Exploring Underdetermined Specifications using Java PathFinder

Alex Gyori[1], Ben Lambeth[1], Sarfraz Khurshid[2], and Darko Marinov[1]
[1]Department of Computer Science, University of Illinois at Urbana-Champaign
[2]Department of Electrical and Computer Engineering, The University of Texas at Austin
{gyori,blambet2,marinov}@illinois.edu, khurshid@utexas.edu

ABSTRACT

Some Java libraries have underdetermined specifications that allow more than one correct output for the same input, e.g., an output array may have its elements in any order. While such specifications have a number of advantages (e.g., a library can change while still satisfying the specification), the non-determinism inherent in underdetermined specifications can lead to failures in client code that erroneously assumes behaviors based on the library implementation instead of only the specification. Our recent work introduced the NonDex approach for detecting such erroneous assumptions by checking client code against models of library methods, which encode all behaviors allowed by the specifications.

We present NonDex for JPF, which includes JPF models for 11 methods from the Java standard library (i.e., all methods that JPF supports from the current methods in NonDex). We use these models to systematically explore state spaces of 46 tests from student homework submissions. Our experiments show several interesting results, which provide new insights into the complexity of exploring the behaviors of code that uses underdetermined APIs and the structure of state spaces that arise in the exploration, and provide basis for future work on better detecting faults in tests that invoke underdetermined APIs as well as developing tool support for writing and maintaining more robust test suites.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging

Keywords: Underdetermined specifications, NonDex, systematic exploration, Java PathFinder

1. INTRODUCTION

Specifications, which state key properties of expected program behaviors, play a useful role in the development and maintenance of reliable software systems. Most specifications are written only informally, e.g., as Javadoc comments in the standard Java library. Many specifications are deterministic, i.e., they permit only one output for some given input. However, some specifications are *underdetermined*, i.e., they permit a number of different outputs for the same input, allowing different implementation-level behaviors.

Underdetermined specifications offer one key advantage: they allow developers flexibility when modifying their implementations. Such modifications should not break the client code that uses an implementation as a library, *provided* that the client code relies only on the library specification and not on the details of the library implementation. However, the non-determinism inherent in underdetermined specifications can result in software bugs when the client code erroneously assumes the implementation to continue to have some deterministic behavior even though it implements an underdetermined specification. Such erroneous assumptions are surprisingly common, e.g., we found 60 bugs in open-source projects and 110 bugs in student homework submissions as reported in our ICST 2016 paper [14] which introduced the NonDex approach for finding erroneous assumptions on underdetermined specifications.

The key idea of NonDex is to check the client code against not just one behavior of the library but against a number of different behaviors—all of which are valid with respect to the underdetermined specification that the library implements—thereby finding subtle bugs in the client code. These bugs are triggered by particular non-deterministic choices in library and can be hard to find otherwise. Some of these bugs may manifest when changing the platform (e.g., a student's homework submission can always pass on their Mac laptop but fail on the Linux grading environment), while some may not manifest for any current implementation of the library but could cause failure for some future implementations. These problems are important to developers. For example, we found out that Google[1] has a tool similar to NonDex (but it handles only one method, the `HashMap` iterator), and developers of open-source projects are willing to accept fixes for these bugs, e.g., we opened 13 pull requests of which 12 were accepted and only 1 rejected [1]. The developer of Checkstyle even integrated our NonDex tool into the project's continuous-integration setup (both into the `pom.xml` build configuration file and into the `.travis.yml` configuration file for running on the Travis CI) [2].

To provide NonDex functionality, we proceed in three steps: (1) find a number of methods with underdetermined specifications, (2) build models that encode all behaviors of the specification, and (3) run these models in an appropriate execution environment to explore multiple behaviors. For example, the method `getDeclaredFields` from the class

[1]Private communication with John Micco.

© 2016 ACM. ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

`java.lang.Class` returns the fields in an unspecified order, and we can build a model that encodes all $k!$ permutations of k fields. (Section 3 shows this example model.)

Java PathFinder (JPF) is a popular execution environment to explore different behaviors of Java models. JPF is a sophisticated Java virtual machine (JVM), written itself in Java, that provides out-of-the-box systematic exploration with advanced features such as state comparison. However, JPF suffers from the problem that it cannot run all Java code, in particular code that uses native methods that are implemented in C/C++ for a regular JVM and need to be re-implemented in Java for JPF. For this reason, our initial experiments [14] used both a NonDex prototype that explores models in a regular JVM and can work with all Java code but does not provide systematic exploration and a simple NonDex implementation in JPF that supported *only one method* (the `HashMap` iterator). Since then we have built a more robust tool for a regular JVM [6,11].

This paper presents our extended work on NonDex for JPF. We develop JPF models for 10 new methods (i.e., all methods that JPF supports from the current methods in NonDex for a regular JVM), and we use these models to *systematically explore* state spaces of 46 tests from student homework submissions. (JPF cannot easily handle the open-source projects in which we found bugs because many of them use I/O operations, advanced reflection, or other methods not supported by JPF.) Inspired by prior work that studied properties of state-space graphs arising in explicit-state model checking [5,12], we study the properties of state-space graphs from NonDex explorations.

Our analysis shows several interesting results. First, the state-space graphs even for simple sequential code can become large when considering underdetermined specifications. Second, when a test fails due to one of the underdetermined specifications we consider, the probability that a randomly selected execution leads to a test failure (rather than to a test success) is high, at least 50%. Third, JPF state matching finds a non-trivial amount of non-deterministic choices due to underdetermined specifications are local and do not create globally non-deterministic states. Fourth, the number of critical choice points that determine the test outcome (failure or success) is relatively small, indicating the key points to focus on during debugging. Fifth, certain orderings are more likely to lead to test failures, and in the future we plan to evaluate specialized heuristics to prioritize orderings.

2. MOTIVATING EXAMPLE

Fig. 1 shows some test simplified from a student homework in a Software Engineering class taught at the University of Illinois. The students were asked to write code for a `Book` class and tests for their code. The method `testGetStringRepresentation1` aims to test that the `Book` object produces a correct string representation: the test checks that the round-trip from the string representation of a `Book` to a `Book` object and back to its string representation yields the same `String` result used to construct the `Book` object. The homework was designed several years ago by a teaching assistant with no knowledge of this research.

The problem with this test is it assumes the order of the fields in the JSON representation of the `Book` object to be the same every time, either `{author="Die...", title="Cos..."}` or `{title="Cos...", author="Die..."}`. This assumption is wrong; it is not supported by the JSON specification: the

```
public class BookTest {
 private String toJSON(String s)
   throws JSONException {
  JSONObject obj = new JSONObject();
  String[] info = s.split(",");
  obj.put("author", info[0].trim());
  obj.put("title", info[1].trim());
  return obj.toString();
 }
 @Test
 public void testGetStringRepresentation1()
   throws JSONException {
  Book book =
   new Book(toJSON("Diego et al., Costization"));
  assertEquals(
   toJSON("Diego et al., Costization"),
   book.getStringRepresentation());
 }
}
```

Figure 1: Example test that fails due to a underdetermined specification

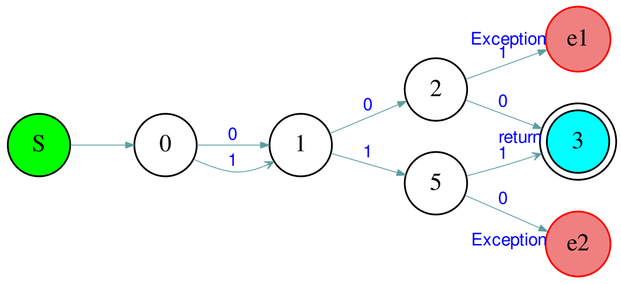

Figure 2: State-space graph for the example test

`author` and `title` could appear in any order in the resulting string. In fact, the data structures used to implement the underlying `JSONObject` do not guarantee the order assumed by this test. Specifically, a `HashMap` is used to store a mapping between field names and their values, and the code in `JSONObject` (not shown here) iterates the `HashMap` to produce the `String` representation. The specification of the `HashMap` explicitly states: *"This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time."* [7]. Code making such wrong assumptions, unsupported by the specification, is brittle because whenever the library changes, the assumptions may stop holding, and the code can break [8,9].

Our NonDex technique finds assumptions on certain APIs by exploring different behaviors permitted by the specification. If exploring these different behaviors triggers a failure, it indicates that the code makes some wrong assumption on the API. In the example in Fig. 1, NonDex would explore different orders of iteration for the underlying `HashMap` of each `JSONObject`. Note that it is necessary to explore an execution where the *two* iteration orders for the two `JSONObject` objects differ, i.e., `{author="Die...", title="Cos..."}` and `{title="Cos...", author="Die..."}`. We manually create models for APIs based on their specifications, and we use JPF to explore these models for all allowed behaviors to find wrong assumptions.

Fig. 2 shows the entire state-space graph resulting from the JPF's exploration of different behaviors of APIs with underdetermined specifications. In the execution of the test `testGetStringRepresentation1`, the program executes three underdetermined APIs, corresponding to the three choice points. Two of these are in the translation of the `JSONObject`

```
// in jpf-core/src/classes/java/lang/Class.java
... class Class ... {
 ...
 public native Field[] getDeclaredFields() throw...;
}
// in jpf-core/src/peers.../JPF_java_lang_Class.java
... class JPF_java_lang_Class extends NativePeer {
 ...
 @MJI
 public int
  getDeclaredFields_____3Ljava_lang_reflect_Field_2
   (MJIEnv env, int objRef) {
  ...
  for (i=0; i<nStatic; i++) {
   FieldInfo fi = ci.getStaticField(i);
   ...
  }
  for (i=0; i<nInstance; i++) {
   FieldInfo fi = ci.getDeclaredInstanceField(i);
   ...
  }
 }
}
```

Figure 3: Original `Class#getDeclaredFields` in JPF

to `String` in the method `toJSON` (called twice from the test), and one is in the body of the method `getStringRepresentation` (not shown here). Each of these choice points is over a collection with two elements (corresponding to the fields `author` and `title`), hence it has two possible orders.

Even this simple graph illustrates some interesting properties. For example, the non-deterministic choice point in state 0 is rather local, and both of its orders lead to the same state 1. The reason is that the first call to `toJSON` can produce two different string objects, but both of them produce the same `Book` object. Effectively, this choice point does not matter for the failure. What does matter is the relationship between the second and third choice points: if they choose the same order, the test passes, but if they choose different orders, the test fails. The probability that a randomly selected execution finds this failure is exactly 50%. Moreover, a simple strategy that always switches between choosing the natural order (the first outgoing edge, marked 0) and its opposite (the last outgoing edge, in this case, marked 1) would definitely find the failure in this example, but this is not always the case. We discuss in Section 4 the results from more examples.

3. TECHNIQUE AND IMPLEMENTATION

The overall NonDex technique is rather simple: we first manually find methods in the standard Java library with underdetermined specifications, then manually build models of these methods, and finally use an appropriate execution environment to explore various behaviors of these models. We next describe how we implemented NonDex models in JPF. One implementation of the `HashMap` iterator was already presented before [14]. Thus, we illustrate here the implementation of another method, and also mention one change we made in the former implementation of the `HashMap` iterator. The key goal of our implementation of NonDex in JPF is to enable *systematic exploration* of all possible behaviors of methods with underdetermined specifications.

To illustrate our encoding of models in JPF, consider the `getDeclaredFields` method from the class `java.lang.Class`. This method returns an array of the type `Field[]` which represents all the fields declared by the class (but excludes inherited fields). The Javadoc for this method states: *"The elements in the returned array are not sorted and are not in any particular order."* [3].

```
// modified Class.java
... class Class ... {
 ...
 public Field[] getDeclaredFields() throw... {
  return NonDex.shuffle(getDeclaredFields0());
 }
 public native Field[] getDeclaredFields0() ...;
}
// modified JPF_java_lang_Class.java
... class JPF_java_lang_Class extends NativePeer {
 ...
 @MJI
 public int
  getDeclaredFields0_____3Ljava_lang_reflect_Field_2
   (MJIEnv env, int objRef) {
  /* body the same as was in getDeclaredFields */
 }
}
```

Figure 4: Modified `Class#getDeclaredFields`

```
import gov.nasa.jpf.vm.Verify;
class NonDex {
 public static <T> T[] shuffle(T[] objs) {
  return shuffle(Arrays.asList(objs)).toArray(objs);
 }
 public static <T> List<T> shuffle(List<T> objs) {
  int permutation =
   Verify.getInt(0, factorial(objs.size()) - 1);
  return nthPermutation(permutation, objs);
 }
 public static <T> List<T> shuffleOld(List<T> objs){
  int k = objs.size();
  for (int i = 0; i < k - 1; i++) {
   Collections.swap(objs, i, Verify.getInt(i, k-1));
  }
  return objs;
 }
 ...
}
```

Figure 5: NonDex methods for shuffling

A typical implementation of this method is deterministic and returns the fields in some particular order. For example, in JPF, this method is implemented as a native peer with the relevant parts shown in Fig. 3. The `Class` implementation declares only that the method `getDeclaredFields` is native, and the actual implementation in `JPF_java_lang_Class.java` returns the array that has static fields before instance fields. Interestingly, the same `JPF_java_lang_Class.java` uses a different order in the method `getFields` which returns an array which represents all the public fields in the class and includes inherited fields—that method returns instance fields before static fields and has a comment *"the spec says there is no guaranteed order so we keep it simple"* [4].

To support NonDex, we modify `getDeclaredFields` such that JPF can explore all possible orders of the fields. We modified the implementation directly at the JPF level as shown in Fig. 4: (1) renamed the original `getDeclaredFields` peer to `getDeclaredFields0` and kept its body and (2) added the method `getDeclaredFields` to first obtain the original array of fields and then shuffle it using our NonDex method `shuffle` (described in the next paragraph). Note that we effectively modified the behavior of an existing native method to add shuffling, which is easy to do in JPF because the native methods are themselves implemented in Java.

We next describe how we implemented the `NonDex#shuffle` methods. Fig. 5 shows the key parts of our implementation. The `shuffle` method for arrays is the one invoked from `get-`

`DeclaredFields`, but many other methods require shuffling a list, so our key logic is in the `shuffle` method for lists. Its implementation is straightforward: given a list `objs`, it computes the total number of permutations of this list ($k!$, where k is the length of the list) and then selects one particular permutation to explore in each invocation, using the JPF library method `Verify#getInt`. (Note that both bounds in `getInt` are inclusive, hence subtracting one from the number of permutations.) The method `nthPermutation` computes the n-th permutation of a given list in the lexicographic order, using a traditional algorithm [13]. Note that several methods in the NonDex library modify their given arguments in place, but we ensure that they are called only when the arguments are copies that can be modified without affecting Java semantics. (While our goal is to explore all *possible* orders using NonDex, we do *not* want to generate some *impossible* order.) For example, the Javadoc for several `Class` methods explicitly states: *"The caller of this method is free to modify the returned array; it will have no effect on the arrays returned to other callers."*

Fig. 5 also shows an old shuffle method that we used in our first NonDex paper [14]. This method also enumerates all $k!$ permutations of the input `objs` list of length k, but it creates a different state-space graph that does not precisely capture the non-determinism inherent in these permutations. This method uses the Knuth shuffle [10] for random permutations but applies it to systematically explore all possible permutations. For each position i, it chooses some position between i and $k-1$ to swap with i.

To illustrate the difference between the methods `shuffle` and `shuffleOld`, consider a list with 4 elements. The current shuffle creates a single choice point with $4! = 24$ outgoing edges, i.e., the state-space graph has 25 nodes (1 choice point and 24 successor states). In contrast, the old shuffle would create one choice point with 4 outgoing edges of which each leads to a choice point with 3 outgoing edges of which each leads to a choice point with 2 outgoing edges, creating a factorial tree. This also gives 24 choices, but the state-space graph now has 40 edges and 41 nodes, i.e., 16 more edges and 16 more nodes than our current non-deterministic choice tree. These additional edges and nodes do not capture the non-determinism but are just the consequence of how permutations are computed. For this reason, all our experiments use the current `shuffle` implementation, not only for the new methods that we added but also for the `HashMap` iterator.

4. EVALUATION

We next present the results of our experiments on 46 student-written tests; we know from our previous work that (1) JPF can run these tests, at least for some executions, and (2) the tests contain wrong assumptions on APIs [14]. In the past, we ran these tests in JPF with only one underdetermined method and to find only one error state, thus we stopped the exploration on the first failure. In the current evaluation, our key goal is to analyze the state-space graphs, thus we run JPF with `search.multiple_errors=true`, and we also run with all 11 models of methods with underdetermined specifications (`HashMap` iterator and 10 methods similar to `getDeclaredFields`).

Table 1 shows the statistics about the state-space graphs. We obtained the full graphs for 46 failing tests. We previously had five additional tests [14]. During the exploration of two tests, JPF ran out of memory (the default 1GB) after finding 450,463 and 1,321,584 errors, respectively. Two tests were affected by a real bug in JPF, namely the JPF native peers in `JPF_java_lang_StringBuilder.java` and `JPF_java_lang_StringBuffer.java` do not work with the latest Java versions. The fifth test was mistakenly reported as failing in the past, because the code under test throws some exceptions that are caught, printed, and "swallowed"; the code does have some bugs but not of the kind that NonDex should find.

Table 1: Statistics of tests exploration

ID	#Nodes	#Edges	#Fail	P_f [%]	#Merges	#Crit
T1	7	7	2	50.00	0	2
T2	7	7	2	50.00	0	2
T3	208	283	64	75.00	29	32
T4	16	19	4	50.00	1	4
T5	7	7	2	50.00	0	2
T6	23	26	8	62.50	1	4
T7	5	4	1	50.00	0	1
T8	941099	950699	875520	98.96	386	9216
T9	53	71	36	72.22	4	6
T10	8	9	2	50.00	1	2
T11	8	9	2	50.00	1	2
T12	8130	8192	4032	98.44	0	64
T13	8	9	2	50.00	1	2
T14	35	42	12	75.00	5	4
T15	140	164	56	87.50	18	8
T16	150279	169994	65280	99.61	1797	256
T17	1124	1348	448	87.50	158	64
T18	10468	13252	3840	93.75	864	256
T19	8	8	2	50.00	0	2
T20	155	194	56	87.50	17	8
T21	224	332	56	87.50	22	8
T22	4	3	1	50.00	0	1
T23	6	5	2	75.00	0	1
T24	8825	9711	3968	96.88	700	128
T25	4	3	1	50.00	0	1
T26	885	1175	296	99.22	47	16
T27	17221	18311	8064	98.44	964	128
T28	7	7	2	50.00	0	2
T29	8	8	2	50.00	0	2
T30	8	8	2	50.00	0	2
T31	2645	3365	960	93.75	222	64
T32	9	8	3	87.50	0	1
T33	11	10	4	87.50	0	1
T34	15	15	6	75.00	0	2
T35	8	9	2	50.00	1	2
T36	6438913	12747262	65280	99.61	2113793	256
T37	5	4	1	50.00	0	1
T38	32	53	3	75.00	10	1
T39	24	37	3	75.00	6	1
T40	6	6	1	50.00	1	1
T41	5	4	1	50.00	0	1
T42	11	12	2	50.00	1	2
T43	1056	1106	552	95.83	28	24
T44	9	9	2	50.00	0	2
T45	20	23	6	87.50	3	2
T46	160	331	56	88.89	41	8

State-Space Graph Size: We tabulate the graph size (number of nodes and edges) as a measure of the uses of underdetermined APIs. We find that many tests have rather simple graphs, similar to the example from Section 2. However, a few tests have large graphs, with the largest (T36) having 6,438,913 nodes and 12,747,262 edges. Note that all the code is single-threaded, so the choice points are due only to the methods with underdetermined specifications. The largest choice point that we allow to be exhaustively explored is for collections with six elements, i.e., 720 outgoing transitions. For larger collections, we explore only one order, as provided by the underlying implementation.

Failure Probability: We also show the number of failing nodes and the failure probability. The latter is computed under the assumption that each (local) choice for each choice point is equally likely, e.g., if a choice point has 6 outgoing edges, each has 1/6 probability to be chosen. The overall failure rate is computed over a reverse topological sort of the graph: each failing node has the failure probability of 1.0, each passing node has the failure probability of 0.0, and an inner node with n children has the failure probability $(p_1 + \ldots + p_n)/n$, where $p_1, \ldots, p_n$ are failure probabilities of the successor nodes. The failure probability of the start node in the graph gives the overall failure probability for the graph. We can see that it can be as high as 99.61%, and is at least 50% in all cases; it means that a random selection of choices has a good chance to find any of these bugs (which confirms why our results with NonDex on JVM are already quite good [14]).

Irrelevant Non-determinism: We further measure how much of the non-determinism becomes irrelevant as the execution leads to the same state irrespective of the choices made at some choice point. Specifically, we count the number of "merge" nodes that have in-degree greater than one. (These are only the internal nodes and do not include the final, pass or fail, nodes.) While some tests have no merge nodes, other have quite a few, even up to almost one third of all nodes (T36 and T38). These merge nodes post-dominate some choice points that can be safely ignored when debugging the cause of failures due to underdetermined specifications in these cases.

Critical States: Collecting the entire state space enables us to determine the number of *critical* states, i.e., states with choice points from which at least one choice leads to paths that end either only in failure(s) or only in pass(es), while other choices lead to paths with different outcomes. In other words, these are the points where the exploration diverges, and so these are the key points for the developer to focus on when debugging failures that NonDex detects. We find that the number of critical states is relatively small compared to all states, the highest ratio being 32/208 for T3. Many cases have just one or two critical states. When JPF can analyze some code, our NonDex tool in JPF can greatly complement our NonDex tool in JVM: we envision a system where the tool in JVM is run first (because it can check all Java code and runs much faster for one execution) for some random choices, and if it detects a failure, then JPF is used to explore the neighborhood around this failure to determine which choice points are critical.

Choice Prioritization: Random exploration has a good chance to find the failure (e.g., with 50% failure probability for each path, trying just 7 independent paths gives over $1-(1/2)^7 > 99\%$ probability to find the failure), but we evaluate whether some prioritization heuristics could increase that chance. One seemingly good heuristic could be to first explore for each choice point the order that is *opposite (O)* of the *natural (N)* order, e.g., if some collection naturally returns `foo, bar, baz`, we could first explore `baz, bar, foo`. The intuition is that most tests pass for the natural order, and the opposite may create a completely unexpected situation. However, this heuristic finds failures in only 9 out of 46 tests. The reason is that many cases require *two* choices to be related for the failure (e.g., our running example requires two choices to differ). Additional heuristics are then to explore orders that alternate O and N, i.e., $ONONON$... or $NONONO$.... All three heuristics can find failures in 37 out of 46 cases, which is greater than 9 but still not perfect. In the future, we hope to identify heuristics that are even more likely to produce failures in most if not all cases.

5. CONCLUSIONS

In this paper we leveraged JPF to build on our previous work on NonDex, which introduced an approach for detecting erroneous assumptions client code makes about the libraries that have underdetermined specifications. We created models for 11 methods from the Java standard library (i.e., all methods that JPF supports from the current NonDex methods), and employed JPF to systematically explore state spaces of 46 tests from student homework submissions running with these models. Our experiments show several interesting results, which further our understanding of the complexity of checking code that invokes underdetermined specifications. We also provide hints for how future work can more efficiently detect tests that are brittle with respect to library changes that conform to specifications.

Acknowledgments

We thank Owolabi Legunsen and August Shi for engaging discussions about NonDex. This work was partially supported by NSF Grant Nos. CCF-1319688, CCF-1409423, CCF-1421503, and CCF-1438982.

6. REFERENCES

[1] Ben Lambeth's GitHub. https://github.com/azy2.
[2] Checkstyle Pull Request. https://github.com/checkstyle/checkstyle/pull/3393.
[3] Class#getDeclaredFields Javadoc. https://docs.oracle.com/javase/8/docs/api/java/lang/Class.html#getDeclaredFields--.
[4] Class#getFields in Java PathFinder. http://babelfish.arc.nasa.gov/hg/jpf/jpf-core/file/0069194b1048/src/peers/gov/nasa/jpf/vm/JPF_java_lang_Class.java#l580.
[5] M. B. Dwyer, S. Person, and S. Elbaum. Controlling factors in evaluating path-sensitive error detection techniques. In *FSE*, pages 92–104, 2006.
[6] A. Gyori, B. Lambeth, A. Shi, O. Legunsen, and D. Marinov. NonDex: A tool for detecting and debugging wrong assumptions on Java API specifications. In *FSE Demo 2016 to appear.*
[7] HashSet Javadoc. https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html.
[8] C. Huo and J. Clause. Improving oracle quality by detecting brittle assertions and unused inputs in tests. In *FSE*, pages 621–631, 2014.
[9] JUnit 4.11 - What's new? Test execution order. http://randomallsorts.blogspot.com/2012/12/junit-411-whats-new-test-execution-order.html.
[10] D. Knuth. Seminumerical algorithms, the art of computer programming, vol. 2 1997.
[11] NonDex Source Code. https://github.com/TestingResearchIllinois/NonDex.
[12] R. Pelánek. Properties of state spaces and their applications. *STTT*, 10(5):443–454, 2008.
[13] R. Sedgewick. Permutation generation methods. *ACM CSUR*, 9(2):137–164, 1977.
[14] A. Shi, A. Gyori, O. Legunsen, and D. Marinov. Detecting assumptions on deterministic implementations of non-deterministic specifications. In *ICST*, pages 80–90, 2016.