations increase bounds imposed

.NET libraries	Short name	KLOC	# public classes	# public methods
mscorlib	mscorlib	178	1316	13199
System	System	149	947	8458
System.Windows.Forms	Forms	226	1403	17785
System.Drawing	Drawing	24	223	2823
System.Xml	Xml	122	270	5426
System.Web.RegularExpressions	RegEx	10	16	162
System.Configuration	Config	17	105	773
System.Data	Data	126	298	5464
System.Web	Web	202	1140	11487
System.Transactions	Trans	9.5	39	405
TOTAL		1063	5757	65982

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

on the exploration to guarantee termination. For example, consider the *timeout* parameter that describes when to stop exploring a PUT. In the first iteration, DyGen sets three minutes for the timeout parameter. This value indicates that DyGen terminates exploration of a PUT after three minutes. In the first iteration, DyGen explores all PUTs with these bounded parameters. In the second iteration, DyGen doubles the values of these parameters. For example, DyGen sets 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, DyGen maintains a pool of all generated tests. DyGen uses the tests in the pool generated by previous iterations as seed tests 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 DyGen in generating regression tests that achieve high coverage of the code under test. Our empirical results show that DyGen is scalable and can automatically generate regression tests for large real-world code bases without any manual efforts. In our evaluations, we use two core .NET 2.0 framework libraries[7] as main subjects. 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 DyGen handle large real-world code bases in automatically generating regression tests that achieve 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?
- RQ3: Can more machine power help generate new regression tests that can achieve more coverage of the code under test?

4.2 Subject Code Bases

We used two core .NET 2.0 framework base class libraries as the main subjects in our evaluations. Since these libraries sit at the core of the .NET framework, it is paramount

[7] `http://msdn.microsoft.com/en-us/library/ms229335.aspx`

Machine Configuration	# of mc	# of pr
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

(a)

Mode	# of Tests	# of blocks	% of incr 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%

(b)

Fig. 5. (a) Three categories of machine configurations used in our evaluations. (b) Generated regression tests.

for the .NET product group to maintain and continually enrich a comprehensive regression test suite, in order to ensure that future product versions preserve the existing behavior, and to detect breaking changes. Table 1 shows the two libraries (mscorlib and System) used in our evaluations and their characteristics such as the number of classes and methods. Column "Short name" shows short names (for each library) that are used to refer to those libraries. The table also shows statistics of eight other libraries of .NET 2.0 framework. Although these other eight libraries are not our primary targets for generating regression tests, they were exercised as well by the recorded program executions. In our evaluations, we use these additional eight libraries also while presenting our coverage results. The table shows that these libraries include 1,063 KLOC with 5,757 classes and 65,982 methods.

4.3 Evaluation Setup

In our evaluations, 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 5(a) shows all three configuration categories. Columns "# of mc" and "# of pr" show the number of machines of each configuration and the number of Pex processes launched on each machine, respectively.

Since we used .NET framework base class libraries in our evaluations, the generated tests may invoke method calls that can cause external side effects and change the machine configuration. Therefore, while executing the code during exploration of PUTs or while running generated tests, we created a sand-box with 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`. We adopted sand-boxing after some of the Pex generated tests had corrupted our test machines. Since we use a 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 (Section 2.2) as a coverage criteria. We report our coverage in terms of the number of blocks covered in the code under test. We give only an approximate upper bound on the number of reachable basic blocks, since we do not know which blocks are actually reachable from the given PUTs for several reasons: we are executing the code in a sand-box, existing code is loaded from the disk only when it is used and new code may be generated at runtime.

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

4.4 RQ1: Generated Regression Tests

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

In the capture phase, DyGen recorded 433,809 dynamic traces and persisted them as C# source code, resulting in ≈1.5 GB of C# source code. The average trace length includes 21 method calls and the maximum trace length includes 52 method calls. Since 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, DyGen uses static analysis to filter out duplicate PUTs. Our static analysis took 45 minutes and resulted in 68,575 unique PUTs. DyGen uses dynamic analysis to filter out 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 5(b) 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 "With-Seeds Iteration 2", DyGen achieved 24.3% higher coverage than the base coverage. Table 2 shows more detailed results of coverage achieved for all ten .NET libraries. Column ".NET libraries" shows libraries under test. Column "Maximum Coverage"

.NET libraries	Maximum Coverage	Base Coverage	WithOutSeeds Iteration 1		WithOutSeeds Iteration 2		WithSeeds Iteration 1		WithSeeds Iteration 2	
	# blocks	# blocks	# blocks	% increase	# blocks	% increase	# blocks	% increase	# blocks	% increase
mscorlib	20437	12827	13063	1.84	13620	6.18	14808	15.44	15018	17.08
System	7786	4651	4062	-12.67	4243	-8.77	5907	27.00	6039	29.84
Forms	2815	1730	1572	-9.13	1774	2.54	1782	3.01	1865	7.80
Drawing	850	570	580	1.75	591	3.68	618	8.42	625	9.65
Xml	2770	1229	1390	13.10	1462	18.96	1959	59.40	2045	66.40
RegEx	854	351	330	-5.98	520	48.15	754	114.81	771	119.66
Config	392	263	297	12.93	297	12.93	302	14.83	306	16.35
Data	865	301	380	26.25	422	40.20	562	86.71	569	89.04
Web	253	154	211	37.01	212	37.66	212	37.66	212	37.66
Trans	59	35	35	0.00	35	0.00	35	0.00	35	0.00
TOTAL/AVG	**37081**	**22111**	**21920**	**<0**	**23176**	**4.80**	**26939**	**21.80**	**27485**	**24.30**

Table 2. Comparison of coverage achieved for ten .NET libraries used in our evaluation.

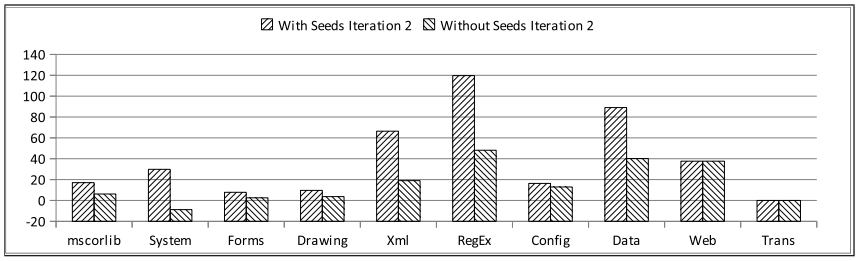


Fig. 6. Comparison of coverage achieved by Mode "WithSeeds Iteration 2" and Mode "WithoutSeeds Iteration 2".

shows an approximation of the upper bound (in terms of number of blocks) of achievable coverage in each library under test. In particular, this column shows the sum of all blocks in all methods that are (partly) covered by any generated test. However, we do not present the coverage results of our four modes as percentages relative to these upper bounds, since these upper bounds are only approximate values, whereas the relative increase of achieved coverage can be measured precisely. Column "Base Coverage" shows the number of blocks covered by seed tests for each library. Column "WithOutSeeds Iteration 1" shows the number of blocks covered ("# blocks") and the percentage of increase in the coverage ("% increase") with respect to the base coverage in this mode. Similarly, Columns "WithOutSeeds Iteration 2", "WithSeeds Iteration 1", and "WithSeeds Iteration 2" show the results for the other three modes.

Since we use seed tests during our exploration in Modes "WithSeeds Iteration 1" or "WithSeeds Iteration 2", the coverage achieved is either the same or higher than the base coverage. However, DyGen has achieved significant higher coverage than base coverage for libraries mscorlib and System (in terms of the number of additional blocks covered). The primary reason is that most of the classes in these libraries are stateless and do not require environment interactions. The results show that DyGen can handle large real-world code bases and can generate large number of regression tests that achieve 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 com-

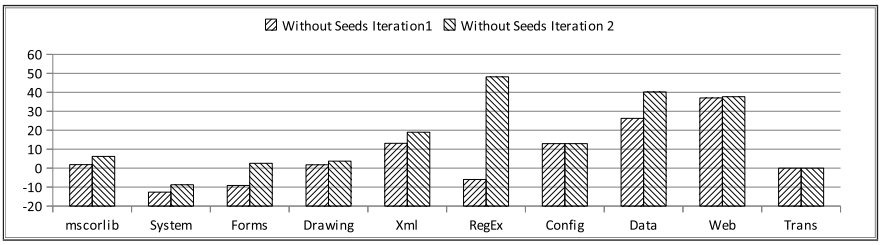


Fig. 7. Comparison of code coverage achieved by Modes "WithoutSeeds Iteration 1" and "WithoutSeeds Iteration 2".

pare the coverage achieved by generated tests in Modes "WithoutSeeds Iteration 2" and "WithSeeds Iteration 2". Figure 6 shows comparison of the coverage achieved in these two modes. The x-axis shows the library under test and y-axis shows the percentage of increase in the coverage with respect to the base coverage. As shown, Mode "WithSeeds Iteration 2" always achieved higher coverage than Mode "WithoutSeeds Iteration 2". On average "WithSeeds Iteration 2" achieved 18.6% higher coverage than "WithoutSeeds Iteration 2". The table also shows that there is a significant increase in the coverage achieved for the `System.Web.RegularExpressions` (`RegEx`) library. In Section 3.3, we described one of the major advantages of seed tests is that seed tests can help cover certain paths that are hard to be covered without using those tests. The `System.Web.RegularExpressions` library is an example for such paths since this library requires complex regular expressions to cover certain paths in the library. It is quite challenging for Pex or any other DSE-based approach to generate concrete values that represent regular expressions. The increase in the coverage for this library shows that concrete values in the seed tests help achieve higher coverage. In summary, the results show that seed tests help achieve higher coverage compared to without using seed tests.

4.6 RQ3: Using More Machine Power

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

Figure 7 shows the comparison of coverage achieved in Modes "WithoutSeeds Iteration 1" and "WithoutSeeds Iteration 2". On average, Mode "WithoutSeeds Iteration 2" achieved 5.73% higher coverage than Mode "WithoutSeeds Iteration 1". This result shows that DyGen can achieve additional coverage in further iterations. However, the coverage from Mode "WithoutSeeds Iteration 1" to Mode "WithoutSeeds Iteration 1" is not doubled. The primary reason is that it gets harder to cover new blocks in further iterations.

Figure 8 shows the comparison of coverage achieved in Modes "WithSeeds Iteration 1" and "WithSeeds Iteration 2". On average, Mode "WithSeeds Iteration 2" achieved 2.0% higher coverage than Mode "WithSeeds Iteration 1". The increase in coverage

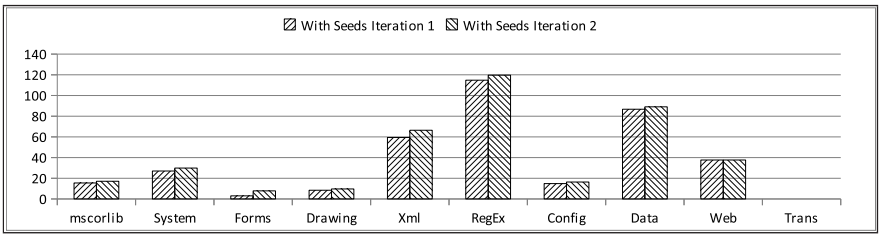


Fig. 8. Comparison of code coverage achieved by Modes "WithSeeds Iteration 1" and "Withseeds Iteration 2".

from Mode "WithSeeds Iteration 1" to Mode "WithSeeds Iteration 2" is less than the increase in the coverage from Mode "WithoutSeeds Iteration 1" to Mode "WithoutSeeds Iteration 2". This difference is due to seed tests that help achieve higher coverage during Mode "WithSeeds Iteration 1", leaving more harder blocks to be covered in Mode "WithSeeds Iteration 2". In summary, the results show that further iterations can help generate new regression tests that can achieve more coverage.

5 Discussion and Future Work

Although our generated tests achieved higher coverage (24.3%) than the seed tests, we did not achieve full overall coverage of our subject code bases (i.e. 100% coverage of all methods stored in the code bases on disk). There are three major reasons for not achieving full coverage. First, using a sand-box reduces the amount of executable code. Second, our recorded dynamic traces do not invoke all public methods of the libraries under analyses. In future work, we plan to address this issue by generating PUTs for all public methods that are not covered. Third, the code under test includes branches that cannot be covered with the test scenarios recorded during program executions. To address this issue, we plan to generate new test scenarios from existing scenarios by using evolutionary techniques [4].

We did not find any previously unknown defects while generating regression tests. We did not expect to find defects, since our subject code bases are well tested both manually and by automated tools, including research tools such as Randoop [7] and Pex [13]. Although regression testing is our ultimate goal, in our current approach, we primarily focused on generating regression tests that achieve high code coverage of the given version of software. In future work, we plan to apply these regression tests on further versions of software in order to detect regression defects. Furthermore, in our evaluation, we used two libraries as subject applications. However, our approach is not specific for libraries and can be applied to any application in practice.

6 Related Work

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

Regression testing. There exist approaches [18][19][16] that use a capture-and-replay strategy for generating regression tests. In the capture phase, these approaches monitor the methods called during program execution and use 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 DSE in generating new regression tests.

Another existing approach, called Orstra [20], augments an existing test suite with additional assertions to detect regression faults. To add these additional assertions, Orstra executes a 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. Another category of existing approaches [21][22][23] 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 the code changed between the two versions of software. However, all these approaches require an existing regression test suite, which is the primary focus of our current approach.

Method-call sequence generation. To test object-oriented programs, existing test-generation approaches [5][6][24][2] 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 [3] replaces concrete values for method arguments with symbolic values and exploits DSE techniques [9][10][11][12] 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 [7] 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. Randoop may also pick values for certain primitive randomly, or from a fixed manually supplied pool of values. Randoop incorporates feedback obtained from previously constructed method-call sequences while generating new sequences. As soon as a method-call sequence is constructed, Randoop executes the sequence and verifies whether the sequence violates any contracts and filters. Since Randoop does not symbolically analyze how the code under test uses arguments, Randoop is often unable to cover data-dependent code paths. On the other hand, DyGen is dependent on method-call sequences obtained via dynamic traces, and so DyGen is often unable to cover code paths that cannot be covered from the scenarios described by those sequences. Therefore, Randoop and DyGen are techniques with orthogonal goals and effects. In our previous approach [13], we applied Pex on a core .NET component for detecting defects. Unlike our new approach that uses realistic scenarios recorded during program executions, our previous approach generates individual PUTs for each public method of all public classes. There, we could not cover portions of the code that require long scenarios. Our new approach complements our previous approach by using realistic scenarios for covering such code portions.

Our approach is also related to another category of approaches based on mining source code [25][26] [27]. These approaches statically analyze code bases and use mining algorithms such as frequent itemset mining [28] 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 [27] 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 three 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 the static traces used in our previous work. Third, our new approach includes additional techniques such as seed tests and distributed setup for assisting DSE-based approaches in effectively generating CUTs from PUTs.

7 Conclusion

Automatic generation of method-call sequences that help achieve high structural coverage of object-oriented code is an important and yet a challenging problem in software testing. Unlike existing approaches that generate sequences randomly or based on analysis of the methods, we proposed a novel scalable approach that generates sequences via mining dynamic traces recorded during (typical) program executions. In this paper, we showed an application of our approach by automatically generating regression tests for two core .NET 2.0 framework libraries. In our evaluations, we showed that our approach recorded ≈1.5 GB (size of corresponding C# source code) of dynamic traces and eventually generated ≈500,000 regression tests, where each test exercised a unique path. The generated regression tests covered 27,485 basic blocks, which represents an improvements of 24.3% over the number of blocks covered by the original recorded dynamic traces. These numbers show that our approach is highly scalable and can be used in practice to deal with large real-world code bases. In future work, we plan to evaluate the effectiveness of generated regression tests in detecting behavioral differences between two versions of software.

References

1. Khurshid, S., Pasareanu, C.S., Visser, W.: Generalized symbolic execution for model checking and testing. In: Proc. TACAS. (2003) 553–568
2. Xie, T., Marinov, D., Notkin, D.: Rostra: A framework for detecting redundant object-oriented unit tests. In: Proc. ASE. (2004) 196–205
3. Inkumsah, K., Xie, T.: Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In: Proc. ASE. (2008) 297–306
4. Tonella, P.: Evolutionary testing of classes. In: Proc. ISSTA. (2004) 119–128
5. Csallner, C., Smaragdakis, Y.: JCrasher: An automatic robustness tester for Java. Softw. Pract. Exper. **34**(11) (2004) 1025–1050

6. Parasoft: Jtest manuals version 5.1. Online manual (2006) http://www.parasoft.com.
7. Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: Proc. ICSE. (2007) 75–84
8. Tillmann, N., Schulte, W.: Parameterized unit tests. In: Proc. ESEC/FSE. (2005) 253–262
9. Clarke, L.: A system to generate test data and symbolically execute programs. IEEE Trans. Softw. Eng. **2**(3) (1976) 215–222
10. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed automated random testing. In: Proc. PLDI. (2005) 213–223
11. King, J.C.: Symbolic execution and program testing. Communications of the ACM **19**(7) (1976) 385–394
12. Koushik, S., Darko, M., Gul, A.: CUTE: A concolic unit testing engine for C. In: Proc. ESEC/FSE. (2005) 263–272
13. Tillmann, N., de Halleux, J.: Pex: White box test generation for .NET. In: Proc. TAP. (2008) 134–153
14. Rosenblum, D.S., Weyuker, E.J.: Predicting the cost-effectiveness of regression testing strategies. SIGSOFT Softw. Eng. Notes **21**(6) (1996) 118–126
15. Z3: An efficient SMT solver (2010) http://research.microsoft.com/en-us/um/redmond/projects/z3/.
16. Saff, D., Artzi, S., Perkins, J.H., Ernst, M.D.: Automatic test factoring for Java. In: Proc. ASE. (2005) 114–123
17. Godefroid, P., Levin, M.Y., Molnar, D.: Automated whitebox fuzz testing. In: Proc. NDSS. (2008)
18. Elbaum, S., Chin, H.N., Dwyer, M.B., Dokulil, J.: Carving differential unit test cases from system test cases. In: Proc. FSE. (2006) 253–264
19. Orso, A., Kennedy, B.: Selective capture and replay of program executions. SIGSOFT Softw. Eng. Notes **30**(4) (2005) 1–7
20. Xie, T.: Augmenting automatically generated unit-test suites with regression oracle checking. In: Proc. ECOOP. (2006) 380–403
21. DeMillo, R.A., Offutt, A.J.: Constraint-based automatic test data generation. IEEE Trans. Softw. Eng. **17**(9) (1991) 900–910
22. Taneja, K., Xie, T.: DiffGen: Automated regression unit-test generation. In: Proc. ASE. (2008) 407–410
23. Evans, R.B., Savoia, A.: Differential testing: A new approach to change detection. In: Proc. ESEC/FSE. (2007) 549–552
24. Pacheco, C., Ernst, M.D.: Eclat: Automatic generation and classification of test inputs. In: Proc. ECOOP. (2005) 504–527
25. Engler, D., Chen, D.Y., Hallem, S., Chou, A., Chelf, B.: Bugs as deviant behavior: A general approach to inferring errors in systems code. In: Proc. SOSP. (2001) 57–72
26. Wasylkowski, A., Zeller, A., Lindig, C.: Detecting object usage anomalies. In: Proc. ESEC/FSE. (2007) 35–44
27. Thummalapenta, S., Xie, T., Tillmann, N., de Halleux, J., Schulte, W.: MSeqGen: Object-oriented unit-test generation via mining source code. In: Proc. ESEC/FSE. (2009) 193–202
28. Wang, J., Han, J.: BIDE: Efficient mining of frequent closed sequences. In: Proc. ICDE. (2004) 79 – 88