Verifying Determinism in Sequential Programs

Rashmi Mudduluru
Paul G. Allen School of Computer Science and Engineering
University of Washington
Seattle, USA
rashmi4@cs.washington.edu

***Abstract*—When a program is nondeterministic, it is difficult to test and debug. Nondeterminism occurs even in sequential programs: for example, as a result of iterating over the elements of a hash table. This seemingly innocuous and frequently used operation can result in diverging test results.**

We have created a type system that can express determinism specifications in a program. The key ideas in the type system are type qualifiers for nondeterminism, order-nondeterminism, and determinism. While state-of-the-art nondeterminism detection tools unsoundly rely on observing runtime output, our approach verifies determinism at compile time, thereby providing stronger soundness guarantees.

We implemented our type system for Java. Our type checker, the Determinism Checker, warns if a program is nondeterministic or verifies that the program is deterministic. In a case study of a 24,000-line software project, it found previously-unknown nondeterminism errors in a program that had been heavily vetted by its developers, who were greatly concerned about nondeterminism errors.

***Index Terms*—nondeterminism, type system, verification, specification**

I. Introduction

A nondeterministic program may produce different output on different runs when provided with the same input. This is a serious problem for software developers and users.

- Nondeterminism makes a program difficult to **test**, because test oracles must account for all possible behaviors while still enforcing correct behaviors [1]–[4].
- Nondeterminism makes it difficult to **compare** two runs of a program on different data, or to compare a run of a slightly modified program to an original program. This hinders debugging and maintenance, and prevents use of techniques such as Delta Debugging [5], [6].

Two well-known sources of nondeterminism are concurrency and coin-flipping (calls to a `random` API). It may be surprising that nondeterminism is common even in sequential programs that do not flip coins. For example, a program that iterates over a hash table may produce different output on different runs. So may any program that uses default formatting, such as Java's `Object.toString()`, which includes a memory address. Other nondeterministic APIs include date-and-time functions and accessing system properties such as the file system or environment variables.

The high-level goal of our work is to provide programmers with a tool for specifying deterministic properties in a program and verifying them statically.

We have created an analysis that detects nondeterminism or verifies its absence in sequential programs. Our analysis permits a programmer to specify which parts of their program are intentionally nondeterministic, and it verifies that the remainder is deterministic. Our analysis works at compile time, giving a guarantee over every possible execution of the program, unlike unsound dynamic tools that attempt to discover when a program has exhibited nondeterministic behavior on a specific run. Our analysis handles collections that will contain the same values, but possibly in a different order, on different runs. Our analysis permits calls to nondeterministic APIs, and only issues a warning if they are used in ways that may lead to nondeterministic output observed by a user. Like any sound analysis, it can issue false positive warnings.

II. Background and Related Work

Other researchers have also recognized the importance of the problem of nondeterminism. Previous work in program analysis for nondeterminism has focused on unsound dynamic approaches that identify flaky test cases. NonDex [2] uses a modified JVM that returns different results on different executions, for a few key JDK methods with loose specifications. Running a test suite multiple times may reveal unwarranted dependence on those methods. DeFlaker [3] looks at a range of commit versions of a code, and marks a test as flaky if it doesn't execute any modified code but still fails in the newer version. These techniques have been able to identify issues in real-world programs, some of which have been fixed by the developers. Identifying and resolving nondeterminism earlier in the software development lifecycle is beneficial to developers, because they can avoid bugs associated with flaky tests—reducing costs [7].

III. Approach and Uniqueness

The core of the determinism type system is the following type qualifiers:

- `NonDet` indicates that the expression might have different values in two different executions.
- `OrderNonDet` indicates that the expression is a collection or a map that contains the same elements in every execution, but possibly in a different order.
- `Det` indicates that the expression evaluates to equal values in all executions; for a collection, iteration also yields the values in the same order.

In `TypeVariable.java`:

```
160:   public List<TypeVariable> getTypeParameters() {
161:-    Set<TypeVariable> parameters = new HashSet<>(super.getTypeParameters());
161:+    Set<TypeVariable> parameters = new LinkedHashSet<>(super.getTypeParameters());
162:     parameters.add(this);
163:     return new ArrayList<>(parameters);
164:   }
```

Fig. 1: Fixes made by the Randoop developers in response to our bug report about improper use of a HashSet. Lines starting with "-" were removed and those starting with "+" were added. Our tool, the Determinism Checker, confirmed that 25 other uses of `new HashSet` were acceptable, as were 15 uses of `new HashMap`.

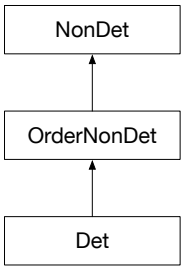

Fig. 2: Determinism type qualifier hierarchy

Figure 2 shows the subtyping relationship among the qualifiers.

Programmers can write these type qualifiers to specify their program's behavior. `OrderNonDet` may only be written on collections and maps. A map is a dictionary or an associative array, such as a hash table. Our type system largely treats a map as a collection of key–value pairs. Both collections and maps may be `Det`, `OrderNonDet`, or `NonDet`. The basetypes of their elements can be specified independently of the collection basetypes. However, an element type qualifier must be a subtype of the collection type qualifier.

1) Behavior of order-nondeterministic collections: A collection of type `OrderNonDet` has special properties, including the following.

1) The individual elements retrieved from it have type `NonDet`. This affects access, iteration, searching, etc.
2) Size-related operations return a deterministic result. This also affects queries of whether an iterator has more elements.
3) If the collection is sorted, or its elements are placed in a collection that does sorting, the result is deterministic.

IV. Evaluation

To evaluate the usability of the Determinism Checker, we applied it to the Randoop test generator [8]. Randoop is intended to be deterministic, when invoked on a deterministic program [9].[1] However, Randoop was not deterministic. This caused the developers problems in reproducing bugs reported by users, reproducing test failures during development, and understanding the effect of changes to Randoop by comparing executions of two similar variants of Randoop.

We first wrote specifications for libraries Randoop uses, such as the JDK, JUnit, and plume-util. Then, we wrote type qualifiers in the Randoop source code to express its determinism specification. Finally, we ran the Determinism Checker. Each warning indicated a mismatch between the specification and the implementation. We addressed each warning by changing our specification, reporting a bug in Randoop, or suppressing a false positive warning.

The Determinism Checker found 5 previously-unknown nondeterminism bugs in Randoop. The Randoop developers accepted our bug reports and committed fixes to the repository. Examples of severe bugs follow, according to the Randoop developers' categorization:

- **HashSet bug**: Suppose that in the code under test, a type variable's lower or upper bound has a type parameter that the type variable itself does not have. Then Randoop is nondeterministic. This situation does occur, even in Randoop's test suite. The developers fixed this by changing a `HashSet` to `LinkedHashSet` (commit c975a9f7, shown in fig. 1). The Determinism Checker confirmed that 25 other uses of `new HashSet` were acceptable, as were 15 uses of `new HashMap`.
- **Classpath bug** Randoop used the CLASSPATH environment variable in preference to the classpath passed on the command line. This can cause incorrect behavior, both in Randoop's test suite and in the field, if a user sets the environment variable. The developers fixed the problem by changing Randoop to not read the environment variable (commit 330e3c56). The Determinism Checker verified that all other uses of system and Java properties did not lead to nondeterministic behavior.

V. Research Contributions

This paper makes the following contributions:

1) We designed a type system for expressing determinism properties.
2) We implemented the analysis, as a pluggable type system for Java, in a tool called the Determinism Checker.
3) In a case study, we ran our analysis on a 24 KLOC project that developers had already spent weeks of testing and inspection effort to make deterministic. The Determinism Checker discovered 5 instances of nondeterminism that the developers had overlooked.

[1] Users of Randoop can pass in a different seed in order to obtain a different deterministic output. Randoop has command-line options that enable concurrency and timeouts, both of which can lead to nondeterministic behavior.

References

[1] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *FSE 2014: Proceedings of the ACM SIGSOFT 22nd Symposium on the Foundations of Software Engineering*, Hong Kong, November 2014, pp. 643–653.

[2] A. Shi, A. Gyori, O. Legunsen, and D. Marinov, "Detecting assumptions on deterministic implementations of non-deterministic specifications," in *ICST 2016: 9th International Conference on Software Testing, Verification and Validation (ICST)*, Chicago, IL, USA, April 2016, pp. 80–90.

[3] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "DeFlaker: Automatically detecting flaky tests," in *ICSE 2018, Proceedings of the 40th International Conference on Software Engineering*, Gothenburg, Sweden, May 2018, pp. 433–444.

[4] P. Sudarshan, https://www.thoughtworks.com/insights/blog/no-more-flaky-tests-go-team, 2012.

[5] A. Zeller, "Yesterday, my program worked. Today, it does not. Why?" in *ESEC/FSE '99: Proceedings of the 7th European Software Engineering Conference and the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Toulouse, France, September 1999, pp. 253–267.

[6] K. Yu, M. Lin, J. Chen, and X. Zhang, "Towards automated debugging in software evolution: Evaluating delta debugging on real regression bugs from the developers' perspectives," *J. Syst. Softw.*, vol. 85, no. 10, pp. 2305–2317, October 2012.

[7] K. Briski, P. Chitale, V. Hamilton, A. Pratt, B. Starr, J. Veroulis, and B. Villard, "Minimizing code defects to improve software quality and lower development costs," *Development Solutions. IBM. Crawford, B., Soto, R., de la Barra, CL*, 2008.

[8] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *ICSE 2007, Proceedings of the 29th International Conference on Software Engineering*, Minneapolis, MN, USA, May 2007, pp. 75–84.

[9] https://randoop.github.io/randoop/manual/index.html, February 2019, version 4.1.1.