Detecting Assumptions on Deterministic Implementations of Non-deterministic Specifications

August Shi, Alex Gyori, Owolabi Legunsen, and Darko Marinov
Department of Computer Science
University of Illinois at Urbana-Champaign, USA
Email: {awshi2,gyori,legunse2,marinov}@illinois.edu

***Abstract*—Some commonly used methods have non-deterministic specifications, e.g., iterating through a set can return the elements in any order. However, non-deterministic specifications typically have deterministic implementations, e.g., iterating through two sets constructed in the same way may return their elements in the same order. We use the term *ADINS code* to refer to code that *A*ssumes a *D*eterministic *I*mplementation of a method with a *N*on-deterministic *S*pecification. Such ADINS code can behave unexpectedly when the implementation changes, even if the specification remains the same. Further, ADINS code can lead to *flaky tests*—tests that pass or fail seemingly non-deterministically.**

We present a simple technique, called NonDex, for detecting flaky tests due to ADINS code. We implemented NonDex for Java: we found 31 methods with non-deterministic specifications in the Java Standard Library, manually built non-deterministic models for these methods, and used a modified Java Virtual Machine to explore various non-deterministic choices. We evaluated NonDex on 195 open-source projects from GitHub and 72 student submissions from a programming homework assignment. NonDex detected 60 flaky tests in 21 open-source projects and 110 flaky tests in 34 student submissions.

I. Introduction

Non-deterministic specifications are not uncommon for many methods, including in the standard libraries of many programming languages. For example, the specification for the `Object#hashCode()` method in Java can return any integer. Non-deterministic specifications are not restricted to simple APIs. The order in which elements of a set are returned by an iterator is not-specified—it can be any order. The order in which entries in a SQL table are returned is also sometimes not specified—it depends on the query. Such specifications give implementers more freedom to develop various implementations for different goals, e.g., to optimize performance, while still satisfying the specification.

Even when specifications allow for non-determinism, typical implementations of such specifications are often deterministic, with respect to certain controlled sources. For example, `Object#hashCode()` could return the same integer (if one controls for all other sources, e.g., OpenJDK Java 8 could return a deterministic value on the first call if the underlying `random()` implementation in C is deterministic). The implementation of `HashSet` is such that iterating through the elements returns them in a deterministic order for one Java version, but that order can change between Java versions.

Code that *A*ssumes a *D*eterministic *I*mplementation of a *N*on-deterministic *S*pecification—which we call *ADINS* code—is often bad. Such ADINS code can behave unexpectedly when the implementation changes, even if the specification remains the same. For example, Java code that assumes a specific iteration order of a `HashSet`, e.g., that a `HashSet` with elements `1` and `2` will be always represented as a string `{1, 2}` rather than `{2, 1}`, is ADINS and not robust: the Java implementation of `HashSet` can change such that the iteration order of the elements changes and the string differs.

Unexpected behavior of ADINS code can lead to *flaky tests*, which are tests that seem to non-deterministically pass or fail. Flaky tests are bad as they can mask bugs (pass when there are bugs) or raise false alarms (fail when there are no bugs). A test that executes ADINS code can be flaky if it assumes that some values are deterministic even if they can change: when the assumptions hold, the test passes, but when the assumptions do not hold, the test may fail. Not all flaky tests are due to ADINS code, e.g., a test asserting that a file system contains `/tmp` could pass on one machine but fail on another. Flaky tests are emerging as an active research topic, with recent work on characterizing [25], detecting [2], [4], [10], [12], [14], [38], and avoiding [1], [22] flaky tests. However, no previous research investigated ADINS code as a cause for flaky tests.

While flaky tests are an important problem in software *practice* and *research*, we also encountered them in *teaching*. Typically, the teaching staff grades students' solutions to programming assignments using automated tests. These tests can be flaky, and as a result students with correct solutions may have failing tests, and students with incorrect solutions may have passing tests. We discuss more details from one recent course in Section IV-B. Besides educating people about flaky tests, how can we help practitioners in the real world and the students in our courses to detect more flaky tests faster?

We propose a simple technique, called NonDex, to detect flaky tests due to ADINS code. We implement NonDex for Java, but it can be easily generalized to any other language. In a nutshell, we identify 31 methods with non-deterministic specifications as discussed in Section III-A, wrote models for these methods to produce various non-deterministic choices, and use an execution environment that can explore various combinations of these non-deterministic choices. Our tool, called also NonDex, modifies a regular Java Virtual Machine (in particular, the OpenJDK JVM version b132) to randomly explore choices by rerunning the test suites multiple times from scratch with different random seeds.

We evaluated NonDex on two sets of programs: 195 open-source projects from GitHub and 72 student submissions from one homework assignment in our software-engineering course. We find NonDex to be highly effective at detecting flaky tests in both open-source projects and student submissions. NonDex detected 60 flaky tests in 21 of the 195 open-source projects. Because our experiments use some older project revisions, three of these tests were already fixed by the developers in the latest revision. (This fixing additionally confirms that flaky tests are important and that developers are willing to address them.) We confirmed that 57 tests are still present in the respective projects' latest revision. We leave it as future work to properly debug these tests and file bug reports. For student submissions, NonDex detected that 34 submissions, representing almost half of 72 considered, fail due to some ADINS code, with a total of 110 flaky tests detected. It is important to note that the homework assignment was designed a few years ago by a teaching assistant who had no knowledge of our research on flaky tests. We plan in the future to expose students to NonDex and teach them to better detect and avoid ADINS code and flaky tests.

This paper makes the following contributions:

- ⋆ **Problem.** We define the problem of ADINS code, identify it as a cause of flaky tests, and raise awareness about the problem of flaky tests in both software development practice and software engineering education.
- ⋆ **Technique and Implementation.** We propose a simple technique for detecting flaky tests caused by ADINS code and describe our non-deterministic models and a tool that embody this technique.
- ⋆ **Evaluation.** We evaluated our NonDex technique on 195 open-source Java projects and 72 student code submissions. NonDex detected 57 previously unknown flaky tests in open-source projects and three flaky tests that have been already fixed by the open-source software developers. NonDex also detected 110 flaky tests in student submissions.

II. Non-Deterministic Specifications

A non-deterministic specification allows for multiple implementations that can yield different outputs when executed with the same input; we consider "input" in a broad sense to include all interactions of code with its environment. For example, consider the method `File#list()` that returns a `String` array with names of all files and directories present in the directory on which the method was invoked. Its JavaDoc specification [6] states *"There is no guarantee that the name strings in the resulting array will appear in any specific order; they are not, in particular, guaranteed to appear in alphabetical order."* This specification allows implementations to return names in any order even when executed with the exact same input (the state of the file system), hence this specification *is* non-deterministic. In contrast, consider the method `File#exists()` that returns a `boolean` value indicating whether or not the file on which the method was invoked exists. This specification *is not* non-deterministic; while the returned value depends on the input (the state of the file system) and can be `true` or `false` on different machines, when executed on the same input, any implementation that conforms to the specification must return the same value.

```
class Book {
 String title;
 String author;
 String getStringRepresentation() { ... }
}
class BookTest {
 @Test
 public void testGetStringRepresentation() {
  Book b = new Book("book", "name");
  assertEquals("{\"title\":\"book\",\"author\":\"name\"}",
               b.getStringRepresentation());
 }
}
```

Fig. 1. Example flaky test simplified from student code

A. An Example Flaky Test

Code that (transitively) calls methods with non-deterministic specifications can be ADINS and lead to flaky tests. Figure 1 shows an example flaky test simplified from a student submission. The `Book` class has two fields, `title` and `author`. The method under test, `getStringRepresentation()`, uses a third-party JSON library that turns an object into a string. The test asserts that the result equals a hard-coded string that has the two fields in a particular order, first `title` and then `author`. However, the library uses a `HashMap` to store the mapping from fields to values, and iterates over this map to produce the resulting string. The iteration order over elements in a `HashMap` is not specified, so while this test can pass for one implementation, it can fail for another implementation that has `author` before `title`. NonDex can detect such tests.

B. Levels of Non-determinism

Some non-deterministic specifications can allow for multiple *levels* of non-determinism. Figure 2 presents an example: the class `HashSet` has a non-deterministic specification that can be (mis)interpreted in different ways. The JavaDoc specification [11] states *"[HashSet] makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time."*. Hence, the elements in the array returned by `HashSet#toArray()` can be in any order. The code first constructs an `Integer HashSet` object `s` with the elements `1` and `2` (lines 1–2). A deterministic implementation could return either of the two orders shown in the two assertions on lines 4 and 5, and one of the assertions should pass, while the other should fail.

Whether the other assertions pass or fail is more open to different interpretations of this non-deterministic specification. First, one could assume that two iterations on the same unchanged set object should yield the same order. However, the specification states that the order can vary "over time", which could mean that the order in which elements are returned can change from one invocation to another even for the same set. Hence, the assertion on line 7 may get a different order and fail. Second, one could assume that the order should not change if the set is only read. Hence, the assertion on line 10

```
1  Set<Integer> s = new HashSet<Integer>();
2  s.add(1); s.add(2);
3  Integer[] a = s.toArray();
4  // assertArrayEquals(a, new Integer[]{1, 2});
5  // assertArrayEquals(a, new Integer[]{2, 1});
6
7  // assertArrayEquals(a, s.toArray()); // differ from "a"?
8
9  s.contains(1); // observer calls on "s" may matter
10 // assertArrayEquals(a, s.toArray());
11
12 s.add(3); s.remove(3); // "s" modified and restored
13 // assertArrayEquals(a, s.toArray()); // differ from "a"?
14
15 Set<Integer> t = new HashSet<Integer>();
16 t.add(1); t.add(2); // "t" constructed same way as "s"
17 // assertArrayEquals(a, t.toArray()); // differ from "a"?
18
19 Set<Integer> u = new HashSet<Integer>();
20 u.add(3); u.add(4); // "u" with different elements
21 Integer[] b = u.toArray();
22 // assertEquals(a[0] < a[1], b[0] < b[1]); // order?
```

Fig. 2. Different levels of non-determinism may fail different assertions

could fail or pass. Third, one could assume that if a set is modified and then restored to its original state, the order in which the elements are iterated can change from that before the modification of the set. Hence, the assertion on line 13 could either pass or fail. Fourth, one could assume that two sets constructed in exactly the same way would yield the same order, but if that does not hold, the assertion on line 17 can pass or fail. Fifth, one could assume that elements are iterated in the order of addition; line 21 creates a new set using different elements but added in the same order as in set `s`, by their natural ordering. One could assume that both sets will be iterated in the same order—in which elements are added, or the natural order; depending on whether this assumption holds, the assertion on line 22 can pass or fail. (This final assumption is not unrealistic; `LinkedHashSet` indeed guarantees the iteration order over elements to be the same as that in which the elements are added [24].)

III. Technique

Our NonDex technique detects flaky tests due to ADINS code making deterministic assumptions on non-deterministic specifications. Section III-A describes how we identified several non-deterministic methods in the Java Standard Library. Section III-B describes the models we developed for those non-deterministic methods. Section III-C presents some implementation details of NonDex.

A. Identifying Non-deterministic Methods

Finding methods which have non-deterministic *specifications* is hard; in particular, one cannot easily look for non-deterministic *implementations* as individual implementations are deterministic most of the time. Rather, non-determinism occurs when non-deterministic specifications allow *multiple* implementations to behave differently from one another while still meeting the specification, even if each implementation is deterministic. For example, upgrading from Java 6 to Java 7 changed the order in which the Java `Reflection` API returned the list of methods in a class. JUnit uses the `Reflection` API for obtaining the list of methods to run.

TABLE I
Non-deterministic methods identified; methods marked * return a collection with a non-deterministic iteration order

Class method(s)	Kind
java.lang.Object#hashCode	*random*
java.util.HashMap#keySet*, values*, entrySet*	*permute*
java.util.concurrent.ConcurrentHashMap keySet*, values*, entrySet*, keys*, elements*	*permute*
java.io.File#list, listFiles, listRoots	*permute*
java.lang.Class getClasses, getFields, getDeclaredFields getConstructors, getAnnotations getMethods, getDeclaredConstructors getDeclaredMethods, getDeclaredClasses getDeclaredAnnotations	*permute*
java.lang.reflect.Method#getParameterAnnotations	*permute*
java.lang.reflect.Field#getDeclaredAnnotations	*permute*
java.text.DateFormatSymbols#getAvailableLocales	*permute*
java.text.BreakIterator#getAvailableLocales	*permute*
java.text.Collator#getAvailableLocales	*permute*
java.text.DecimalFormatSymbols#getAvailableLocales	*permute*
java.text.NumberFormat#getAvailableLocales	*permute*
java.text.DateFormat#getAvailableLocales	*permute*
java.text.DateFormatSymbols#getZoneStrings	*extend*

Thus, when run on Java 6, methods were returned in one order, but in a completely different order in Java 7. This seemingly innocuous change caused tests run by JUnit to fail [21] due to test-order dependencies [1], [2], [10], [14], [22], [38]. Finding non-deterministic methods solely from the code is infeasible; one must reason about the specification itself to find if a method can be non-deterministic. This makes it inherently hard for any static or dynamic analysis technique to find such non-deterministic methods from one implementation.

To find non-deterministic methods in the Java Standard Library, we first searched for methods that may have such specifications and then carefully reasoned from their JavaDoc to determine if their specifications are indeed non-deterministic. We used two queries, based on (1) JavaDoc keywords and (2) return types. Specifically, the first query searches through JavaDoc for the following keywords that could indicate non-deterministic specifications: *"order"*, *"deterministic"*, and *"not specified"*. The second query searches for all public methods that return arrays. These queries produced many false positives, e.g., because not every method that mentions *"order"* is non-deterministic, and some methods that return arrays can return elements in a specified order. Our search is definitely not complete, and we leave as future work to develop better approaches to find non-deterministic specifications.

After inspection, we found the non-deterministic methods summarized in Table I. We tabulate the class name, method name(s), and the kind of specification non-determinism. We found three kinds, which we call "*random*", "*permute*", and "*extend*". For *random*, the specific `int` returned by `Object#hashCode()` is not specified, so relying on it to return some specific value is ADINS. For *permute*, the specifications of some methods that return arrays or collections can have an unspecified order of elements. For *extend*, the specification of one method specifies just a lower bound on the length of the returned array but not the precise length.

For class `Object`, it is well known that `hashCode()` is non-deterministic. The inner class `HashMap$HashIterator` does not have a specified iteration order and can return the map's elements in any order; this inner class is exposed to the clients via some methods from Table I (`keySet()`, `entrySet()` and `values()`), so code that calls these methods can be ADINS. Moreover, `HashMap` is the underlying data structure for many other data structures, e.g., `HashSet`; we do not count separately the other non-deterministic methods, e.g., `HashSet#iterator()`, that could lead to ADINS code. However, changing one piece of code in `HashMap` can affect many types of objects. The specification for iterating through `ConcurrentHashMap` is similar to the specification for iterating through `HashMap`. The `File` class has multiple `list` methods that return an array of files in a given directory; the specification allows these arrays to be in any order. The classes `Class`, `Method`, and `Field` provide several reflection methods that return arrays of elements, e.g., an array of all methods in a class or an array of all annotations on a field; the specifications for most of these methods allow these arrays to be in any order. The classes in the package `java.text` return arrays of available locales and zone strings which can be in any order. Finally, the `DateFormatSymbols#getZoneStrings()` method returns an array of arrays, each of which has length at least five; these arrays are indeed of length five in Java 7 but of length seven in Java 8.

We also briefly explored an option of automatically finding non-deterministic methods in the Java Standard Library. We attempted to automatically generate tests that could show a behavior difference between Java 7 and Java 8. To that end, we used Randoop [28] to generate tests. We first instructed Randoop to generate tests for a large number of classes in the Java Standard Library on Java 8 and then ran the generated tests (that still compile) on Java 7. However, the tests (and assertions) that Randoop generated were unable to detect any changes in the behavior of the two Java versions. Even focusing Randoop on only one class, `HashMap`, did not generate (after one hour) a single test for Java 8 that would fail when run on Java 7. The reason is that the search space for `HashMap` is large, with 29 methods, and only a tiny ratio of method sequences in that space can show the difference between the two Java versions. In the end, we were able to generate tests that can reveal differences between Java 8 and Java 7 only after manually focusing Randoop to only four methods in the `HashMap`.

B. *Non-deterministic Models*

We developed models to explore potential non-determinism allowed by the specifications of the identified methods. NonDex has a model for each non-deterministic method, and each model has up to four different levels of non-determinism: FULL, ID, EQ, and ONE.

For non-deterministic methods of the *permute* kind, the FULL level is the most non-deterministic and each different

TABLE II
LEVELS THAT CAN FAIL (✓) ASSERTIONS FROM FIGURE 2

Assertion / Levels	4,5	7	10	13	17	22
FULL	✓	✓	✓	✓	✓	✓
ID	✓	-	-	✓	✓	✓
EQ	✓	-	-	-	-	✓
ONE	✓	-	-	-	-	-

method invocation on an unchanged object will return a different order. The ID level preserves the same order for objects that have the same identity (unchanged objects) but can return different orders for the same object if the object is ever modified. The EQ level preserves the order for objects that are equal (not necessarily the same object) but can change the orders of non-equal objects. The ONE level changes the order of an object only when it is first accessed (so it can have a different order) but this order does not change in the rest of the run. Table II shows which of the assertions in Figure 2 (referred to by line numbers in the column headers) can fail under the four levels that NonDex supports.

Either assertions 4 and 5 can fail on any of the levels, because all levels explore different orderings of the elements in the `HashSet` than the ones in the both assertions (recall that in a deterministic JVM, one of the assertions will always pass and one will always fail, whereas in our exploration the they can both pass, both fail or swap the order of pass/fail). Assertions 7 and 10 can only fail in FULL since they will only fail in levels that allow different orders on two successive invocations (including observer methods). Assertions 13 and 17 can fail in FULL and ID since these levels explore different orderings of objects based on their identity. Assertion 22 can fail in all levels except ID which would permute elements in the same way for both objects.

For non-deterministic methods of the *random* kind, when using the `Object` class, it should not be assumed that the `hashCode()` method returns a specific integer value. In particular, it should not be expected to return the same value across different runs. However, the returned value should be unique for an object in the same run. We model these potentially different values by randomizing the value returned by `hashCode()` on the initial invocation and then cache this value for future calls. For non-deterministic methods of the *extend* kind, we model the possibility that the lengths of arrays returned are increased non-deterministically on any invocation.

C. *Implementations of Models*

We implemented our NonDex technique for the Java programming language by modifying a regular Java Virtual Machine, in particular, the OpenJDK JVM version b132, which corresponds to Java 8. We downloaded the publicly available OpenJDK code, which consists of the C/C++ code that implements the core virtual machine, and the Java code for the Java Standard Library. We changed the Java Standard Library to add non-deterministic models for all methods listed in Table I except for `hashCode()`, for which we modified the C++ implementation to return different values. For each of the

```
1 class HashMap {
2  Node<K,V>[] table;  // internal table of key-value pairs
3  int modCount = ...  // stores modification count
4  class Node<K,V> { ... } // stores a key-value pair
5  ...
6  class HashIterator { // inner class of HashMap
7   Node<K,V> next;       // next entry to return
8   Node<K,V> current;    // current entry
9   int expectedModCount; // for fast-fail
10  int index;            // current slot
11
12  final boolean original_hasNext() {
13   return next != null; // original code
14  }
15  final Node<K,V> original_nextNode() {
16   // original code, advances "index" and "next"
17   ...
18  }
19  final void original_remove() {
20   // original code, can modify the entire "table"
21   ...
22  }
23  HashIterator() {
24   expectedModCount = modCount;
25   Node<K,V>[] t = table;
26   current = next = null;
27   index = 0;
28   if (t != null && size > 0) { // advance to first entry
29    do {} while
30     (index < t.length && (next = t[index++]) == null);
31   }
32 /** all (and only) the code below is NonDex extension **/
33   List<Node<K, V>> original = new ArrayList<>();
34   while (original_hasNext())
35    original.add(original_nextNode());
36   NonDex.shuffle(original, (NonDex.level == ID)
37      ? System.identityHashCode(HashMap.this) + modCount :
38    (NonDex.level == EQ) ? HashMap.this.hashCode() : 0);
39   NonDex_iter = original.iterator();
40  }
41
42  Iterator<Node<K, V>> NonDex_iter;
43  public final boolean hasNext() {
44   return NonDex_iter.hasNext();
45  }
46  final Node<K, V> nextNode() {
47   if (modCount != expectedModCount)
48    throw new ConcurrentModificationException();
49   current = NonDex_iter.next();
50   return current;
51  }
52  public final void remove() {
53   original_remove();
54  }
55 }
56 }
```

Fig. 3. Non-deterministic model for HashMap

methods whose output permutes, we call NONDEX and shuffle on the returned value in the library code.

Figure 3 shows the model we use for exploring different orderings when iterating over a `HashMap` object. The iteration is done using the inner class `HashIterator`. We kept the original code and renamed its methods with the prefix `original_`. The constructor, starting at line 33, computes the order that the original code would have normally returned, applies a shuffle depending on the NONDEX level, and stores the resulting order in an `Iterator` object. The `next` method returns the elements in the shuffled order. The `hasNext` method is based on the new order and delegates to the new `Iterator` object. The `remove` method just delegates to the original method that changes the table.

Figure 4 shows how NONDEX performs shuffling depending on the level. For FULL, NONDEX uses the same `Random`

```
class NonDex {
 static int level; // FULL, ID, EQ, or ONE
 int seed = ...;
 static Random full = new Random(seed);

 public static <T> List<T> shuffle(List<T> l, int v) {
  int size = l.size();
  Random rand = (level == FULL) ? full :  // Full
   (level == ID) ? new Random(seed + v) : // Same object
   (level == EQ) ? new Random(seed + v) : // Equal object
   (level == ONE) ? new Random(seed);     // Once
  for (int i = 0; i < size - 1; i++) {
   int s = rand.getNext(i, size);
   if (s == i) continue;
   T obj = l.get(i);
   l.set(i, l.get(s));
   l.set(s, obj);
  }
  return l;
 }
}
```

Fig. 4. Implementation of exploration

TABLE III
21 PROJECTS (OUT OF 195) WITH AT LEAST ONE FLAKY TEST

PID	PROJECT	SHA
P1	EsotericSoftware/reflectasm	455f612e
P2	EsotericSoftware/yamlbeans	2ccfbd9d
P3	JodaOrg/joda-time	07002501
P4	OryxProject/oryx	833c3fea
P5	Thomas-S-B/visualee	410a80f0
P6	apache/commons-cli	a0dcd6a0
P7	apache/commons-lang	fad946a1
P8	benas/easy-batch	4761ba5a
P9	bpsm/edn-java	c1d891d6
P10	caelum/vraptor	443cf0ed
P11	fernandezpablo85/scribe-java	0311a435
P12	geosolutions-it/geoserver-manager	a4268dda
P13	jknack/handlebars.java	83dd013a
P14	joel-costigliola/assertj-core	e8a696e8
P15	jscep/jscep	a224cc25
P16	junit-team/junit	1d63100e
P17	ning/org-json	9be37018
P18	qos-ch/slf4j	52fcbbe8
P19	sematext/ActionGenerator	10f4a3e6
P20	stickfigure/objectify	819eb72f
P21	versly/wsdoc	89480c5d

object to perform all shufflings, which means two consecutive shufflings of the same object can yield different orders,. For ID, NONDEX considers a value representing the identity of the object; for `HashMap`, this value is the sum of the identity hash code and the `modCount` field counting the number of modifications, which means that for the same object with the same `modCount`, NONDEX uses a fresh `Random` object with the same seed. Similarly, for EQ, NONDEX considers the value-based hash code of the object to produce a new `Random` object. For ONE, NONDEX always creates a fresh `Random` object using the same seed.

IV. EVALUATION

We evaluated our NONDEX technique on 195 open-source projects and 72 student submissions from a software-engineering course. Section IV-A describes our experiments with the open-source projects, and Section IV-B describes our experiments with the student code.

TABLE IV
FLAKY TESTS DETECTED IN OPEN-SOURCE PROJECTS

PID	TESTCLASS#TESTNAME	FULL	ID	EQ	ONE	CAUSE
P1	FieldAccessTest#testIndexSetAndGet	48	0	0	0	Class#getDeclaredFields
P2	GenericTest#testWrite	73	75	54	53	HashMap#entrySet
P3	TestDateTimeZone#testGetShortName	35	53	53	53	DateFormatSymbols#getZoneStrings
P4	TextUtilsTest#testJSONMap	51	52	60	53	HashMap#entrySet
P5	JPAExaminerTest#testFindAndSetAttributesManyT...	8	5	5	6	Class#getDeclaredMethods
P5	JavaSourceTest#testGetDependenciesOfType	12	12	12	4	Class#getDeclaredMethods
P6	OptionGroupTest#testToString	42	0	0	0	HashMap#values
P6	BugCLI162Test#testPrintHelpLongLines	51	55	55	53	HashMap#values
P7	MultilineRecursiveToStringStyleTest#boolArray	100	100	100	100	Class#getDeclaredFields
P7	...other 14 similar tests, total failures...	1296	1216	1215	1138	Class#getDeclaredFields
P7	FieldUtilsTest#testGetAllFields	100	0	0	0	Class#getDeclaredFields
P7	FieldUtilsTest#testGetAllFieldsList	100	0	0	0	Class#getDeclaredFields
P7	FieldUtilsTest#testGetFieldsWithAnnotation	56	51	53	45	Class#getDeclaredFields
P8	GsonRecordMarshallerTest#marshal	86	77	77	84	Class#getDeclaredFields
P8	JacksonRecordMarshallerTest#marshal	87	81	81	84	Class#getDeclaredFields
P8	XstreamRecordMarshallerTest#marshal	96	94	94	97	Class#getDeclaredFields
P9	PrinterTest#testPrettyPrinting	69	73	54	53	HashMap#entrySet
P10	XStreamSerializerTest#shouldSerializeCollection	41	48	45	52	Class#getDeclaredFields
P10	...other 13 similar tests, total failures...	736	709	737	764	Class#getDeclaredFields
P11	MapUtilsTest#shouldPrettyPrintMap	97	94	97	97	HashMap#entrySet
P12	GSLayerEncoder21Test#testMetadata	84	81	71	100	HashMap#entrySet
P13	TagTypeTest#collectSectionAndVars	100	100	100	100	HashMap#keySet
P14	Maps_format_Test#should_format_Map_containing...	76	50	62	53	HashMap#entrySet
P15	DefaultCertStoreInspectorTest#example	92	94	59	53	HashMap#keySet
P15	HarmonyCertStoreInspectorTest#example	95	96	59	53	HashMap#keySet
P16	MethodSorterTest#testJvmMethodSorter	100	0	0	0	Class#getDeclaredMethods
P17	TestSuite#testJSONStringerObject	79	77	83	84	Class#getFields
P18	EventLoggerTest#testEventLogger	100	0	0	0	Class#getDeclaredMethods
P19	BulkJSONDataESSinkTest#testGetBulkData	49	37	47	43	HashMap#entrySet
P19	JSONUtilsTest#testGetElasticSearchAddDocument	35	35	43	47	HashMap#entrySet
P19	XMLUtilsTest#testGetSolrAddDocument	36	43	43	47	HashMap#entrySet
P20	CollectionTests#testBasicSets	100	96	91	84	HashMap#keySet
P20	CollectionTests#testCustomSet	85	79	91	84	HashMap#keySet
P21	JaxRSRestAnnotationProcessorTest#stabilitySet...	71	86	51	53	HashMap#keySet
P21	SpringMVCRestAnnotationProcessorTest#stabilit...	76	75	51	53	HashMap#keySet
Flaky Tests Found		60	54	54	54	
Total Failures		4362	3744	3643	3590	
Min Failures		8	0	0	0	
Max Failures		100	100	100	100	

A. Experiments on Open-Source Projects

We evaluated NONDEX on 195 open-source projects. We selected these projects and their specific revisions from our previous studies with open-source projects [4], [23], [33]. All these projects are from GitHub [8], use Maven to build [26], and compile successfully using Java 8. For each project, we first ran NONDEX with 10 randomly generated seeds, using the FULL level. If any test failed with these 10 seeds, we examined it to determine what caused the failure. (As a side note, we found that manually inspecting these failures was rather challenging, and we leave it as future work to automate debugging test failures due to ADINS code.)

We detected 60 flaky tests in the 21 projects listed in Table III. We tabulate a short ID for ease of reference, the project name, and the project revision on which we ran NONDEX. For each project with a flaky test, we then reran that project's tests using NONDEX with 100 randomly generated seeds, using all non-deterministic levels. We obtained the number of times each flaky test fails out of the 100 seeds.

Table IV shows a partial list of the 60 tests that we examined. We tabulate the PID (from Table III), the name of the test class and its flaky test method, the number of failures detected for the each of the four levels, and the most likely non-deterministic method that causes the failures. The apache/commons-lang project has 14 tests similar to MultilineRecursiveToStringStyleTest#boolArray, and the caelum/vraptor project has 13 tests similar to XStreamSerializerTest#shouldSerializeCollection, so the two table rows show the total number of failures for each level across all 14 and 13 tests, respectively. Our evaluation started on older revisions of these projects, and three tests (GenericTest#testWrite, TestDateTimeZone#testGetShortName, and TagTypeTest#collectSectionAndVars) are already fixed on the current revisions of their respective projects.

Running NONDEX using the FULL level may introduce too much non-determinism, and one might initially consider some detected flaky tests to be false alarms. However, Table IV shows that only six flaky tests (FieldAccessTest#testIndexSet-

```
1 public class OptionGroupTest {
2  public void testToString() {
3   OptionGroup g1 = new OptionGroup();
4   g1.addOption(new Option(null, "foo", false, "Foo"));
5   g1.addOption(new Option(null, "bar", false, "Bar"));
6   if (!"[--bar Bar, --foo Foo]".equals(g1.toString())) {
7    assertEquals("[--foo Foo, --bar Bar]", g1.toString());
8   }
9   ...
10  }
11 }
12
13 public class OptionGroup ... {
14  Map<String, Option> om = new HashMap<String, Option>();
15  public OptionGroup addOption(Option option) {
16   om.put(option.getKey(), option);
17   return this;
18  }
19  public String toString() {
20   StringBuilder buff = new StringBuilder();
21   Iterator<Option> iter = getOptions().iterator();
22   buff.append("[");
23   while (iter.hasNext()) {
24    /* ... populate buff with the values in iter ... */
25    return buff.toString();
26   }
27  }
28 }
```

Fig. 5. Flaky test from apache/commons-cli

AndGet, OptionGroupTest#testToString, FieldUtilsTest#test-GetAllFields, FieldUtilsTest#testGetAllFieldsList, Method-SorterTest#testJvmMethodSorter, and EventLoggerTest#test-EventLogger) fail sometimes for the FULL level but not fail at all for any of the 100 randomly generated seeds for any of the other levels. The remaining 54 flaky tests are also detected by the other levels, suggesting that these are not false alarms.

For each flaky test, the table shows the number of seeds/runs on which it fails. For most flaky tests, the number of seeds is fairly high, with only 8–14 flaky tests failing for less than 50 seeds for each level (not counting flaky tests that have 0 failures for a given level), and only two of those tests fail less than 30 times for each level. These high numbers suggest that it is likely that a flaky test can be detected by running NONDEX with just a few seeds. Assume that the actual probability of a flaky test failing for a seed is equal to the percentage of seeds that fail out of the 100 seeds that were run. For example, if the probability of a flaky test failing for a seed is 30%, then the probability of the flaky test *not* failing for 10 different, independent seeds is $(1-0.3)^{10} = 0.028$; in other words, there is a less than 3% chance of NONDEX missing to detect that flaky test running with 10 seeds. In the most extreme case we detected, in the Thomas-S-B/visualee project, the expected probability of a flaky test failing for a seed in the FULL level is only 8%, so the chance of NONDEX missing this flaky test running with 10 seeds is $(1 - 0.08)^{10} = 0.434$. Even in this case, there is more than 50% chance of detecting such a flaky test with 10 seeds, despite the chance of it failing for any one seed being rather low.

In summary, a developer using NONDEX to detect flaky tests may not need to run with many seeds and can still have some confidence that NONDEX does not miss to detect any flaky tests. Therefore, we recommend that NONDEX by default be run for 10 seeds while increasing the level of non-determinism, from ONE to FULL. Because the common threats to validity apply to our study, our results may not generalize to other projects or flaky tests. Of particular concern is that our experiments could have missed some flaky tests even in the projects that we ran with 10 seeds. If some test fails infrequently, it may be missed; there might be many such tests that NONDEX missed, so we could not have even studied them in more detail. In the future, we plan to evaluate more systematic exploration to check whether this is indeed the case.

```
1 public class MapUtilsTest {
2  @Test public void shouldPrettyPrintMap() {
3   Map<Integer, String> map = new HashMap<>();
4   map.put(1, "one"); map.put(2, "two");
5   map.put(3, "three"); map.put(4, "four");
6   assertEquals(
7    "{ 1 -> one , 2 -> two , 3 -> three , 4 -> four }",
8    MapUtils.toString(map));
9  }
10 }
11
12 public class MapUtils {
13  public static <K,V> String toString(Map<K,V> map) {
14   ...
15   StringBuilder result = new StringBuilder();
16   for(Map.Entry<K,V> entry : map.entrySet()) {
17    result.append(String.format(", %s -> %s ",
18      entry.getKey().toString(),
19      entry.getValue().toString()));
20   }
21   return "{" + result.substring(1) + "}";
22  }
23 }
```

Fig. 6. Flaky test from fernandezpablo85/scribe-java

We next discuss in more detail three flaky tests detected by NONDEX in open-source projects.

1) Overly Non-deterministic Level: A case where the FULL level detects a flaky test that is never detected for any other level is OptionGroupTest#testToString from the apache/commons-cli project. Figure 5 shows that flaky test. Lines 4 and 5 add some *options* to the `OptionGroup g1`. `OptionGroup` stores options in a `HashMap` (line 16), and its `toString()` method (lines 19–27) iterates over this map. The developer realized that the iteration order over the `HashMap` is not guaranteed, so lines 6 and 7 check that the result of calling `toString()` on `g1` is either of the two hard-coded strings. However, `toString()` is invoked twice, and in the FULL level, NONDEX can reshuffle the order differently for the two invocations, causing the assertion to potentially fail. The developer made a reasonable assumption that calling `toString()` on the same, unchanged object twice returns the same string both times; we see that the other levels of NONDEX never flag this test as flaky. Nevertheless, the test could be still changed to call `toString()` only once and then to assert that it returns one of the two possible values.

2) Example New Flaky Test: We detected 57 flaky tests that are not fixed on the current revision of the projects, and Figure 6 shows one such flaky test, MapUtils-Test#shouldPrettyPrintMap from the fernandezpablo85/scribe-java project. The test (lines 3–5) makes and populates a `HashMap` and then compares the result of calling `MapUtils#toString()` with a hard-coded string (lines 6–8). However, `MapUtils#toString()` calls `entrySet()` on

```
1 public class DefaultNameProvider implements NameProvider {
2  public String getName(...) {
3   String[] nameSet = getNameSet(...);
4   return nameSet[0];
5  }
6  private synchronized String[] getNameSet(...) {
7   String[][] z = DateTimeUtils.getDateFormatSymbols(...).
        getZoneStrings();
8   String[] setEn = null;
9   ...
10  for (String[] s : z) {
11   if (s != null && s.length == 5 && id.equals(s[0])) {
12    setEn = s;
13    break;
14   }
15  }
16  ...
17 }
18 }
19
20 public class TestDateTimeZone extends TestCase {
21  public void testGetShortName() {
22   DateTimeZone zone = DateTimeZone.forID(...);
23   assertEquals("BST", zone.getShortName(...));
24   ...
25  }
26 }
27
28 public abstract class DateTimeZone ... {
29  public String getShortName(...) {
30   String name;
31   NameProvider np = getNameProvider();
32   if (np instanceof DefaultNameProvider) {
33    name = ((DefaultNameProvider) np).getShortName(...);
34   }
35   ...
36   return name;
37  }
38  private static NameProvider getDefaultNameProvider() {
39   NameProvider nameProvider = null;
40   ...
41   if (nameProvider == null) {
42    nameProvider = new DefaultNameProvider();
43   }
44   return nameProvider;
45  }
46 }
```

Fig. 7. Flaky test from JodaOrg/joda-time

its input `Map`, and the order of iteration is not fixed, so the assertion on lines 6–8 can sometimes fail. More precisely, it fails in all but one of the 4! orderings, i.e., in about 96% of cases, as also obtained in our experiments.

3) Example Fixed Flaky Test: We next describe a flaky test that NONDEX detected when run on an older revision of the JodaOrg/joda-time project; the test has been fixed since then. Figure 7 shows TestDateTimeZone#testGetShortName and the relevant portions of the code under test. The call to `getShortName()` on line 23 eventually leads to a call to the `DefaultNameProvider#getNameSet()` method defined on lines 6–17. The problem is the guard condition, `s.length == 5` on line 11. In Java 7, the call to `DateFormatSymbols#getZoneStrings()` on line 7 returned each array element of `z` of exactly length five. However, the specification of that method only guarantees that each element of `z` has length of *at least* five. In fact, in Java 8, the implementation changed such that each array element has length exactly seven, which still satisfies the specification but is different from what was the case in Java 7. This change in the implementation revealed the developer's reliance on the length of the elements of `z`. NONDEX was able to detect this on an older revision of the code, and the developers have since fixed this problem by changing checks such as the one shown on line 11 to be `s.length >= 5` instead.

B. Experiments on Student Code

We also evaluated NONDEX on 72 student submissions for a programming assignment. We first describe the assignment that the students were supposed to do. We then describe how we set up our experiments for the student submissions. We finally describe high-level results concerning our findings of running NONDEX on the student submissions.

1) Assignment: The assignment asked the students to create a simple library-management application[1]. The students were expected to write both code that implements such an application and unit tests using JUnit [20] to test the different components of the application.

The teaching staff provided the students some skeleton code outlining the basic expected components of the application. The application should represent a library containing books which can be organized into collections. The `Book` class represents a book and has only two fields, a title and an author, both represented by `String` objects. This `Book` class extends the abstract class `Element`. The `Collection` class represents a collection of such `Element`s that are stored in a `List`. Furthermore, the `Collection` class also extends `Element`, so a `Collection` is allowed to contain other `Collection` objects, creating a hierarchy that illustrates the composite design pattern [7]. Finally, at the top level, there is a `Library` class that can hold a `List` of `Collection` objects.

Students were expected to implement several methods and constructors for each of these classes. We discuss those that are most relevant for this study. For both `Book` and `Collection`, students must implement a method `getStringRepresentation()` that returns `String` representations of objects of those classes. Given such a string representations, students must implement a constructor for `Book` that takes the string representation and constructs the corresponding `Book` object. For `Collection`, students must similarly implement a static method `restoreCollection` that takes a string representation of a `Collection` and constructs the corresponding `Collection` object. For `Library`, students must implement (1) the constructor that takes a file containing string representations of a sequence of `Collection` objects and constructs the corresponding `Library` and (2) the method `saveLibraryToFile()` that writes out the `Library` to a file.

Along with the skeleton code and implementation requirements, the teaching staff made further restrictions and suggestions. First, the students' code must build successfully on a common platform used by the entire class. This platform uses OpenJDK Java 7, so students' code must also compile to Java 7 bytecode and run successfully using the OpenJDK Java 7 JVM. Students must also write tests for each of the

[1]This library-management assignment was first created three years ago and has been minimally updated by different teaching staff members over the years; this year's iteration of the assignment was updated by two teaching assistants who were not involved in this study.

TABLE V
FLAKY TESTS DETECTED IN STUDENT SUBMISSIONS

	FULL	ID	EQ	ONE
Flaky Tests Found	110	88	34	34
Total Failures	8159	6785	2031	1827
Min Failures	37	0	0	0
Max Failures	100	100	81	78

three classes they implement, with at least nine tests for the entire application. Finally, the staff strongly encouraged the students to use some third-party library to handle the pretty-printing/parsing of objects to/from strings, as the `Library` can potentially have complex structures involving deeply nested `Collection` objects. However, the staff did not restrict the students to a specific third-party library, so the students chose whatever library they felt comfortable with. Many used various libraries for JSON or XML.

2) Experimental Setup: For our evaluation on student code, we started from the 89 submissions that built successfully (both compiled and had all tests pass) on the common platform that uses OpenJDK Java 7. With these 89 submissions, we ran the tests on another platform that is exactly the same as the platform provided to the students, except this other platform uses OpenJDK Java 8 instead. By running the students' tests against their own code on a platform using Java 8, we already detected some students' tests to be flaky as they assumed specific behavior of the libraries (either the Java Standard Library or the third-party libraries used), and most likely failing due to the presence of ADINS code.

Running the students' tests in this Java 8 environment, we found 17 submissions that fail. In fact, in the past, running in multiple environments (e.g., on Linux virtual machines and on Mac and Windows laptops from teaching assistants) was the only approach that we could use to detect (some) flaky tests. Using NONDEX, we can have a more thorough detection of flaky tests; even if some tests pass on both Java 7 and Java 8, it does not imply they do not contain any ADINS code that could fail on some future Java 9 (or even on Java 8 on another OS or by another JVM provider, say, IBM). We therefore focus the rest of our evaluation on the remaining 72 submissions.

3) Results: We ran the student submissions using our NONDEX tool in all four non-deterministic levels and for 100 randomly generated seeds. (The tests from students submissions run much faster than the open-source projects, so we could immediately use 100 seeds.) NONDEX detected 34 student submissions with at least one flaky test. In total, NONDEX detected 110 flaky tests. Table V summarizes the results. We tabulate the number of flaky tests detected in each level (up to 110), and the total, minimum, and maximum number of the 100 seeds that cause a failure for one of those flaky tests in each level. We elide detailed results for each individual test as in Table IV because there are too many tests.

From the table, we see that the FULL level detects the most flaky tests, followed by ID, and then by EQ and ONE, which both detect the same number of flaky tests. Unlike for open-source projects where all three partial levels behaved the same (either all three had at least one failure or all three had no failure), for student submissions, ID detected more flaky tests than either EQ or ONE that detected exactly the same flaky tests. Considering the total number of failures, like for open-source projects, we see that the FULL level detects more failures than the ID level, followed by the EQ level and finally by the ONE level.

Running NONDEX even with 100 different seeds may not detect all flaky tests, as the random choices explored can miss some cases that would cause a test to fail. To more systematically explore these tests, we used Java PathFinder (JPF) [17], [34]. JPF provides a specialized JVM, implemented in Java, that can explore all non-deterministic choices. However, JPF cannot handle all Java code out-of-the-box. In particular, it cannot handle code that depends on native methods, such as those in `Gson` or `XStream` that students used.

To use JPF, we changed our model for `HashMap` to not make some random choices but rather systematically explore all choices; specifically, considering Figure 4, on line 13, we replaced `rand.getNext(i, size)` with `Verify.getInt(i, size - 1)`, where the JPF method `Verify.getInt` explores all possible values between the bounds `i` and `size - 1`, inclusive. Hence, JPF shuffles each `List` to explore all permutations. Because JPF can systematically explore all choices until either the test fails or all possible permutations execute and the test does not fail, JPF will not miss any test that depends on some ordering for the elements in a `HashMap`. Because JPF cannot handle most file system operations, we could not run the tests for the `Library` class which need to read/write some file from/to disk. Even without running the tests for `Library`, there were still many cases where JPF throws some internal exception because of functionality that is not yet implemented in JPF. As a result, we ignored the 22 submissions that do not work with JPF and only ran the tests for `Book` and `Collection`.

Using JPF, we detected 22 submissions where exploring permutations on `HashMap` iterator order causes a test to fail. In total, JPF detected 51 flaky tests. We compared these results with those obtained from running NONDEX on our modified JVM, using only the non-deterministic methods in `HashMap`, and running only tests for `Book` and `Collection`. We found that NONDEX detected all tests detected by JPF, increasing confidence that 100 seeds suffice for student submissions.

4) Discussion: In the 17 cases where the students' tests fail just by switching from Java 7 to Java 8, the flaky tests check the functionality of the methods that get the string representation of a `Book` or a `Collection` object. The tests generally construct some `Book` or `Collection` objects and assert that the return of the method that gets the string representation matches some hard-coded string value. In all but one of these submissions, students either directly use a Java `HashMap` as part of their implementation for constructing a string representation, or they use a third-party library (e.g., JSON in Java [18] or JSON.Simple [19]) where the serialization is backed by a Java `HashMap`. The assertions against the hard-coded strings succeed in Java 7 because the order remains consistent across different runs of the JVM, but in Java 8, the underlying implementation of `HashMap` changed such that the iteration order can differ from that of Java 7. The one remaining failing submission uses an XML serialization library (XStream [37]) to construct a string representation of a `Collection` object, but the order of the declared fields for a class is also not guaranteed, so the comparison with a hard-coded string value here once again fails in this later version of Java. In summary, all these 17 submissions have ADINS code and fail due to relying on some assumed order that is not guaranteed to hold.

In the student submissions that do not fail on Java 8, NONDEX detected more flaky tests that fail due to the non-determinism in the ordering provided by the iterator for a `HashMap`. As with the tests that fail on Java 8, these flaky tests generally construct `Book` and `Collection` objects and assert their string representation to be equal to a hard-coded string. Similar to some cases in the open-source projects, some failures are due to "too much" non-determinism in the orderings, e.g., when a test calls `getStringRepresentation()` on an object and then compares the string against another call of `getStringRepresentation()` of an equal object rather than asserting the string to be the same as a hard-coded string. In such a case, the FULL or ID level would shuffle both calls to `getStringRepresentation()` and potentially end up failing the assertion where the other two levels do not fail. Moreover, the FULL level also detects as flaky some cases that depend on the field ordering, which other levels never detect.

V. RELATED WORK

Detecting problems due to wrong assumptions that developers make about specifications and implementations has been explored in other domains. For example, Jin et al. [15] reported how wrong assumptions about code can lead to performance bugs, in particular, they find the second most common reason for the introduction of performance bugs to be that "developers misunderstand the performance feature of certain functions". NONDEX does not target performance bugs but helps detect another class of bugs that are due to specification misunderstanding. As another example, from a security perspective, Wang et al. [35] propose a technique to analyze implicit assumptions that are necessary for the secure use of libraries. Their work involves building models of methods which are then used to find bugs in software that fail to meet these implicit assumptions, finding serious security vulnerabilities in the process. Their techniques are mostly *static*, while NONDEX uses a *dynamic*, randomized exploration of methods with non-deterministic specifications.

Randomness has been applied in different contexts to detect bugs, with many of these applications for concurrent code. For example, Eytani et al. [5] developed a tool that monitors shared variable accesses and applies random context switching when shared variables are accessed in order to trigger bugs in concurrent code; Parizek and Kalibera [29] used an abstract environment in software model checkers that randomly selects sequence of method calls in each thread to detect bugs in concurrent programs; and Joshi et al. [16] applied randomness in thread scheduling to create resource deadlocks in multi-threaded programs. Moreover, JPF can also control thread schedules to potentially explore all paths in the code [34]. In contrast, NONDEX focuses on sequential code and exploration of non-deterministic specifications.

Non-determinism has been also studied for various other domains. For example, for map-reduce programs, Xiao et al. [36] studied non-determinism that arises due to non-commutative reducers and found many bugs due to non-commutative reducers that make assumptions on the order of input data rows. For GUI code, Memon and Cohen [27] showed various factors that may cause non-determinism and hence impact the results of analyses and experiments based on GUI software. For state machines, testing conformance of deterministic implementations against non-deterministic specifications has a long history [13], [30]–[32]. More recently, Cook and Koskinen [3] aim to find restrictions on non-deterministic value-choices using a CEGAR loop; they apply their technique to examples drawn from real code. NONDEX explores non-determinism in the context of abstract data-type specifications using concrete exploration of real code.

VI. CONCLUSIONS AND FUTURE WORK

Non-deterministic specifications are good because they allow implementers to provide various implementations. However, non-deterministic specifications are bad because they can result in seemingly random failures. In particular, ADINS code that assumes a deterministic implementation of non-deterministic specification is susceptible to failures that arise from changing implementations. Tests that depend on ADINS code can become flaky tests that seemingly non-deterministically pass or fail. We proposed a novel NONDEX technique to detect flaky tests due to ADINS code. NONDEX detected many flaky tests in both larger, open-source projects and small-sized student code submissions.

In the future, we plan to investigate how to automate debugging of failures that NONDEX reports. While it is good that NONDEX detected many flaky tests, manually investigating them turned out to be hard. Interestingly, we found that only 7 of the 31 non-deterministic models were the likely causes for all detected flaky tests. Considering our experience with Java 7 and Java 8, we also plan to study advanced approaches to find behavior differences between different JVM implementations, e.g., using automated test generation that previously revealed differences in IDEs, Java compilers, and JPF [9]. Finally, to help developers avoid misunderstandings of non-determinism, we envision that code could have some determinism annotations, e.g., `@Unordered` to specify methods that provide no guarantee on the order of elements. Overall, ADINS code seems to open an interesting new line of research.

Acknowledgments: We thank Lamyaa Eloussi, Wajih Ul Hassan, Nicholas Lu, Xinyue Xu, and Tifany Yung for help in this project. This research was partially supported by the NSF Grant Nos. CCF-1012759, CCF-1409423, CCF-1421503, and CCF-1439957.

References

[1] J. Bell and G. Kaiser, "Unit test virtualization with VMVM," in *ICSE*, 2014.

[2] J. Bell, G. Kaiser, E. Melski, and M. Dattatreya, "Efficient dependency detection for safe Java test acceleration," in *ESEC/FSE*, 2015.

[3] B. Cook and E. Koskinen, "Reasoning about nondeterminism in programs," in *PLDI*, 2013.

[4] L. Eloussi, "Detecting flaky tests from test failures," Master's thesis, UIUC, 2015.

[5] Y. Eytani, E. Farchi, and Y. Ben-Asher, "Heuristics for finding concurrent bugs," in *IPDPS*, 2003.

[6] "File - list JavaDoc," http://docs.oracle.com/javase/8/docs/api/java/io/File.html#list--.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[8] "GitHub," https://github.com/.

[9] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov, "Test generation through programming in UDITA," in *ICSE*, 2010.

[10] A. Gyori, A. Shi, F. Hariri, and D. Marinov, "Reliable testing: Detecting state-polluting tests to prevent test dependency," in *ISSTA*, 2015.

[11] "HashSet JavaDoc," https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html.

[12] K. Herzig and N. Nagappan, "Empirically detecting false test alarms using association rules," in *ICSE SEIP*, 2015.

[13] R. Hierons and M. Harman, "Testing conformance of a deterministic implementation against a non-deterministic stream X-machine," *Theoretical Computer Science*, 2004.

[14] C. Huo and J. Clause, "Improving oracle quality by detecting brittle assertions and unused inputs in tests," in *FSE*, 2014.

[15] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," in *PLDI*, 2012.

[16] P. Joshi, C.-S. Park, K. Sen, and M. Naik, "A randomized dynamic program analysis technique for detecting real deadlocks," in *PLDI*, 2009.

[17] "JPF home page," http://babelfish.arc.nasa.gov/trac/jpf/.

[18] "JSON in Java," http://www.json.org/java/.

[19] "JSON-Simple," https://code.google.com/p/json-simple/.

[20] "JUnit," http://junit.org/.

[21] "JUnit 4.11 - What's new? Test execution order," http://randomallsorts.blogspot.com/2012/12/junit-411-whats-new-test-execution-order.html.

[22] W. Lam, S. Zhang, and M. D. Ernst, "When tests collide: Evaluating and coping with the impact of test dependence," University of Washington Department of Computer Science and Engineering, Tech. Rep., 2015.

[23] O. Legunsen, D. Marinov, and G. Rosu, "Evolution-aware monitoring-oriented programming," in *ICSE NIER*, 2015.

[24] "LinkedHashSet JavaDoc," http://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashSet.html.

[25] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *FSE*, 2014.

[26] "Maven Surefire plugin," http://maven.apache.org/surefire/index.html.

[27] A. M. Memon and M. B. Cohen, "Automated testing of GUI applications: Models, tools, and controlling flakiness," in *ICSE*, 2013.

[28] C. Pacheco, S. Lahiri, M. Ernst, and T. Ball, "Feedback-directed random test generation," in *ICSE*, 2007.

[29] P. Parizek and T. Kalibera, "Efficient detection of errors in Java components using random environment and restarts," in *TACAS*, 2010.

[30] A. Petrenko, N. Yevtushenko, and G. V. Bochmann, "Testing deterministic implementations from nondeterministic FSM specifications," in *IWTCS*, 1996.

[31] A. Petrenko, N. Yevtushenko, A. Lebedev, and A. Das, "Nondeterministic state machines in protocol conformance testing," in *IWPTS*, 1994.

[32] T. Savor and R. E. Seviora, "Supervisors for testing non-deterministically specified systems," in *ITC*, 1997.

[33] A. Shi, T. Yung, A. Gyori, and D. Marinov, "Comparing and combining test-suite reduction and regression test selection," in *FSE*, 2015.

[34] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs," *ASE Journal*, 2003.

[35] R. Wang, Y. Zhou, S. Chen, S. Qadeer, D. Evans, and Y. Gurevich, "Explicating SDKs: Uncovering assumptions underlying secure authentication and authorization," in *USENIX*, 2013.

[36] T. Xiao, J. Zhang, H. Zhou, Z. Guo, S. McDirmid, W. Lin, W. Chen, and L. Zhou, "Nondeterminism in MapReduce considered harmful? An empirical study on non-commutative aggregators in MapReduce programs," in *ICSE SEIP*, 2014.

[37] "XStream," http://x-stream.github.io/.

[38] S. Zhang, D. Jalali, J. Wuttke, K. Muslu, M. Ernst, and D. Notkin, "Empirically revisiting the test independence assumption," in *ISSTA*, 2014.