Study of Integrating Random and Symbolic Testing for Object-Oriented Software*

Marko Dimjašević[1], Falk Howar[2], Kasper Luckow[3], and Zvonimir Rakamarić[1]

[1] School of Computing, University of Utah, Salt Lake City, UT, USA
marko@cs.utah.edu,zvonimir@cs.utah.edu
[2] Department of Computer Science, Dortmund University of Technology, Dortmund, Germany falk.howar@tu-dortmund.de
[3] Carnegie Mellon University Silicon Valley, Mountain View, CA, USA
kasper.luckow@sv.cmu.edu

Abstract. Testing is currently the main technique adopted by the industry for improving the quality, reliability, and security of software. In order to lower the cost of manual testing, automatic testing techniques have been devised, such as random and symbolic testing, with their respective trade-offs. For example, random testing excels at fast global exploration of software, while it plateaus when faced with hard-to-hit numerically-intensive execution paths. On the other hand, symbolic testing excels at exploring such paths, while it struggles when faced with complex heap class structures. In this paper, we describe an approach for automatic unit testing of object-oriented software that integrates the two techniques. We leverage feedback-directed unit testing to generate meaningful sequences of constructor+method invocations that create rich heap structures, and we in turn further explore these sequences using dynamic symbolic execution. We implement this approach in a tool called JDoop, which we augment with several parameters for fine-tuning its heuristics; such "knobs" allow for a detailed exploration of the various trade-offs that the proposed integration offers. Using JDoop, we perform an extensive empirical exploration of this space, and we describe lessons learned and guidelines for future research efforts in this area.

1 Introduction

The software industry nowadays heavily relies on testing for improving the quality of its products. There are, of course, good reasons for adopting this practice. First, as opposed to more heavy-weight techniques such as static analysis, testing is easy to deploy and understand, and most developers are familiar with software testing processes and tools. Second, testing is scalable (i.e., millions of tests can be executed within hours even on large programs) and precise (i.e., it does not generate false alarms that impede developers' productivity). Third, while testing cannot prove the absence of bugs, there is ample evidence that testing does find important bugs that are fixed by developers. Despite these advantages, testing

*Supported in part by the National Science Foundation (NSF) award CCF 1421678.

```
public class HardToHit {
  private int x;
  public HardToHit(int x) {
    this.x = (x < 0) ? -x : x;
  }
  public void setX(int x) {
    this.x = x;
  }
  public int distance(int y) {
    y = (y < 0) ? -y : y;
    int out;
    if (x > y) out = x - y;
    else out = y - x;
    assert out >= 0;
    return out;
  }
}
```

```
// Assertion Violation
public void testHardToHit() {
  HardToHit h = new HardToHit(0);
  h.setX(-1);
  h.distance(Integer.MAX_VALUE);
}
// Random Parameters
public void testHardToHit1() {
  HardToHit h =
      new HardToHit(random());
  h.distance(random());
}
// Random Method Sequence
public void testHardToHit2() {
  HardToHit h = new HardToHit(0);
  h.setX(0);
  h.distance(0);
}
```

Fig. 1. Class with an assertion in one method (left). Input `x` is not properly sanitized in method `setX`. Consequence: assertion can be violated by combination of method sequence and specific input values (right).

is not a silver bullet since crafting good tests is a time consuming and costly process, and even then achieving high coverage and catching all defects using testing can be challenging. For example, tester to developer ratio at Microsoft is around 1-to-1, and yet important defects still escape into production. Naturally, there has been a great deal of research on alleviating these problems by developing techniques that aim to improve the automation and effectiveness (in terms of achieved coverage and defects found) of software testing.

Random testing is the most basic and straightforward approach to automating software testing. Typically, it completely automatically generates and executes millions of test cases within hours, and quickly covers many statements (or branches) of a *software under test* (SUT). However, a drawback of random testing is that, depending on the characteristics of the SUT, the achieved coverage plateaus due to unlikely execution paths. Fig. 1 gives our motivating example JAVA program that illustrates this point (left) together with a specific test case that triggers an assertion violation (top right). To apply random testing on the example, we generate randomized unit test shown in the middle of the right half of the figure. Clearly, it is trivial to execute this simple unit test many times, each time with a new pair of random numbers being generated. It is impossible, however, that executing it would generate inputs that violate the assertion. We would additionally need to generate more complex sequences of method calls (as is shown in lower right of the figure). Exploring both dimensions (parameter values and method sequences) randomly tends to plateau and not hit paths that require specifc combinations of method sequence and parameters values.

A more heavyweight approach could be based on symbolic execution, which leverages automatic constraint solvers to compute test inputs that cover such hard-to-cover branches. For example, the JDART [25] dynamic symbolic execution tool when run on method `testHardToHit2` generates test cases covering all branches in less than a second, thereby triggering an assertion violation. The authors also show that JDART improves coverage over random testing for a class of numerically-intensive SUTs. However, symbolic-testing-based methods mainly excel in automatically generating test inputs over primitive numeric data types, and have hence been successfully applied as either system-level (e.g., SAGE [17], KLEE [5]) or method-level (e.g., JDART [25], JCUTE [33]) test generators.

Generalizing from the above example, generating unit tests for object-oriented software poses a two-dimensional challenge: instead of taking just primitive types as input, methods in object-oriented software require a rich heap structure of class objects to be generated. While several approaches have been proposed that automatically generate symbolic heap structures [24], logical encoding of such structures results in more complex constraints that put an additional burden on constraint solvers; hence, these approaches have not yet seen wider adoption on large SUTs. On the other hand, generating heap structures by randomly creating sequences of constructor+method invocations was shown to be effective, in particular when advanced search- and feedback-directed algorithms are employed (e.g., RANDOOP [27], EvoSuite [12]). It is then natural to attempt to integrate the two approaches by using random testing to perform global/macro exploration (by generating heap structures using sequences of constructor+method invocations at the level of classes) and dynamic symbolic execution to perform local/micro exploration (by generating inputs of primitive types using constraint solvers at the level of methods). In this paper, we describe, implement, and empirically evaluate such a hybrid approach.

Our hybrid approach integrates feedback-directed unit testing with dynamic symbolic execution. We leverage feedback-directed unit testing to generate constructor+method sequences that create heap structures and drive a SUT into interesting global (i.e., macro) states. We feed the generated sequences to a dynamic symbolic execution engine to compute inputs of primitive types that drive the SUT into interesting local (i.e., micro) states. We implemented this approach as a tool named JDOOP,[1] which integrates feedback-directed unit testing tool RANDOOP [27] with state-of-the-art dynamic symbolic execution engine JDART [25]. Given that such an integration has not been thoroughly empirically studied in the past, we also assess the merits of this approach through a large-scale empirical evaluation.

Our main contributions are as follows:

- We developed JDOOP, a hybrid tool that integrates feedback-directed unit testing with dynamic symbolic execution to be able to experiment with large-scale automatic testing of object-oriented software.

[1] Note that a very preliminary version of JDOOP was presented earlier as a short workshop extended abstract [10].

- We implemented a distributed benchmarking infrastructure for running experiments in isolation on a cluster of machines; this allows us to execute large-scale experiments that ensure statistical significance, and also advances the reproducibility of our results.
- We performed an extensive empirical evaluation and comparison between random (our baseline) and hybrid testing approaches in the context of automatic testing of object-oriented software.
- We identified several open research questions during our evaluation, performed additional targeted experiments to obtain answers to these questions, and provided guidelines for future research efforts in this area.

2 Background

We provide background on dynamic symbolic execution and feedback-directed random testing.

2.1 Dynamic Symbolic Execution

Dynamic symbolic execution [16, 34, 5] is a program analysis technique that executes a program with concrete and symbolic inputs at the *same* time. It systematically collects constraints over the *symbolic* program inputs as it is exploring program paths, thereby representing program behaviors as algebraic expressions over symbolic values. The program effects can thus be expressed as a function of such expressions.

Dynamic symbolic execution maintains—in addition to the concrete state defined by the concrete program semantics—the symbolic state, which is a tuple containing symbolic values of program variables, a path condition, and a program counter. A path condition is a conjunction of symbolic expressions over the symbolic inputs that characterizes an execution path through the program. It is generated by accumulating (symbolic) conditions encountered along the execution path, so that concrete data values that satisfy it can be used to drive its concrete execution. Path conditions are stored as a *symbolic execution tree* that characterizes all the paths exercised as part of the symbolic analysis.

In dynamic symbolic execution, the symbolic execution tree is built by repeatedly augmenting it with new paths that are obtained from unexplored branches in the tree. This is done by employing an exploration strategy such as depth-first, breadth-first, or random. A constraint solver is used to obtain a valuation for a yet-unexplored branch by feeding it the corresponding path condition. The new valuation drives a new iteration of dynamic symbolic execution that augments the symbolic execution tree with a new path. JDart is a dynamic symbolic execution engine that uses the Java PathFinder framework [42, 22] and for executing Java programs and recording path conditions. Maintaining the symbolic state is achieved by a customized implementation of the bytecode instructions in the JVM of Java PathFinder that performs concrete and symbolic operations simultaneously. In JDoop, we configure JDart to use the Z3 [8] constraint

solver for finding concrete inputs that drive execution along previously unexplored symbolic paths.

A limitation of this approach is that native code is outside the scope of the analysis. Based on the NHANDLER extension [36] to JAVA PATHFINDER, JDART offers two strategies for dealing with native code.

- **Concrete Native.** In this mode, JDART executes native code on concrete data values, and no symbolic execution of native parts is performed—only concrete values are passed to and from native calls, and symbolic values are not updated and cannot taint native return values. The return value is annotated with a new symbolic variable. As a consequence, the concrete side of an execution is faithful to the respective execution on a normal JVM. However, branches in the native code are not recorded in symbolic path conditions, which can lead to JDART not being able to explore branches after a native call as well. Another downside of this mode is that the implementation in JAVA PATHFINDER is relatively slow.
- **No Native.** In this mode, JDART does not execute native code at all. Instead, it returns a default concrete value every time a native method is called and a return value is expected. The concrete value is annotated with the corresponding symbolic variable, using the method signature of the native method as the name of that variable. Concrete execution, in this case, is not faithful to the respective execution on a normal JVM as the introduced default values in most cases are not equal to the values that would be returned by the actual method invocations (and side effects are ignored as well). Recorded symbolic branches cannot be explored even if solutions are found by a constraint solver as there currently is no mechanism that allows feeding these values into the execution (instead of the default return values of native methods).

Since the 'No Native' mode is more performant and since currently there is no way of solving most of the recorded constraints in 'Concrete Native' mode (cf. results in Sec. 4), JDOOP runs JDART in 'No Native' mode for native code. We use the 'Concrete Native' mode in our evaluation for analyzing the potential limiting impact of not executing native code faithfully and not being able to find and inject values that target branches in native code.

JDART produces the following outputs: a symbolic execution tree that contains all explored paths along with performance statistics, vectors of concrete input values that execute paths in the tree, and a suite of test cases (based on these vectors). A symbolic execution tree contains leaf nodes for all explored paths and additionally leaves for branches off of executed paths that could not be explored because the constraint solver was not able to produce adequate concrete values or because native code is not executed (in fully symbolic mode). For these leaves JDART does not generate input vectors or test cases.

2.2 Feedback-Directed Random Testing

A simple approach to automatic unit testing of object-oriented software is to completely randomly generate sequences of constructor+method invocations to-

gether with the respective concrete input values. However, this typically results in a large overhead since numerous sequences get generated with invalid prefixes that lead to violations of common implicit class or method requirements (e.g., passing null reference to a method that expects an allocated object). Moreover, such sequences cause trivial, uninteresting exceptions to be thrown early, thereby preventing deep exploration of the SUT state space. Hence, instead of generating unit tests blindly and in a completely random fashion, useful feedback can be gathered from previous test executions to direct the creation of new unit tests. In this way, unit tests that execute long sequences of method calls to completion (i.e., without exceptions being thrown) can be generated. This approach is known as *feedback-directed random testing* and is implemented in the RANDOOP automatic unit testing tool [27].

RANDOOP uses information from previous test executions to direct further unit test generation. The tool maintains two sets of constructor+method invocation sequences: those that do not violate a property (i.e., property-preserving) and those that do (i.e., property-violating). The property-violating set is initially empty, while the property-preserving set is initialized with an empty sequence. The default property that is maintained is unit test termination without any errors or exceptions being thrown. RANDOOP randomly selects a public method (or a constructor) and an existing sequence from the property-preserving set. It then appends the invocation of the selected constructor/method to the end of the sequence, and replaces primitive type arguments with concrete values that are randomly selected from a preset pool of values. Next, the newly generated sequences are compared against all previously generated sequences in the two sets. If it already exists, it is simply dropped and random selection is repeated. Otherwise, RANDOOP executes the new sequence and checks for property violations. If no properties are violated, the sequence is added to the property-preserving set and otherwise to the property-violating set. RANDOOP keeps on extending property-preserving sequences until it reaches a provided time limit.

3 Hybrid Approach

In this section, we describe our hybrid approach that integrates dynamic symbolic execution and feedback-directed random testing into an algorithm for automatic testing of object-oriented software. We implemented this algorithm as the JDOOP tool that is freely available.[2] Fig. 2 shows the flow of the algorithm, which is iterative and each iteration consists of several stages that we describe next.

3.1 Generation of Sequences

The first stage of every iteration of our algorithm is feedback-directed random testing using RANDOOP, which generates constructor+method sequences as described in Sec. 2.2. RANDOOP takes advantage of a pool of concrete primitive

[2] JDOOP is available under the GNU General Public License version 3 (or later) at `https://github.com/psycopaths/jdoop`.

values to be used as constructor/method arguments when generating sequences. In the first iteration, we use the default pool with few values, which for the integer type are -1, 0, 1, 10, 100. Hence, an instance of a generated sequence for our running example from Fig. 1 is the one shown in the middle of the right half of the figure. Our algorithm grows the pool for subsequent iterations with concrete inputs generated by dynamic symbolic execution, which we describe later. The sequences generated in this stage serve two purposes. First, we employ them as standalone unit tests that exercise the SUT, which is their original intended purpose. Second, our hybrid algorithm also employs them as *driver programs* to be used in the subsequent dynamic symbolic execution stage.

3.2 Selection and Transformation of Sequences

The previous stage typically generates far too many sequences to be successfully explored with a dynamic symbolic execution engine in a reasonable amount of time. For example, several thousands of valid sequences are often generated in just a few seconds. Hence, it is prudent to select a promising subset of the generated sequences to be transformed into inputs for the subsequent dynamic symbolic execution with JDart. The second stage implements the selection and transformation of constructor+method sequences.

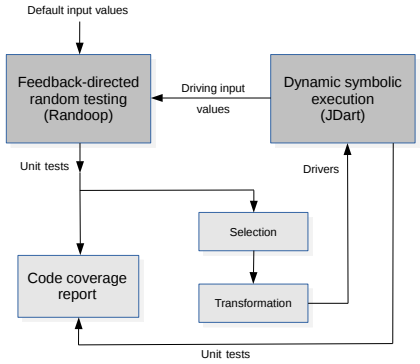

Fig. 2. Iterative algorithm of JDoop for unit test generation. The algorithm integrates dynamic symbolic execution and feedback-directed random testing.

Note that dynamic symbolic execution techniques have limitations, which is why we implemented the hybrid approach in the first place. In particular, they can typically treat symbolically only method arguments of primitive types. For example, if a sequence contains method calls with non-primitive types only, JDart will not be able to explore any additional paths. Hence, not every generated sequence is suitable for dynamic symbolic execution with JDart, and as the first step of this stage, we filter out all sequences with no arguments of a primitive type. Next, we have two strategies (i.e., heuristics) for selecting promising sequences. The first strategy randomly selects a subset of sequences. The second strategy prioritizes candidate sequences with more symbolic variables, which is based on the intuition that having more symbolic variables leads to more paths (and also branches and instructions) being covered. We compare the two strategies in our empirical evaluation. Once promising sequences are selected, they have to be appropriately transformed into driver programs for JDart.

Every candidate sequence is transformed for the final stage that performs dynamic symbolic execution. We achieve this by turning all constructor and method arguments of primitive types, which are supported by JDart, into symbolic input values. In our implementation, this is a simple source-to-source transforma-

tion. For instance, our example sequence results in the following driver program:

```
public class TestClass {
  void test1(int s0,int s1,int s2) {
    HardToHit h = new HardToHit(s0);
    h.setX(s1);
    h.distance(s2);
  }
}
```

```
static void main(String[] a) {
  TestClass tc = new TestClass();
  @Symbolic int x, y, z;
  tc.test1(x, y, z);
}
```

In the driver, the integer inputs to constructor `HardToHit` and methods `setX` and `distance` are transformed into the arguments of the `test1` test method. The `test1` method is called from the main method that is added as an entry point for dynamic symbolic execution. Finally, JDart is instructed that the `s0`, `s1`, and `s2` inputs to `test1` are treated symbolically.

3.3 Dynamic Symbolic Execution of Sequences

The last stage of every iteration is exploring the generated driver programs using dynamic symbolic execution as implemented in JDart. JDart explores paths through each driver program by solving path constraints over the specified symbolic inputs as described in Sec. 2.1. In the process, it generates additional unit tests, where each unit test corresponds to an explored path. The generated unit tests are added into the final set of unit tests. In addition to generating these unit tests, we also collect all the concrete input values that JDart generates in the process. We add these values back into the Randoop's concrete primitive value pool for the sequence generation stage of the next iteration. By doing this, we feed the information that the dynamic symbolic execution generates back into the feedback-directed random testing stage.

4 Empirical Evaluation

We aim to answer the following research questions using the results of our empirical evaluation.

1. Can JDoop cover paths that plain random test case generation does not, and how big is the positive impact of covering such paths? To answer this question, we compare the performance of Randoop (as our baseline) and JDoop, using code coverage as a metric for the quality of the generated test suites.
2. Can dynamic symbolic execution enable randomized test case generation to access regions of a SUT that remain untested otherwise, i.e., does the feedback loop from JDart to Randoop (see Fig. 2) have a measurable impact on achieved coverage? To answer this question, we run JDoop in multiple configurations with varying amounts of runtime attributed to Randoop and JDart, enabling a feedback loop in some configurations and preventing it in others.

Table 1. SF110 Benchmarks we use in the evaluation. Column #B is the number of branches, #I instructions, #M methods, and #C classes.

Benchmark	#B	#I	#M	#C
1_tullibee	915	8402	204	19
2_a4j	544	9773	522	45
3_gaj	22	415	52	10
5_templateit	564	5391	195	23
6_jnfe	132	7545	339	52
7_sfmis	146	4386	185	19
9_falselight	16	1189	32	14
11_imsmart	103	2244	86	17
13_jdbacl	3098	49385	1578	198
14_omjstate	52	954	67	14
16_templatedetails	38	656	87	24
22_byuic	2124	15031	195	14
23_jwbf	949	16032	609	86
26_jipa	128	1488	36	5
28_greencow	0	7	2	1
30_bpmail	208	3372	208	32
31_xisemele	150	3036	269	50
34_sbmlreader2	76	1447	26	8
37_petsoar	208	3445	377	58
42_asphodel	64	1139	101	20
46_nutzenportfolio	1183	18335	826	62

Benchmark	#B	#I	#M	#C
47_dvd-homevideo	376	10670	161	48
48_resources4j	312	3223	104	12
49_diebierse	197	4859	185	19
50_biff	814	7348	49	6
53_shp2kml	26	656	30	6
55_lavalamp	128	2907	236	48
63_objectexplorer	959	14118	902	84
65_gsftp	517	6587	181	32
67_gae-app-manager	68	1405	46	8
68_biblestudy	424	6005	313	23
69_lhamacaw	2016	51698	1437	101
72_battlecry	674	9550	130	15
74_fixsuite	374	6520	241	36
76_dash-framework	12	188	37	17
79_twfbplayer	1132	18315	902	160
84_ifx-framework	299	136363	26257	3900
90_dcparseargs	88	654	21	6
94_jclo	110	1094	43	4
95_celwars2009	850	15208	164	32
98_trans-locator	40	1097	39	6

3. What are the constituting factors impacting the effectiveness of JDoop in terms of the code coverage that can be achieved through automated generation of test suites? More specifically, can we confirm or refute the conjecture from related work [13] that robustness of the used dynamic symbolic execution engine is pivotal or do other factors exist that have an impact on the achievable coverage (e.g., selection of test cases for symbolic execution)? To answer this question, we analyze statistics produced by JDart and vary the strategy in JDoop for selecting method sequences for execution with JDart as discussed in Sec. 3 (either selecting sequences randomly or prioritizing those with many symbolic variables).

In the remainder of this section, we introduce the benchmarks we used in our evaluation, describe our experimental setup, and present and discuss the results of the evaluation.

4.1 Benchmarks

We performed our empirical evaluation using the SF110 benchmark suite [35]. The suite consists of 110 Java projects that were randomly selected from the SourceForge repository of free software to reduce the threat to external validity (see Sec. 5). In our evaluation, we chose the largest subset of SF110 that both JDoop and Randoop can successfully execute on. Benchmarks that were excluded can be grouped into the following categories: unsuitable environment, inadequate or empty benchmarks, and deficiencies of testing tools. In the unsuitable environment category, benchmarks require privileged permissions in the operating system, a properly set configuration file, or a graphical subsystem to be available. There are several empty benchmarks, benchmarks that call the

`System.exit()` method that is not trapped by testing tools, and benchmarks that are otherwise inadequate because of conflicting dependencies with our testing infrastructure. Finally, for some benchmarks RANDOOP generates test cases that do not compile. All such problematic benchmarks were excluded from consideration, which left us with 41 benchmarks total, as listed in Table 1. For each benchmark we list the number of instructions, branches, methods, and classes, which demonstrates we use a wide range of SUTs in terms of their size and complexity.

4.2 Experimental Setup

We used two tools in our empirical evaluation: JDOOP and RANDOOP (version 3.0.10). We explored several configurations of JDOOP, where each configuration is determined by three parameters. The first parameter is the time limit for the first stage of every iteration, which is when RANDOOP runs (see Sec. 2.2); we vary this parameter as 1, 9, and 20 minutes. The second parameter is the time limit for the second and third stages combined, which is when JDART runs; we vary this parameter as 1, 9, and 40 minutes. The third parameter determines the strategy for selecting constructor+method call sequences as candidates for dynamic symbolic execution between: (1) random selection (denoted by R), and (2) prioritization based on the number of symbolic variables (denoted by P). Each configuration is code-named as JD-O-J-S, where O is the time limit for RANDOOP, J is the time limit for JDART, and S is the sequence selection strategy used. We explored the following six JDOOP configurations: JD-1-9-P, JD-1-9-R, JD-9-1-P, JD-9-1-R, JD-20-40-P, and JD-20-40-R.

We carried out the evaluation in the Emulab testbed infrastructure [43]. We used 20 identical machines, each of which was equipped with two 2.4 GHz 64-bit 8-core processors, 64 GB of DDR4 RAM, and an SSD disk; the machines were running Ubuntu 16.04. We developed our testing infrastructure around the Apache Spark cluster computing framework. To facilitate reproducibility, each execution of a testing tool on a benchmark is performed in a pristine sandboxed virtualization environment. This is achieved via LXC containers running a reproducible build of Debian GNU/Linux code-named Stretch. We allocated 4 dedicated CPU cores and 8 GB of RAM to each container. Both RANDOOP and JDOOP are multi-threaded, and hence they utilized the multiple available CPU cores. Our testing infrastructure is freely available for others to use and extend.[3]

We allocate a one hour time limit per benchmark per testing tool/configuration for test case generation. Subsequent test case compilation and code coverage measurement phases are not counted toward the 1 hour time limit. Given that both RANDOOP and JDOOP employ randomized heuristics, we repeat each run 5 times to account for this variability — for each benchmark we compute an average and a standard deviation. In terms of code coverage metrics, we measured instruction and branch coverage at the JAVA bytecode level using JACOCO [19].

[3]The testing infrastructure is available under the GNU Affero GPLv3+ license at `https://github.com/soarlab/jdoop-wrapper`.

Table 2. Branch coverage (including standard deviations) averaged across 5 runs. The highest and lowest numbers per benchmark are given in bold and italic, respectively.

Benchmark	RANDOOP	JD-1-9-P	JD-1-9-R	JD-9-1-P	JD-9-1-R	JD-20-40-P	JD-20-40-R
1_tullibee	*28.0* ± 1.2	29.4 ± 1.4	29.7 ± 1.0	29.4 ± 0.0	**31.6 ± 0.5**	28.7 ± 0.0	29.0 ± 0.2
2_a4j	58.5 ± 0.5	*57.9 ± 0.0*	60.0 ± 0.6	59.6 ± 0.2	**62.4 ± 0.1**	60.7 ± 0.1	60.7 ± 0.0
3_gaj	40.9 ± 0.0	40.9 ± 0.0	40.9 ± 0.0	40.9 ± 0.0	40.9 ± 0.0	40.9 ± 0.0	40.9 ± 0.0
5_templateit	2.1 ± 0.0	2.1 ± 0.0	2.1 ± 0.0	2.1 ± 0.0	2.1 ± 0.0	2.1 ± 0.0	2.1 ± 0.0
6_jnfe	48.5 ± 0.0	48.5 ± 0.0	48.5 ± 0.0	48.5 ± 0.0	48.5 ± 0.0	48.5 ± 0.0	48.5 ± 0.0
7_sfmis	*35.9 ± 0.9*	40.4 ± 4.2	39.7 ± 4.2	**42.5 ± 0.0**	40.5 ± 5.5	37.5 ± 2.7	37.1 ± 2.7
9_falselight	6.3 ± 0.0	6.3 ± 0.0	6.3 ± 0.0	6.3 ± 0.0	6.3 ± 0.0	6.3 ± 0.0	6.3 ± 0.0
11_imsmart	17.5 ± 0.0	17.5 ± 0.0	17.5 ± 0.0	17.5 ± 0.0	17.5 ± 0.0	17.5 ± 0.0	17.5 ± 0.0
13_jdbacl	36.6 ± 0.7	32.2 ± 3.1	32.2 ± 1.8	37.0 ± 0.5	**38.5 ± 0.6**	34.2 ± 1.0	33.6 ± 0.8
14_omjstate	48.1 ± 0.0	48.1 ± 0.0	48.1 ± 0.0	48.1 ± 0.0	**48.8 ± 3.1**	42.3 ± 0.0	42.3 ± 0.0
16_templatedetails	71.1 ± 0.0	68.4 ± 0.0	70.0 ± 1.8	71.1 ± 0.0	71.1 ± 0.0	68.4 ± 0.0	68.4 ± 0.0
22_byuic	7.8 ± 0.0	7.8 ± 0.0	7.8 ± 0.0	7.8 ± 0.0	7.8 ± 0.0	7.8 ± 0.0	7.8 ± 0.0
23_jwbf	26.6 ± 2.1	26.5 ± 1.7	27.2 ± 0.9	28.0 ± 0.6	**28.2 ± 1.9**	26.1 ± 0.5	*26.0 ± 0.0*
26_jipa	*18.8 ± 0.0*	24.2 ± 0.0	24.2 ± 0.0	24.2 ± 0.0	24.2 ± 0.0	23.4 ± 0.0	23.4 ± 0.0
28_greencow	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
30_bpmail	36.9 ± 0.5	36.9 ± 1.5	*36.1 ± 1.2*	**37.3 ± 0.6**	37.2 ± 0.6	37.2 ± 0.6	37.1 ± 0.5
31_xisemele	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
34_sbmlreader2	10.5 ± 0.0	10.5 ± 0.0	10.5 ± 0.0	10.5 ± 0.0	10.5 ± 0.0	10.5 ± 0.0	10.5 ± 0.0
37_petsoar	**54.1 ± 0.7**	*52.8 ± 1.6*	52.9 ± 1.4	53.4 ± 0.0	53.4 ± 0.0	53.7 ± 0.7	53.7 ± 0.7
42_asphodel	9.4 ± 0.0	9.4 ± 0.0	9.4 ± 0.0	9.4 ± 0.0	9.4 ± 0.0	9.4 ± 0.0	9.4 ± 0.0
46_nutzenportfolio	5.5 ± 0.0	*5.2 ± 0.0*	5.3 ± 1.6	5.6 ± 0.0	5.6 ± 0.6	5.5 ± 0.0	5.5 ± 0.0
47_dvd-homevideo	0.8 ± 0.0	0.8 ± 0.0	0.8 ± 0.0	0.8 ± 0.0	0.8 ± 0.0	0.8 ± 0.0	0.8 ± 0.0
48_resources4j	0.6 ± 0.0	0.6 ± 0.0	0.6 ± 0.0	0.6 ± 0.0	0.6 ± 0.0	0.6 ± 0.0	0.6 ± 0.0
49_diebierse	13.7 ± 0.0	*13.4 ± 3.0*	**19.7 ± 1.0**	14.2 ± 0.0	15.2 ± 15.1	13.7 ± 0.0	18.6 ± 13.1
50_biff	0.1 ± 0.0	0.1 ± 0.0	0.1 ± 0.0	0.1 ± 0.0	0.1 ± 0.0	0.1 ± 0.0	0.1 ± 0.0
53_shp2kml	19.2 ± 0.0	19.2 ± 0.0	19.2 ± 0.0	19.2 ± 0.0	19.2 ± 0.0	19.2 ± 0.0	19.2 ± 0.0
55_lavalamp	49.8 ± 0.6	48.4 ± 0.0	48.8 ± 1.6	51.9 ± 0.7	**52.0 ± 0.7**	48.4 ± 0.0	*48.0 ± 2.0*
63_objectexplorer	25.3 ± 0.0	24.6 ± 1.8	*24.5 ± 1.0*	**26.4 ± 0.3**	26.3 ± 0.9	25.0 ± 0.0	25.0 ± 0.2
65_gsftp	9.8 ± 1.0	9.9 ± 1.0	**10.0 ± 0.9**	9.9 ± 0.0	9.9 ± 0.0	9.5 ± 0.0	9.5 ± 0.0
67_gae-app-manager	2.9 ± 0.0	2.9 ± 0.0	2.9 ± 0.0	2.9 ± 0.0	2.9 ± 0.0	2.9 ± 0.0	2.9 ± 0.0
68_biblestudy	**37.5 ± 0.0**	*36.9 ± 0.7*	37.0 ± 0.4	37.3 ± 0.0	37.2 ± 0.8	37.0 ± 0.0	37.0 ± 0.0
69_lhamacaw	42.7 ± 0.4	40.1 ± 0.6	*39.9 ± 1.0*	**46.1 ± 0.5**	45.7 ± 0.6	40.3 ± 0.7	40.1 ± 0.4
72_battlecry	0.1 ± 0.0	0.1 ± 0.0	0.1 ± 0.0	0.1 ± 0.0	0.1 ± 0.0	0.1 ± 0.0	0.1 ± 0.0
74_fixsuite	17.5 ± 6.5	17.1 ± 3.1	*15.5 ± 1.3*	19.2 ± 1.8	**19.6 ± 4.0**	17.4 ± 2.8	17.2 ± 3.3
76_dash-framework	50.0 ± 0.0	50.0 ± 0.0	50.0 ± 0.0	50.0 ± 0.0	50.0 ± 0.0	50.0 ± 0.0	50.0 ± 0.0
79_twfbplayer	27.3 ± 0.0	23.2 ± 1.8	*21.5 ± 1.3*	29.4 ± 0.0	29.3 ± 1.0	**29.5 ± 0.0**	29.4 ± 0.1
84_ifx-framework	30.8 ± 0.0	32.6 ± 9.7	31.0 ± 8.9	**32.9 ± 2.8**	32.0 ± 2.8	29.5 ± 4.9	*28.8 ± 2.3*
90_dcparseargs	64.8 ± 0.0	64.8 ± 0.0	64.8 ± 0.0	64.8 ± 0.0	64.8 ± 0.0	64.8 ± 0.0	64.8 ± 0.0
94_jclo	*42.7 ± 0.0*	**46.0 ± 1.6**	44.5 ± 0.0	44.5 ± 0.0	44.7 ± 0.8	44.5 ± 0.0	44.5 ± 0.0
95_celwars2009	2.1 ± 0.0	2.1 ± 0.0	2.1 ± 0.0	**2.2 ± 5.2**	2.1 ± 0.0	2.1 ± 0.0	2.1 ± 0.0
98_trans-locator	25.0 ± 0.0	*15.0 ± 36.5*	18.0 ± 37.7	25.0 ± 0.0	**27.0 ± 3.7**	25.0 ± 0.0	25.0 ± 0.0

Furthermore, to get more insight into the performance of JDART, we collect statistics on the number of successful and failed runs, additional test cases it generates, symbolic variables in driver programs, times a constraint solver could not find valuation for a path condition, and JDART runs that explored one path versus multiple paths.

4.3 Evaluation of Test Coverage

Table 2 gives branch coverage results for each tool and configuration on all of the benchmarks. Most results are stable across multiple runs, meaning that the calculated standard deviations are very small. In particular, the standard deviations for RANDOOP on a vast majority of benchmarks are 0, even though we used a different random seed for every run. This suggests that RANDOOP reaches saturation and is unable to cover more branches. For the most part there are only small differences in the achieved coverage between different tools/configurations when looking at the total number of covered branches. However, JDOOP (in one of its configurations) consistently achieves higher coverage than RANDOOP. Given that pure RANDOOP saturates, we can conclude that the improvements in coverage we observe with JDOOP are due to leveraging dynamic symbolic

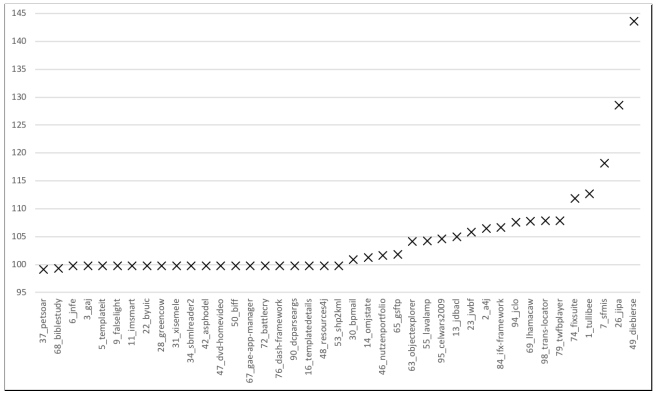

Fig. 3. Increases in branch coverage per benchmark by JDoop over baseline of Randoop (in % of coverage by baseline).

execution. Among JDoop configurations, best-performing are the two 9-1 configurations where in an iteration Randoop runs for 9 minutes and JDart for 1 minute; there are 6 such iterations in the 1 hour time limit.

Fig. 3 shows the increase in branch coverage per benchmark over pure Randoop that is achieved by some configuration of JDoop. The increase is measured as a percent increase in number of branches covered by JDoop over Randoop. Standard deviation is omitted in this graph as it was small in most instances (cf. Table 2). In two benchmarks JDoop performes slightly worse than pure Randoop. In roughly half of the benchmarks no change is observed — and in most cases with no variance. This suggests that these benchmarks are simply not amenable to increasing coverage by use of symbolic execution. In the remaining half of the benchmarks, branch coverage is increased. Increases range from 101.1% to 143.8% achieved coverage relative to the baseline of pure Randoop, with an average increase of 109.6% across this half of benchmarks.

4.4 Profiling Dynamic Symbolic Execution

To analyze the potential impact of the robustness of dynamic symbolic execution on the validity of our results, we collected data from runs on all benchmarks for all configurations. We perform this analysis on data from single runs of JDoop as the other results show very little variation of results between different runs in most cases. Table 3 reports statistics on the JDart operation in different series of experiments. Data in the table is explained and discussed in the following paragraphs.

Modes of Operation. For all of the analyzed configurations of JDoop, JDart runs successfully in the vast majority of cases and produces significant numbers of test cases (up to 16,588 in total for all benchmarks in one experiment). Most additional test cases are produced in the JD-1-9 configurations that enable the feedback loop between Randoop and JDart but grant the bulk of runtime

Table 3. Statistics produced by JDART for single runs of all benchmarks in different configurations of JDOOP. JDART uses NHANDLER in the 'No Native' mode, except for one experiment that we performed in the 'Concrete Native' (CN) mode.

	JD-20-40		JD-1-9		JD-9-1		JD-9-1 (CN)
Sequence Selection Strategy	R	P	R	P	R	P	R
Potential Impact / Best Mode of Operation							
# Successful Runs	33,390	28,316	46,976	43,770	4,629	1,017	3,885
Successful Runs (%)	98.5	97.8	98.2	97.5	98.5	100.0	96.3
# Additional Tests	6,436	10,802	11,272	16,588	914	5,382	n/a
# Benchmarks with Additional Tests	19	9	20	13	18	4	n/a
Robustness and Scalability of JDART							
# Failed Runs	507	648	853	1,121	69	0	148
due to unhandled native code	3	1	14	6	1	0	10
due to classloading in SUT	504	647	839	1,115	68	0	138
Failed Runs (%)	1.5	2.2	1.8	2.5	1.5	0.0	3.7
# D/K Paths	17	192	170	84	5	0	26,915
D/K Paths (%)	0.3	1.8	1.5	0.5	0.0	0.0	93.6
Amenable Test Cases							
# Symbolic Variables per Test Case (Avg.)	2.1	6.6	1.9	4.7	2.0	6.2	1.9
# Runs of Single Paths	32,410	27,293	45,268	42,162	4,495	988	2,801
# Runs with Multiple Paths	980	1,023	1,708	1,608	134	29	1,084

to JDART. Across all configurations, random selection of method sequences for JDART leads to generating additional test cases for more benchmarks than prioritizing sequences with many symbolic variables. Prioritization, on the other hand, leads to more additional test cases in total.

Robustness and Scalability. Our data indicates that JDART is robust. Only a small number of runs fail (between 0.0% and 2.5%). Of these failures, only a tiny fraction is due to unhandled native code (less than 1%).[4] The vast majority of failed runs is caused by class-path issues in the benchmarks (more than 99%). There are only very few cases in which the constraint solver was not able to solve constraints of all paths in symbolic execution trees (between 0.0% and 1.8%).

Using NHANDLER in the 'Concrete Native' mode leads to native calls being executed faithfully and to longer recorded path conditions, as discussed in Sec. 2. This yields constraints that are marked as not solvable ('don't know' or D/K for short) in 93.6% of all discovered paths in symbolic execution trees. This indicates the likelihood of JDART not being able to explore most of the paths that could be explored with proper symbolic treatment of native methods. Table 4 reports the number of occurrences for all encountered native methods in one run of JDOOP. As can be seen from the data, the *charAt* method of the String class offers by far the greatest potential for improving on the number of explored paths. Note, however, that numbers in the table do not necessarily translate into the same number of additional paths as occurrences are counted along paths in trees and the same method call may appear on multiple paths.

Amenable Test Cases. The number of symbolic variables per test case behaves as expected: it increases when using prioritization of sequences with many vari-

[4] These are methods for which NHANDLER was not configured to take over execution, leading to a crash of JDART. We configured NHANDLER to take care of all native methods of java.lang.String.

Table 4. Symbolic Variables introduced by NHANDLER in the 'Concrete Native' mode in a single run of JD-9-1.

Method	Occurrences
java.lang.String.charAt(I)C	2,157,258
java.lang.String.indexOf(I)I	430,951
java.lang.String.indexOf(II)I	18,199
java.lang.Character.isWhitespace(C)Z	63,723
java.lang.Character.isLetterOrDigit(C)Z	18,517
java.lang.Character.toLowerCase(C)C	16,506
java.lang.Math.min(II)I	2,800
java.lang.Float.floatToRawIntBits(F)I	81
sun.misc.Unsafe.compareAndSwapInt(Ljava/lang/Object;JII)Z	4,008

ables. Prioritization, however, comes at a cost since there tends to be more runs of JDART in configurations that do not use prioritization. For all benchmarks, a high number of runs yields only one path and hence no additional test cases. A considerable number of these runs may be attributed to using NHANDLER in the 'No Native' mode, thereby hiding branches by not executing native code. On the other hand, even in the experiment in which NHANDLER was used in the 'Concrete Native' mode, two thirds of all runs explored only a single path. This indicates that many method sequences that were selected for JDART simply do not branch on symbolic variables.

4.5 Discussion

The obtained results allow us to provide answers to our research questions.

Question 1: Covering More Paths. JDOOP consistently outperforms RANDOOP on roughly 50% of the benchmarks (see Table 2 and Fig. 3). Measured in absolute number of branches, the margins are relatively slim in many cases. There are, however, cases in which the achieved branch coverage is increased by 28% — resulting in an increase in code coverage by 5.4 percentage points (26_jipa). On about 50% of the benchmarks no variation can be seen in coverage between both approaches. Together with the little variance that is observed between different runs this indicates that RANDOOP in many cases reaches a state where achievable coverage is (nearly) saturated. It makes sense that in such a scenario JDOOP does not add many percentage points in code coverage. It merely adds coverage through those hard-to-hit corner cases.

Question 2: Reachable Regions. Our results indicate that the feedback loop has a positive impact. The JD-9-1 configurations perform better than other configurations in most cases. Regarding the time distribution between RANDOOP and JDART the picture is not as clear. There is a lot more variation in the margins of coverage increase (or decrease sometimes) for the configuration that grants most of the time to JDART. In one particularly amenable case this results in coverage increase by 43% (from 13.7% to 19.7% for 49_diebierse).

Question 3: Robustness of Symbolic Execution. Here, we have to refute the conjecture that was made in related work [13], namely that a robust dynamic symbolic execution engine can reap big increases in code coverage — or at least

curb expectations about achievable coverage increases. Our experiments showed that JDart handles most benchmarks without many problems. Proper analysis of native code (especially for String methods) certainly has the potential to improve code coverage further, but the consistently high number of symbolic analyses that result in a single path (even in the control experiment) points to another important factor that contributes to small margins: the generated test cases simply do not allow exploring many new branches in most cases.

The experiments even indicate that it does not pay off to prioritize method sequences with many variables for JDart. Prioritization adds cost twice: once for analyzing test cases and then for exploring with many variables. Taking into account the observation from the first answer, that Randoop (almost) achieves saturation of coverage in one hour, this again indicates that in JDoop corner cases are discovered by JDart. Covering more search space beats investigating the few locations more intensively in such a scenario.

Remark on Achievable Coverage. Our observations correlate well with the observations made in [11], where the results of a static analysis of the SF110 benchmark suite are reported. The analysis revealed that only 6.6% of methods in the benchmark suite have path constraints that are exclusively composed of primitive type elements. On the other hand, the study identified objects in path constraints, calls to external libraries or native code, and exception-dependent paths as challenges to symbolic execution. The authors report that one third of methods have paths that deal with exceptions.

The low coverage (in absolute numbers) and low variance across all benchmarks for Randoop and JDoop in our experiments suggests that many branches simply cannot be covered by test cases that only rely on calling methods of objects from a project under test. Many branches rely on return values of calls to external libraries or the occurence of exceptions, which are not triggered in a simple testing environment. Since there is no simple or automated approach for determining the achievable coverage for a benchmark, we sampled a few individual benchmarks and indeed quickly found cases where `catch`-blocks in the code contained comments to the effect that the block is unreachable.

Taking into account the results from [11] and our findings, we conjecture that the branch coverage that is achieved by JDoop is close to the coverage that can be achieved without making the environment of a tested project symbolic.

5 Threats to Validity

Threats to External Validity. While the main purpose of the SF110 corpus of benchmarks is to reduce the threat to external validity since they were chosen randomly, we cannot be absolutely sure that the benchmarks we used are representative of Java programs. In addition, we excluded a number of problematic benchmarks from our evaluation (see Sec. 4.1). Hence, our results might not generalize to all programs. In JDoop we integrated Randoop and JDart, and we used Randoop as the baseline in our evaluation. We attempted to include another contemporary state-of-the-art Java testing tool into the comparison, and

EvoSuite was an obvious choice to try. However, to the best of our ability we did not manage to get it to work with JaCoCo (the tool we use for measuring code coverage) on our benchmark suite despite exchanging numerous emails with the EvoSuite authors. This is a well-known problem caused by the online bytecode modifications that EvoSuite often performs.[5] While others successfully combined EvoSuite and JaCoCo in the past, that was accomplished only on very simple programs; in addition, others also reported differences in coverage results between EvoSuite's internal measurements and JaCoCo.[6,7] Hence, we could not perform a direct comparison and our results might not generalize to other tools. However, earlier work on EvoSuite reports similar results to ours with respect to using dynamic symbolic execution in combination with random testing [13]. Finally, note that we do not include the environment and dependencies of benchmarks into unit test generation, which might lead to sub-optimal coverage.

Threats to Internal Validity. In our evaluation, we experimented with 3 different time allocations for Randoop and JDart that we identified as representative. While our results show no major differences between these different time allocations, we did not fully explore this space and there might be a ratio that would lead to a different outcome. JDart currently cannot symbolically explore native calls, which might lead to not being able to cover program paths (and hence also branches and instructions) that depend on such calls. Our evaluation shows that this indeed happens and that native implementations of methods of the String class in Java are the main culprit, but it does not allow us to provide an estimate of the impact on the achieved code coverage. Finally, while we extensively tested JDoop to make sure it is reliable and performed sanity checks of our results, there is a chance for a bug to have crept in that would influence our results.

Threats to Construct Validity. Here, the main threat is the metrics we used to assess the quality of the generated test suites, and in particular branch coverage in the presence of dead code [**?**,**?**]. This threat is reduced by previous work showing that branch coverage performs well as a criterion for comparing test suites [15].

6 Related Work

Symbolic Execution. Dynamic symbolic execution [16, 34] is a well-known technique implemented by many automatic testing tools (e.g., [5, 17, 41, 33]). For example, SAGE [17] is a white-box fuzzer based on dynamic symbolic execution. It has been routinely applied to large software systems, such as media players and image processors, where it has been successful in finding security bugs.

[5] http://www.evosuite.org/documentation/measuring-code-coverage

[6] https://groups.google.com/forum/#!topic/evosuite/ctk2yPIqIoM

[7] https://stackoverflow.com/questions/41632769/evosuite-code-coverage-does-not-match-with-jacoco-coverage

Khurshid et al. [24] extend symbolic execution to support dynamically allocated structures, preconditions, and concurrency.

Several symbolic execution tools specifically target Java bytecode programs. A number of them implement dynamic symbolic execution via Java bytecode instrumentation. JCute [33], the first dynamic symbolic execution engine for Java, uses Soot [37] for instrumentation and lp_solve for constraint solving. CATG [39] uses ASM [2] for instrumentation and CVC4 [9] for constraint solving. Another dynamic symbolic execution engine, LCT [23], supports distributed exploration; it uses Boolector and Yices for solving, but it does not have support for float and double primitive types. A drawback of instrumentation-based tools is that instrumentation at the time of class loading is confined to the SUT. For example, LCT does not by default instrument the standard Java libraries thus limiting symbolic execution only to the SUT classes. Hence, the instrumentation-based tools discussed above provide the possibility of using symbolic models for non-instrumented classes or using pre-instrumented core Java classes.

Several dynamic symbolic execution tools for Java are not based on instrumentation. For example, the dynamic symbolic white-box fuzzer jFuzz [20] is based on Java PathFinder (as is JDart) and can thus explore core Java classes without any prerequisites. Symbolic PathFinder (SPF) [30] is a Java PathFinder extension similar to JDart. In fact, JDart reuses some of the core components of an older version of SPF, notably the solver interface and its implementations. While at its core SPF implements symbolic execution, it can also switch to concrete values in the spirit of dynamic symbolic execution [28]. That enables it to deal with limitations of constraint solvers (e.g., non-linear constraints).

Hybrid Approaches. There are several approaches similar to ours that combine fuzzing or a similar testing technique with dynamic symbolic execution. Garg et al. [14] propose a combination of feedback-directed random testing and dynamic symbolic execution for C and C++ programs. However, they are addressing challenges of a different target language and on a much smaller collection of benchmarks that they simplified before evaluation. The Driller tool [38] interleaves fuzzing and dynamic symbolic execution for bug finding in program binaries, and it targets single-file binaries in search of security bugs. Galeotti et al. [13] apply dynamic symbolic execution in the EvoSuite tool to explore test cases generated with a genetic algorithm. Even though their evaluation is carried out in a different way than the one presented in this paper, the general conclusion is the same in spirit — dynamic symbolic execution does not provide a lot of additional coverage on real-world object-oriented Java software on top of a random-based test case generation technique. MACE [6] combines automata learning with dynamic symbolic execution to find security vulnerabilities in protocol implementations.

There are other automated hybrid software testing tools that do not strictly combine with symbolic execution (e.g., OCAT [21], Agitator [4], Evacon [18], Seeker [40], DSD-Crasher [7]). Because these tools either focus on a single method at a time or just form random method call sequences, they often fail to drive pro-

gram execution to hard-to-reach sites in a SUT, which can result in suboptimal code coverage.

Random Testing. Randoop [27] is a feedback-directed random testing algorithm that forms random test cases that are sequences of method calls, while ensuring basic properties such as reflexivity, symmetry, and transitivity. Search-based software testing [26] approaches and tools are gaining traction, which is reflected in four annual search-based software testing tool competitions in recent years [31]. A prominent search-based tool is EvoSuite [12], which combines a genetic algorithm and dynamic symbolic execution. T3 [29] is a tool that generates randomized constructor and method call sequences based on an optimization function. JTExpert [32] keeps track of methods that can change the underlying object and constructs method sequences that are likely to get the object into a desired state. All the search-based testing tools are geared toward testing at the class level, while JDoop performs testing at the application/library level.

Benchmarking Infrastructures. In computer science, any extensive empirical evaluation, software competition, or reproducible research requires a significant software+hardware infrastructure. The Software Verification Competition's BenchExec [3] is a software infrastructure for evaluating verification tools on programs containing properties to verify. It comes with an interface for verification tools to follow, which did not fit our needs: our coverage measurement outcomes cannot be judged in terms of program correctness. The Search-based Software Testing Competition [31] community created an infrastructure for the competition as well. However, just like tools that participate in the competition, their infrastructure is geared toward running a testing tool on just one class at a time. Emulab [43] and Apt [1] are testbeds that provide researchers with an accessible hardware and software infrastructure. They allow for repeatable and reproducible research, especially in the domain of computer systems, by providing an environment to specify the hardware to be used, on top of which users can install and configure a variety of systems.

7 Conclusions

We introduced a hybrid automatic testing approach for object-oriented software, described its implementation JDoop, and performed an extensive empirical exploration of this space. Our approach is an integration of feedback-directed random testing (Randoop) and dynamic symbolic execution (JDart), where random testing performs global exploration, while dynamic symbolic execution performs local exploration (around interesting global states) of a SUT. It is an iterative algorithm where these two exploration techniques are interleaved in multiple iterations. Our evaluation on real-world object-oriented software shows that dynamic symbolic execution provides consistent improvements in terms of code coverage on top of our baseline (pure feedback-directed random testing) on those examples that are amenable to this method of testing.

References

1. Apt testbed facility. https://www.aptlab.net
2. ASM: A Java bytecode engineering library. http://asm.ow2.org
3. Beyer, D.: Reliable and reproducible competition results with BenchExec and witnesses (report on SV-COMP 2016). In: TACAS. pp. 887–904 (2016)
4. Boshernitsan, M., Doong, R., Savoia, A.: From Daikon to Agitator: Lessons and challenges in building a commercial tool for developer testing. In: ISSTA. pp. 169–180 (2006)
5. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI. pp. 209–224 (2008)
6. Cho, C.Y., Babić, D., Poosankam, P., Chen, K.Z., Wu, E.X., Song, D.: MACE: Model-inference-assisted concolic exploration for protocol and vulnerability discovery. In: Proceedings of the 20th USENIX Security Symposium (2011)
7. Csallner, C., Smaragdakis, Y., Xie, T.: DSD-Crasher: A hybrid analysis tool for bug finding. ACM Transactions on Software Engineering and Methodology pp. 1–37 (2008)
8. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS. pp. 337–340 (2008)
9. Deters, M., Reynolds, A., King, T., Barrett, C.W., Tinelli, C.: A tour of CVC4: how it works, and how to use it. In: FMCAD. p. 7 (2014)
10. Dimjašević, M., Rakamarić, Z.: JPF-Doop: Combining concolic and random testing for Java. In: Java Pathfinder Workshop (JPF) (2013), extended abstract
11. Eler, M.M., Endo, A.T., Durelli, V.H.S.: Quantifying the characteristics of java programs that may influence symbolic execution from a test data generation perspective. In: COMPSAC. pp. 181–190 (2014)
12. Fraser, G., Arcuri, A.: EvoSuite: Automatic test suite generation for object-oriented software. In: ESEC/FSE. pp. 416–419 (2011)
13. Galeotti, J.P., Fraser, G., Arcuri, A.: Improving search-based test suite generation with dynamic symbolic execution. In: ISSRE. pp. 360–369 (2013)
14. Garg, P., Ivančić, F., Balakrishnan, G., Maeda, N., Gupta, A.: Feedback-directed unit test generation for C/C++ using concolic execution. In: ICSE. pp. 132–141 (2013)
15. Gligoric, M., Groce, A., Zhang, C., Sharma, R., Alipour, M.A., Marinov, D.: Comparing non-adequate test suites using coverage criteria. In: ISSTA. pp. 302–313 (2013)
16. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed automated random testing. In: PLDI. pp. 213–223 (2005)
17. Godefroid, P., Levin, M.Y., Molnar, D.: SAGE: Whitebox fuzzing for security testing. Queue **10**(1), 20:20–20:27 (2012)
18. Inkumsah, K., Xie, T.: Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In: ASE. pp. 297–306 (2008)
19. JaCoCo Java code coverage library. http://www.jacoco.org/jacoco
20. Jayaraman, K., Harvison, D., Ganesh, V.: jFuzz: A concolic whitebox fuzzer for Java. In: NFM. pp. 121–125 (2009)
21. Jaygarl, H., Kim, S., Xie, T., Chang, C.K.: OCAT: Object capture-based automated testing. In: ISSTA. pp. 159–170 (2010)
22. Java PathFinder (JPF). http://babelfish.arc.nasa.gov/trac/jpf

23. Kähkönen, K., Launiainen, T., Saarikivi, O., Kauttio, J., Heljanko, K., Niemelä, I.: LCT: An open source concolic testing tool for Java programs. In: BYTECODE. pp. 75–80 (2011)
24. Khurshid, S., Păsăreanu, C.S., Visser, W.: Generalized symbolic execution for model checking and testing. In: TACAS. pp. 553–568 (2003)
25. Luckow, K., Dimjašević, M., Giannakopoulou, D., Howar, F., Isberner, M., Kahsai, T., Rakamarić, Z., Raman, V.: JDart: A dynamic symbolic analysis framework. In: TACAS. pp. 442–459 (2016)
26. McMinn, P.: Search-based software testing: Past, present and future. In: 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops. pp. 153–163 (2011)
27. Pacheco, C., Lahiri, S., Ernst, M., Ball, T.: Feedback-directed random test generation. In: ICSE. pp. 75–84 (2007)
28. Pasareanu, C.S., Rungta, N., Visser, W.: Symbolic execution with mixed concrete-symbolic solving. In: ISSTA. pp. 34–44 (2011)
29. Prasetya, I.S.W.B.: Budget-aware random testing with T3: Benchmarking at the SBST2016 testing tool contest. In: SBST. pp. 29–32 (2016)
30. Păsăreanu, C.S., Mehlitz, P.C., Bushnell, D.H., Gundy-Burlet, K., Lowry, M., Person, S., Pape, M.: Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In: ISSTA. pp. 15–26 (2008)
31. Rueda, U., Just, R., Galeotti, J.P., Vos, T.E.J.: Unit testing tool competition — round four. In: SBST. pp. 19–28 (2016)
32. Sakti, A., Pesant, G., Guéhéneuc, Y.G.: JTExpert at the fourth unit testing tool competition. In: SBST. pp. 37–40 (2016)
33. Sen, K., Agha, G.: CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In: CAV. pp. 419–423 (2006)
34. Sen, K., Marinov, D., Agha, G.: CUTE: A concolic unit testing engine for C. In: ESEC/FSE. pp. 263–272 (2005)
35. The SF110 benchmark suite. `http://www.evosuite.org/experimental-data/sf110` (July 2013)
36. Shafiei, N., Breugel, F.v.: Automatic handling of native methods in Java PathFinder. In: SPIN. pp. 97–100 (2014)
37. Soot: A Java optimization framework. `http://sable.github.io/soot`
38. Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Driller: Augmenting fuzzing through selective symbolic execution. In: NDSS (2016)
39. Tanno, H., Zhang, X., Hoshino, T., Sen, K.: TesMa and CATG: Automated test generation tools for models of enterprise applications. In: ICSE. pp. 717–720 (2015)
40. Thummalapenta, S., Xie, T., Tillmann, N., de Halleux, J., Su, Z.: Synthesizing method sequences for high-coverage testing. SIGPLAN Notices **46**(10), 189–206 (2011)
41. Tillmann, N., Halleux, J.d.: Pex—white box test generation for .NET. In: TAP. pp. 134–153 (2008)
42. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. Automated Software Engg. **10**(2), 203–232 (Apr 2003)
43. White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., Joglekar, A.: An integrated experimental environment for distributed systems and networks. SIGOPS Oper. Syst. Rev. **36**(SI), 255–270 (2002)