Verifying Determinism in Sequential Programs

Rashmi Mudduluru
University of Washington
rashmi4@cs.washington.edu

Jason Waataja
University of Washington
jwaataja@cs.washinton.edu

Suzanne Millstein
University of Washington
smillst@cs.washinton.edu

Michael D. Ernst
University of Washington
mernst@cs.washinton.edu

***Abstract*—When a program is nondeterministic, it is difficult to test and debug. Nondeterminism occurs even in sequential programs: e.g., by iterating over the elements of a hash table.**

We have created a type system that expresses determinism specifications in a program. The key ideas in the type system are type qualifiers for nondeterminism, order-nondeterminism, and determinism; type well-formedness rules to restrict collection types; and enhancements to polymorphism that improve precision when analyzing collection operations. While state-of-the-art nondeterminism detection tools rely on observing output from specific runs, our approach soundly verifies determinism at compile time.

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 case studies of 90097 lines of code, the Determinism Checker found 87 previously-unknown nondeterminism errors, even in programs that had been heavily vetted by developers who were greatly concerned about nondeterminism errors. In experiments, the Determinism Checker found all of the non-concurrency-related nondeterminism that was found by state-of-the-art dynamic approaches for detecting flaky tests.

***Index Terms*—nondeterminism, type system, verification, specification, hash table, flaky tests**

I. Introduction

A nondeterministic program may produce different output on different runs when provided with the same input. Nondeterminism 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. Test oracles that are too strict lead to flaky tests, which sometimes pass and sometimes fail [1], [2], [3], [4], [5]. Flaky tests must be re-run, or developers ignore them; in either case, their utility to detect defects is limited.
- 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 [6], [7].
- Nondeterminism reduces users' and developers' **trust** in a program's output [8], [9].

These problems motivate the field of deterministic replay [10].

Nondeterminism is common even where it is not expected. For example, a program that relies on the iteration order of a hash table, or on any other property of hash codes, 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 `random`, date-and-time functions, and accessing system properties such as the file system or environment variables. Another source of nondeterminism is concurrency, but our work focuses on sequential programs.

The high-level goal of our work is to provide programmers with a tool for **specifying** determinism properties in a program and **verifying** them statically. 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 [2], [3]. 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 and reduces costs [11].

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. The programmer specifies whether a particular part of the program is allowed to be nondeterministic or not. The tool reports when the program deviates from that behavior. Any deviation is a bug either in the (possibly defaulted) specification or in the program. The tool identifies nondeterminism where the specification does not permit it. Then, the programmer can fix the inconsistency.

Our approach is based on type systems that analyzes determinism at compile time. It does not rely on examining output from specific runs. Type systems are as expressive as any other static analysis [12]. A type-based approach divides the responsibility between the user and the tool. Ours is a specification-and-verification approach. The user writes a specification of the intended behavior of the program, and the tool reports whether the program violates the specification.

If our analysis issues no warnings, then the program produces the same output when executed twice on the same inputs, modulo the limits of the analysis (see section VIII). 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 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.

Our main contributions are a type system for expressing determinism properties (section II) and an implementation for Java, in a tool called the Determinism Checker (section III). While the approach is applicable to any statically typed object-oriented programming language, our implementation works only for Java. To validate our work, we performed case studies and experiments. In the case studies, we ran our analysis on 90097 LoC (13 projects), including ones whose developers had already spent weeks of testing and inspection effort to make deterministic (section IV). The Determinism Checker discovered 87 instances of nondeterminism that the developers had overlooked. The developers fixed most of these issues when we reported them. Figure 2 shows an example. In the experiments, we compared our tool against state-of-the-art flaky test detectors, on their benchmarks (sections V and VI). The Determinism Checker found all the non-concurrency nondeterminism found by the other tools.

II. A TYPE SYSTEM FOR DETERMINISM

This section presents the key aspects of our type system. Section II-A reviews the notion of type qualifiers and how they help type-checking. Section II-B introduces determinism type qualifiers informally. Section II-C formalizes the type system, gives examples, and proves soundness. Section II-D discusses how polymorphism enables more precise specification of (non)deterministic behavior.

A. Preliminaries and notation

A programming language provides *basetypes*, such as `Int`. A *type qualifier* [13] on a basetype adds additional constraints; that is, it reduces the set of values that the type represents. An example type qualifier is `Positive`, and an example type is `Positive Int`.

A type qualifier constrains the set of possible run-time values, that is, `Positive Int` $<:$ `Int`. As a result, a qualifier type system does not allow any values that the original type system does not, in the same program without qualifiers. However, the qualifier type system may reject more programs, and thus affords stronger guarantees.

B. Determinism types

The core of the determinism type system is the type qualifiers `NonDet` $:>$ `OrderNonDet` $:>$ `Det`.

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

`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 (see fig. 1).

Our approach is applicable to any object-oriented programming language. For concreteness, our formalism and implementation build upon Java.

C. Formalizing our type system

We formalized a core language as an extension to Featherweight Generic Java (FGJ) (Figure 4 in [14]). Due to space constraints, the full formalism and detailed proofs appear in [15]. Our language adds the following language features to FGJ: 1) determinism type qualifiers, 2) aliasing, 3) mutation, 4) collection classes from the JDK, and 5) arrays.

Our core language adds aliasing and mutation to FGJ via expressions $e.f = e$ and $e = e.f$. Additionally, basetypes in our language include arrays and consequently array accesses $e = e[i]$ and array mutations $e[i] = e$. Collection classes are invariant with respect to the determinism qualifier. That is, a `NonDet Collection` is unrelated by subtyping to an `OrderNonDet Collection` which is unrelated to a `Det Collection` (for details, see section II-C5). Arrays are treated as covariant with respect to determinism type qualifiers. This is sound because we forbid mutating arrays of type `NonDet e[OrderNonDet i]` or `NonDet e[Det i]`. These core language features express the essential features of our type system.

1) Type well-formedness and Collection types: `OrderNonDet` may not be written on types other than `Collection`, `Iterator`, and `Map`. Figure 1 gives examples. The type qualifier on the type argument of a `Collection` must be a subtype of the type qualifier on the `Collection` type.

Our design uses types to distinguish between expressions that evaluate to the same values on each execution, and those that may not. To achieve this goal, determinism is a "deep" rather than a "shallow" property: if an expression of collection or array type is nondeterministic, then so are its elements, and if an expression of reference type is nondeterministic, then so are its fields.

2) Behavior of order-nondeterministic collections: A collection whose type qualifier is `OrderNonDet` has the following properties:

1) Elements retrieved from it (via access, iteration, searching, etc.) have type `NonDet`.
2) Size-related operations, and queries of whether an iterator has more elements, return a deterministic result.

To restate the first point, the typical type for a list access operation, such as `get`, is $\forall E.$ `List`$\langle E \rangle \times$ `Int` $\rightarrow E$, but this type is correct only when both arguments are `Det`. The precise type is more complex, because if *either* argument to `get` is `OrderNonDet` or `NonDet`, then the result is `NonDet`. Section II-D shows how to express this polymorphism.

3) Typing rules and field accesses: In our type system, whenever a field is accessed on the RHS of a (pseudo-) assignment, the type qualifier of that expression is the least upper bound (`lub`) of the type qualifier of the field type and

`NonDet List⟨NonDet Int⟩`	`NonDet List⟨OrderNonDet Set⟩`	`NonDet List⟨Det Int⟩`
~~`OrderNonDet List⟨NonDet Int⟩`~~	`OrderNonDet List⟨OrderNonDet Set⟩`	`OrderNonDet List⟨Det Int⟩`
~~`Det List⟨NonDet Int⟩`~~	~~`Det List⟨OrderNonDet Set⟩`~~	`Det List⟨Det Int⟩`

Fig. 1: A collection's type qualifier must be a supertype or equal to the element type qualifier. The struck-out types are invalid.

that of the field's class type. To illustrate the need for this rule, consider the example below:

```
class MyClass {
    Det Integer dField;
    NonDet Integer getFieldOfFirst(
    OrderNonDet List<Det MyClass> list) {
        NonDet MyClass element = list.get(0);
        return element.dField;
    }
}
```

The iteration order of the formal parameter `list`, of type `Order-NonDet List`, is arbitrary. Therefore, the type of `list.get(0)` has the type qualifier `NonDet`. In other words, `element` could have different values across executions. As a result, the expression `element.dField` is `NonDet` even though `dField` is declared as `Det`.

An assignment $x.f = y$ is valid iff the type qualifier on the type of x is a subtype of the type qualifier on the declaration type of f. The following example justifies this rule:

```
class MyClass {
    Det Integer dField;
    void bad(OrderNonDet List<Det MyClass> list) {
        NonDet MyClass element = list.get(0);
        element.dField = ...;       // This is invalid
    }
}
```

Since `element` could have different values across executions, it is `NonDet`. Suppose `list` had two elements `elem1` and `elem2`. In one execution, `list.get(0)` could return `elem1` and the statement `element.dField = ..;` would set a field of `elem1`. In another execution, `list.get(0)` could return `elem2` and the assignment statement would set a field of `elem2`. In other words, the method `bad` creates a `NonDet` alias to a `Det` instance which allows the instance to be mutated non-deterministically. To prevent this unsoundness, our type system forbids the assignment to `element.dField`.

4) Theorems and proofs: The following theorems imply that our type system is sound: it suffers no false negatives. If our type system issues no warnings, then no expression with deterministic type evaluates to a different value on different runs over the same inputs.

Theorem 1 (Type preservation)**.** *When an expression e reduces to another expression e', e''s type is a subtype of e's type. More formally, if $\Delta;\Gamma \vdash e : T$ and $e \rightarrow e'$ then $\Delta;\Gamma \vdash e' : T'$ for some T' such that $\Delta \vdash T' <: T$*

Proof. By induction on the derivation of $e \rightarrow e'$, similar to the proof of Theorem 3.4.1 in [14]. We show the cases for our new reduction and type validity rules.

- *Case 1:* $e_1 = e_2.f$. $e_2.f \rightarrow e'$ implies $e' : lub(\lambda, \kappa)\ V$ where $e_2 : \kappa\ U$, $f : \lambda\ V$. (Here, κ, λ are type qualifiers and U, V are basetypes.) Upon execution of the statement $e_1 = e_2.f$, e_1 reduces to $e_2.f$. So the type of e_1 is exactly the type of e' which satisfies $e' <: e_1$.
- *Case 2:* $e_1.f = e_2$. The assignment is either invalid or trivially preserves types ($e_1.f$ has the same type as that of e_2).
- *Case 3:* $e_1 = e_2[i]$. Similar to *Case 1*.
- *Case 4:* $e_1[i] = e_2$. Similar to *Case 2*.

The determinism type qualifiers do not change reduction rules — they only affect subtyping and type validity. Invariant collections only add constraints to subtyping, which does not affect the reduction $e \rightarrow e'$. □

Theorem 2 (Progress)**.** *If e is well-typed, then one of the following applies: 1. e contains a failed downcast, 2. e contains a failed assignment due to the violation of field assignment rule, 3. e contains a failed assignment due to the violation of array assignment rule, or 4. there is a valid reduction rule.*

Proof. This theorem is proved by a case analysis of all expression types. The only difference from the proof of Theorem 3.2 in [14] is due the type validity of field and array accesses on the LHS. In the case of field assignment $e_1.f = e_2$, either the expression $e_1.f$ is invalid or it reduces to the same type as that of e_2. A similar analysis handles array accesses. □

5) Collection aliasing, mutation, and invariance: This section discusses why collection classes are invariant with respect to determinism type qualifiers. Assume for the sake of contradiction that collections are not invariant w.r.t. type qualifiers. That would make the following code valid:

```
void test(Det List<Det String> dList,
                NonDet List<Det String> nList) {
    NonDet List<Det String> ndList = dList;
    ndList.addAll(nList);
}
```

The above code is unsafe because `nList` and `dList` are aliases to the same `List` object and `nList.addAll(nList)` mutates the `Det` reference `dList` non-deterministically. That is, it would violate case 2 of the proof of theorem 1. To avoid such unsoundness, our type system disallows collection instances to have two aliases that differ in their determinism types. It achieves this by declaring all collection classes to be invariant with respect to determinism type qualifiers.

D. Polymorphism

As described so far, our type system is sound, but it suffers from poor expressiveness. An implementation would issue many false positive warnings, because programmers could write only coarse specifications of methods. Adding polymorphism to our type system increases its expressiveness without compromising soundness [16], [17]. This section focuses on precise specifications (method signatures), rather than on the type-checking that ensures that the method body conforms to the specification.

Section II-D1 first describes basic polymorphism over type qualifiers and over basetypes. The subsequent sections describe polymorphic extensions.

1) Qualifier and basetype polymorphism: Our type system supports parametric polymorphism [18], [19]. A polymorphic abstraction (a class or method) is written and type-checked once. Informally, it acts as if it has multiple different types, and each use site is typechecked using the most specific applicable instantiated non-polymorphic type.

Our type system supports both basetype polymorphism and qualifier polymorphism.

- To achieve typical type polymorphism, use both basetype polymorphism and qualifier polymorphism. For example, the type of the identity function is $\forall T.\ T \rightarrow T$, which can be equivalently written as $\forall \kappa,\ U.\ \kappa\ U \rightarrow \kappa\ U$.
- Qualifier polymorphism is commonly needed. For example, the `length` method on strings has type `length` : $\forall\kappa.\ \kappa$ `String` $\rightarrow \kappa$ `Int` .

This paper adopts the convention that polymorphism is not instantiated in ways that would create invalid types. For example, the `length` polymorphic function would not be instantiated at $\kappa =$ `OrderNonDet`. (This makes no difference for the `length` function, because such an instantiation would never be the most specific applicable one.)

2) Polymorphism rules for collections: As described so far, polymorphism cannot express the collection behaviors of section II-C2. Consider this potential typing for the `size` method in class κ `Collection`$\langle\tau\ E\rangle$:

`size` : $\forall\kappa.\ \kappa$ `Collection`$\langle\tau\ E\rangle \rightarrow \kappa$ `Int`

It cannot be instantiated at $\kappa =$ `OrderNonDet` as

`size` : `OrderNonDet Collection`$\langle\tau\ E\rangle \rightarrow$ `OrderNonDet Int`

because such an instantiation would include the invalid return type `OrderNonDet Int`.

Our type system resolves this problem by introducing two operators over polymorphic type variables, $\uparrow$ and $\downarrow$. The $\uparrow$ operator converts `OrderNonDet` to `NonDet` and leaves the other qualifiers unchanged. The upward-pointing arrow is a mnemonic for replacing `OrderNonDet` by something higher in the type hierarchy. The $\downarrow$ operator is analogous, but it converts `OrderNonDet` to `Det`, which is lower in the type hierarchy.

The precise type for `size` is

`size` : $\forall\kappa.\ \kappa$ `Collection`$\langle\tau\ E\rangle \rightarrow \kappa\!\downarrow$ `Int`

This can be instantiated at all three type qualifiers without creating any invalid types:

`size` : `NonDet Collection`$\langle\tau\ E\rangle \rightarrow$ `NonDet Int`
`size` : `OrderNonDet Collection`$\langle\tau\ E\rangle \rightarrow$ `Det Int`
`size` : `Det Collection`$\langle\tau\ E\rangle \rightarrow$ `Det Int`

These instantiations implement the semantics of section II-C2.

An example use of $\uparrow$ is in a method that returns the first element of a list. Its type is `first` : $\forall\kappa.\ \kappa$ `List`$\langle\tau\ E\rangle \rightarrow \kappa\!\uparrow E$ which can be instantiated as

`first` : `NonDet List`$\langle\tau\ E\rangle \rightarrow$ `NonDet` E
`first` : `OrderNonDet List`$\langle\tau\ E\rangle \rightarrow$ `NonDet` E
`first` : `Det List`$\langle\tau\ E\rangle \rightarrow$ `Det` E

	`ns : NonDet String`	`ds : Det String`
`l : NonDet List`$\langle$`NonDet String`$\rangle$	`l.add(ns)`	`l.add(ds)`
`l : NonDet List`$\langle$`Det String`$\rangle$	`l.add(ns)`	`l.add(ds)`
`l : OrderNonDet List`$\langle$`Det String`$\rangle$	~~`l.add(ns)`~~	`l.add(ds)`
`l : Det List`$\langle$`Det String`$\rangle$	~~`l.add(ns)`~~	`l.add(ds)`

TABLE I: `add` invocations for a well-formed list.

In addition to the $\uparrow$ and $\downarrow$ operators, our type system also defines the *shuffle*() operator which converts `Det` to `OrderNonDet` and leaves the other qualifiers unchanged. This enables precise specification of certain collection operations. For instance, the type of the `HashSet` constructor is:

`HashSet()` : $\forall\kappa.\ \kappa\ E \rightarrow$ *shuffle*(κ) `HashSet`$\langle\kappa\ E\rangle$

That is, invoking the `HashSet` constructor with an argument of type `Det E` will construct an `OrderNonDet HashSet`.

3) Differentiating binding and use: Precisely specifying mutation operations on collections requires another extension to polymorphism. We discuss our approach to annotating mutation methods in three parts: (1) determinism types on non-receiver parameters, (2) excluding `OrderNonDet`, (3) aliasing.

a) Determinism types on non-receiver parameters: Consider a mutator method `add`. Its type without determinism qualifiers is `add` : `List`$\langle$`String`$\rangle \times$ `String` $\rightarrow ()$. (For simplicity, this discussion treats the return type of `add` as `void` even though in the JDK it is `String`.)

Table I shows all possible invocations of `add` for a well-formed `List` type. The specification for `add` must reject the calls that are struck out, or else the body will not type-check (and unsafe client code would type-check). The specification should permit all the calls that are not struck out, or else some safe client code will not type-check. It achieves these goals via another variant of qualifier variables, $use(\kappa)$, which represents a *use* of κ that does not affect the *instantiation* of κ.

Ordinarily, a polymorphic function is instantiated at the least upper bound of the types of all the arguments that correspond to uses of the type parameter. For example, function

f : $\forall\kappa.\ \kappa$ `Int` $\times$ `Det Int` $\times\ \kappa$ `Int` $\times\ \kappa$ `Int`

is instantiated at the least upper bound of the types of its first, third, and fourth arguments at a given call. (If no such instantiation exists with valid types, or if any other argument does not conform to its corresponding formal parameter type, the call does not type-check.) By contrast, function

f : $\forall\kappa.\ \kappa$ `Int` $\times$ `Det Int` $\times\ use(\kappa)$ `Int` $\times\ \kappa$ `Int`

is instantiated at the least upper bound of the types of its first and fourth arguments, and the third argument must conform to that instantiation. That is, the type qualifier of the third argument must be a subtype of the least upper bound of the type qualifiers of the first and fourth arguments. Given this type system feature, the type of `List`'s `add` method can be precisely specified:

`add` : $\forall\kappa,\beta.\ \kappa$ `List`$\langle\beta\ E\rangle \times use(\kappa)\ E \rightarrow ()$

At a call site, if β is not a subtype of κ, the `List` type is invalid and the call does not type-check. As another example for the $use()$ operation, the precise type of `addAll` is:

addAll : $\forall\kappa,\beta.\ \kappa$ List$\langle\beta\ E\rangle \times use(\kappa)$ Collection$\langle...\rangle \rightarrow$ $\kappa\!\downarrow boolean$

b) Excluding `OrderNonDet`: The specification of the JDK must prohibit certain mutation operations on collections. For example, the annotations in the JDK must prohibit removing from `OrderNonDet` collections at deterministic indices. The following client code must not type-check:

```
void mustBeProhibited(OrderNonDet List<Det String> lst,
                      Det int index) {
    lst.remove(index);
}
```

Since the iteration order on `OrderNonDet` collections is not guaranteed, the element at `index` could differ across executions. As a result, `lst.remove(index)` could remove different elements on different executions, leaving `lst` with different contents on different executions, which violates the contract of the `OrderNonDet` type qualifier.

However, the specification of `remove` should permit removal from `Det` and `NonDet` collections. A precise type is

remove : $\forall\kappa \in \{\texttt{NonDet}, \texttt{Det}\}, \tau.$
$$\kappa\ \texttt{List}\langle\tau\ E\rangle \times use(\kappa)\ \texttt{int} \rightarrow ()$$

Section III-C gives the Java syntax of this qualifier polymorphism that excludes `OrderNonDet`.

c) Aliasing: It is possible to create `NonDet` aliases to `OrderNonDet` or `Det` collections. (For instance, by calling `next` or `get`.) Consider the example below:

```
void aliasTest(OrderNonDet Set<Det List<Det String>> set,
               NonDet int index, Det String str) {
    NonDet List<Det String> lst = set.iterator().next();
    lst.add(index, str);
}
```

Variable `lst` has type `NonDet List`$\langle$`Det String`$\rangle$ (as a result of set iteration) but an alias has type `Det List`$\langle$`Det String`$\rangle$ (as a member of `set`).

The call `list.add()` is unsafe and must not typecheck. It mutates `lst` non-deterministically thereby violating the determinism guarantees provided by the `Det` reference.

Our type system prevents the unsafe behavior by prohibiting any mutation of collections having types `NonDet Collection`$\langle$`OrderNonDet` E$\rangle$ or `NonDet Collection`$\langle$`Det` E$\rangle$. It achieves this via JDK annotations on collection classes that guarantee that, in the above example, `list.add()` does not typecheck.

A few JDK operations (like iteration or access) can create such unsafe aliases among otherwise invariant `Collection` types. These operations can return a `NonDet` alias to an `OrderNonDet` or a `Det` type. Our approach of preventing mutations on `NonDet Collection`$\langle$`OrderNonDet` E$\rangle$ and `NonDet Collection`$\langle$`Det` E$\rangle$ is sufficient to prevent unsafe behavior.

III. Implementation of our type system

We implemented our type system for Java, in a tool named the Determinism Checker. The implementation consists of 5047 lines of Java code built atop the Checker Framework, plus 3322 lines of tests, 1034 annotated library methods, a 3138-line manual, etc. (All line measurements are non-comment, non-blank lines.) The Determinism Checker works with Java version 8 and 11. It is publicly available at https://github.com/t-rasmud/checker-framework/tree/nondet-checker. Sections III-A to III-C discuss Java type qualifiers, qualifiers for collections, and polymorphic qualifiers, respectively. Section III-D describes how the Determinism Checker implements invariant types for collections. To reduce the annotation burden on the programmer, the Determinism Checker uses defaulting and type refinement as presented in sections III-E and III-F.

A. Determinism type qualifiers

A type qualifier is written in Java source code as a type annotation. A type annotation has a leading "`@`" and is written immediately before a Java basetype.

The Determinism Checker supports the type qualifiers `@NonDet`, `@OrderNonDet`, and `@Det`, plus others described below. The meaning of `@Det` is with respect to value equality, not reference equality.

For simplicity, section III uses the term "collection" and the type `Collection` to represent arrays and any type that implements the `Iterable` or `Iterator` interfaces; this includes all Java collections including `List`, `Set`, and user-defined classes.

B. Java collection types

A `Map` is deterministic if its `entrySet` is deterministic.

The most widely used `Map` implementations have the following properties:

- `HashMap` is implemented in terms of a hash table, which never guarantees deterministic iteration over its entries. A `@Det HashMap` does not exist.
- `LinkedHashMap`, like `List`, can have any of the `@NonDet`, `@OrderNonDet`, or `@Det` type qualifiers. Iterating over a `LinkedHashMap` returns its entries in the order of their insertion. An `@OrderNonDet LinkedHashMap` can be created by passing an `@OrderNonDet Map` to its constructor, as in `new LinkedHashMap(myOndMap)`.
- `TreeMap` can be `@Det` or `@NonDet`. An `@OrderNonDet TreeMap` does not exist because the entries are always sorted.

The Determinism Checker prohibits the creation of a `@Det HashMap` or an `@OrderNonDet TreeMap`.

C. Polymorphism

Java does not provide a syntax that can be used for qualifier polymorphism, so the Determinism Checker follows the Checker Framework convention [20] and uses a special type qualifier name, `@PolyDet`. (`@PolyDet` stands for "polymorphic determinism qualifier".) A qualifier-polymorphic method `m` with type $\forall\kappa.\ \kappa\ \texttt{int} \times \texttt{Det boolean} \rightarrow \kappa\ \texttt{String}$ is declared as `@PolyDet String m(@PolyDet int, @Det boolean)`. Each use of `@PolyDet` stands for a use of the qualifier variable κ, and there is no need to declare the qualifier variable κ. Qualifier polymorphism is common on methods that a programmer might think of as deterministic. For example, an addition method should be defined as

```
@PolyDet int plus(@PolyDet int a, @PolyDet int b) {...}
```

This can be used in more contexts than

```
@Det int plus(@Det int a, @Det int b) {...}
```

Just as a qualifier variable κ is written as `@PolyDet` in Java source code, $\kappa\uparrow$ is written as `@PolyDet("up")`, $\kappa\downarrow$ as `@PolyDet("down")`, and $\mathit{shuffle}(\kappa)$ as `@PolyDet("shuffle")`. An occurrence of a qualifier variable that does not affect the binding of that variable ($use(\kappa)$ in section II-D3) is written `@PolyDet("use")`. A qualifier variable that excludes `OrderNonDet` is written as `@PolyDet("noOrderNonDet")`. An occurrence of a qualifier variable that does not affect the binding of `@PolyDet("noOrderNonDet")` is written `@PolyDet("use,noOrderNonDet")`. All of this syntax is legal Java code that can be compiled with any Java 8 or later compiler.

D. Determinism invariant types

A class or interface annotated with `@HasQualifierParameter` is treated as invariant with respect to determinism type qualifiers. For example, the `Collection` class is annotated as

```
@HasQualifierParameter(NonDet.class)
public interface Collection<E> extends Iterable<E> {..}
```

Every subtype (e.g., `List`) of a type annotated with `@HasQualifierParameter` inherits this annotation and is therefore invariant w.r.t. determinism qualifiers. At a use site, suppose a `List` type is annotated as `@NonDet List<@Det String> lst`. Any polymorphic field (that is, a field whose type qualifier is `@PolyDet`) in `List` accessed via `lst` will resolve to `@NonDet`.

E. Defaulting

The Determinism Checker applies a default qualifier at each unqualified Java basetype (except uses of type parameters, which already stand for a type that was defaulted at the instantiation site where a type argument was supplied). This does not change the expressivity of the type system; it merely makes the system more convenient to use by reducing programmer effort and code clutter. Defaulted type qualifiers are not trusted: they are type-checked just as explicitly-written ones are. In other words, defaulting is a syntactic convenience that does not change the semantics or expressiveness of the type system. As a result, defaults never lead to false alarms. The tool might issue an alarm that indicates that the default specification is not consistent with the code. This is not a false alarm. Rather, it indicates that the programmer needs to write the specification for that part of the program.

Formal parameter and return types default to `@PolyDet`. The programmer can change the default for formal parameters and return types to `@Det`. The `@Det` default makes it easier to annotate a codebase and requires less use of the Determinism Checker's polymorphism features, but it makes the code usable by fewer clients; it is appropriate for programs but not for libraries. Fields of a class annotated with `@HasQualifierParameter` default to `@PolyDet`. Types are inferred for unannotated local variables; see section III-F. The default annotation for other unannotated types is `@Det`, because programmers generally expect their programs to behave the same when re-run on the same inputs.

F. Type refinement via dataflow analysis

Our type system is flow-sensitive [21], [22], [23]. That is, an expression may have a different type qualifier on every line of the program, based on assignments and side effects. Type preservation (theorem 1) is not violated, because the refined type is always consistent with (that is, a subtype of) the declared type. Type refinement does not apply to types that are invariant (annotated with `@HasQualifierParameter`), because they have no subtypes.

Flow-sensitive type refinement applies to arbitrary expressions, including fields and pure method calls. A type refinement is lost whenever a side effect might affect the value.

This flow-sensitive type refinement achieves local variable inference, freeing programmers from writing many local variable types.

Although the Determinism Checker performs local type inference within method bodies, it does not perform whole-program type inference. This makes separate compilation possible.

Project	LoC	Bugs found	#Warning suppressions	#Annotations
Randoop	25176	15	84	3385
Checkstyle	36182	13	96	511
CF dataflow analysis	13519	43	92	0
bcel-util	2472	2	96	170
bibtex-clean	53	0	0	1
html-pretty-print	51	0	0	0
icalavailable	388	1	0	6
lookup	283	0	0	2
multi-version-control	1220	3	100	16
options	1818	5	99	22
plume-util	7688	2	68	1037
reflection-util	802	2	100	28
require-javadoc	445	1	0	3

TABLE II: Results of the Determinism Checker case studies

IV. CASE STUDIES

To evaluate the usability of the Determinism Checker, we applied it to several projects: Randoop (a test generator), Checkstyle (a linter), the Checker Framework's dataflow analysis, and the plume-lib utilities. All the materials are publicly available at [24] for reproducibility.

We chose Randoop [25] because it is frequently used in software engineering experiments, and its developers have struggled with nondeterminism [26], [27].

We chose Checkstyle [28] because it was the only buildable project with a confirmed non-concurrency determinism bug in DeFlaker's experiments.

We chose the dataflow analysis [29] because, while building the Determinism Checker on top to the Checker Framework, we discovered a determinism bug in this component. We began our case study after that bug was fixed.

We chose the plume-lib utilities [30] because they are used by Randoop (and thus were subject to the same extensive vetting process) and have the same maintainers, who were responsive to us. Some of the projects are libraries, and some are programs.

Table II shows the results of the Determinism Checker case studies. For reasons of space, this paper discusses the Randoop case study in greatest detail and the others more briefly.

A. Case Study 1: Randoop

Randoop is intended to be deterministic, when invoked on a deterministic program [31].[1] However, Randoop was not deterministic. This caused the developers problems in reproducing bugs reported by users, in reproducing test failures during development, and in understanding the effect of changes to Randoop by comparing executions of two versions of Randoop.

The developers took extensive action to detect and mitigate nondeterminism. They used Docker images to run tests, to avoid system dependencies such as a different JDK having a different number of classes or methods. They wrote tests with relaxed oracles (assertions) that permit multiple possible answers — for example, in code coverage of generated tests. They used linters such as Error Prone to warn if `toString` is used on objects, such as arrays, that do not override `Object.toString` and therefore print a hash code which may vary from run to run. They used a library that makes hash codes deterministic, by giving each object of a type a unique ID that counts up from 1 rather than using a memory address as `Object.hashCode` does. They wrote specialized tools to preprocess output and logs to make them easier to compare, such as by removing or canonicalizing hash codes, dates, and other nondeterministic output. These efforts were insufficient.

In July 2017, the Randoop developers spent two weeks of full-time work to eliminate unintentional nondeterministic behavior in Randoop (commits e15f9155–32f72234). Their methodology was to repeatedly run Randoop with verbose logging enabled, look for differences in logging output, find the root cause of nondeterminism, and eliminate it (personal communication, 2019). Some of the nondeterminism was in libraries, such as the JDK. The most common causes were `toString` routines and iteration order of sets and maps. The most common fixes were to change the implementations of `toString` and to use `LinkedHashSet` and `LinkedHashMap` or to sort collections before iterating over them. The developers did not make every `Set` and `Map` a `LinkedHashSet` or `LinkedHashMap`, because that was unnecessary and would have increased memory and CPU costs. They chose not to make every order-nondeterministic `List` a `Set`, for similar reasons: deduplication was not always desired, and even where it was acceptable, it would have increased costs.

That coding sprint did not find all the problems. The developers debugged and fixed 5 additional determinism defects over the next 12 months, using a similar methodology (commits c15ccbf2, 44bdeebd, 5ff5b4c4, 22eda87f, and b473fd14). We analyzed Randoop after all these fixes.

1) Methodology: We wrote type qualifiers in the Randoop source code to express its determinism specification, then 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.

[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.

In `TypeVariable.java`:
```
  public List<TypeVariable> getTypeParameters() {
    Set<TypeVariable> parameters =
-     new HashSet<>(super.getTypeParameters());
+     new LinkedHashSet<>(super.getTypeParameters());
    parameters.add(this);
    return new ArrayList<>(parameters);
  }
```

Fig. 2: The fix 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 24 other uses of `new HashSet` were acceptable, as were 18 uses of `new HashMap`.

We annotated the core of Randoop (the `src/main/java` directory), which contains 25K non-comment, non-blank lines of code. We did not annotate Randoop's test suite.

We annotated one package at a time, starting with the packages that are most depended upon. Within a package, we followed a similar strategy, annotating supertypes first. We reverse-engineered each specification, largely from the methods it calls. (If the determinism of classes and methods had been documented, then our annotation effort would have been easy, just converting English into type qualifiers.) The effort would have been much easier for someone familiar with Randoop, and yet easier if done while code is being written and is malleable.

Running `./gradlew clean compileJava` takes 18 seconds to compile all files of Randoop. While also running the Determinism Checker as a compiler plugin, the command takes 32 seconds. These numbers are the median of 5 trials on an 8-core Intel i7-3770 CPU running at 3.40GHz with 32GB of memory.

2) Results: The Determinism Checker found 15 previously-unknown nondeterminism bugs in Randoop. The Randoop developers accepted our bug reports and committed fixes to the repository. A summary of these bugs follows, according to the Randoop developers' categorization:

Severe issues: Nondeterminism in Randoop output.

- **HashSet bug**: The "code under test" is the code Randoop is testing (contrast to Randoop's source code, which the Determinism Checker is verifying). Suppose that, in the code under test, a type variable's bound has a type parameter that the type variable itself does not have. (This situation does occur, even in Randoop's test suite.) Then Randoop's output depends on the iteration order of a `HashSet`. The developers fixed this by changing `HashSet` to `LinkedHashSet` (commit c975a9f7, shown in fig. 2). The Determinism Checker confirmed that 24 other uses of `new HashSet` were acceptable, as were 18 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 both the problems 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.

Moderate issues: Nondeterministic diagnostic output (**Comparator bug** is user-visible on stdout in the default configuration).

- **HashMap bug**: Randoop iterated over a `HashMap` in arbitrary order, making the diagnostic output difficult to compare across different executions. The class already implemented `Comparable`, so the developers changed `methodWeights.keySet()` to `new TreeSet<>(methodWeights.keySet())` in a `for` loop (commit f212cc7e).
- **Comparator bug**: Randoop prints a list of methods in code under test that might be flaky, sorted by a flakiness metric. This list was itself nondeterministic, when Randoop considered two methods to be equally likely to be flaky. The developers added a secondary sort key to a comparator (commit 3d6cfb33).
- **Library bug**: The Jacoco library uses a `HashMap` internally and returns a collection built from it. This led to nondeterministic diagnostic output when Randoop iterated over the collection. The Randoop developers sorted before iterating (commit 97828027).

Minor issues: Hash codes and timestamps. The Randoop developers may have overlooked these issues because their log-postprocessing tools remove timestamps and some hash codes from the log.

- **Hash code bug**: Diagnostic output printed a hash code for brevity. The developers changed it to have deterministic output (commit 661a4970). This is similar to problems the Randoop developers fixed in the past.
- **Timestamp bug**: Diagnostic output printed a timestamp. The Randoop developers fixed it by making the diagnostic code obey an existing option about whether to print timestamps (commit a460df97).
- **toString bugs**: Four classes inherited the `Object.toString()` implementation, so they printed nondeterministically. The developers defined `toString` methods (commit f8bdf992).
- **Formatting bug**: Diagnostic output used `ObjectContract.toString()`, which is inherited from `Object`. The developers changed the call to `toCodeString()`, which is deterministic and is more informative (commit dff32159).

Unfixable issues: The Determinism Checker issued 2 other true positive warnings because Randoop processes Java code as part of its input. The Determinism Checker identified that the code under test might behave nondeterministically. The Randoop developers could do nothing about these problems. Randoop is documented to behave nondeterministically only if the code under test is also nondeterministic.

B. Case Study 2: Checkstyle

The Checkstyle bugs were due to dependence on system properties (6 instances), nondeterministic logging (5 instances), and nondeterministic exception messages (2 instances). Of the 5 nondeterministic logging instances, one was due to iteration over an `OrderNonDet` collection. We suggested a fix for this bug which was accepted by the developers of Checkstyle (commit 5d2df145).

C. Case Study 3: Checker Framework dataflow analysis

The Determinism Checker revealed 12 instances in which the control flow graph data structure is nondeterministic. These are similar to the problem that we encountered when building the Determinism Checker: we had difficulty debugging because small changes in one part of the graph changed other parts. It also significantly changed logging output and error messages by affecting iteration order. We did not discover a case in which an algorithm's output was semantically different due to this nondeterminism. The maintainers fixed all of these (commits 601b6b58, 3057728a, 8e7287b0, 18f22f83, 67702a13, 0a0ea102).

The Determinism Checker revealed 31 instances in which debug output was nondeterministic because it included hash codes. The maintainers fixed these by assigning each object a unique ID that is printed instead of a hash code (commits bcba3cb7, 24148f91, 0ffe4902). The ID is based on order of creation, so it is deterministic across runs.

D. Case Study 4: plume-lib utilities

The Determinism Checker found 16 determinism bugs across the plume-lib utilities. One of the true positives is because of nondeterminism in logging output (commit 1a9ad3bd). The remainder are in normal user-visible output, and their causes are use of nondeterministic `toString` (5), the file system (3), system properties (2), mutating polymorphic collections (3), and collection ordering (2). The file system nondeterminism is dependence on files in the user's home directory; we did not count merely reading files passed on the command line as nondeterminism.

E. False positive warnings

The Determinism Checker issued a total of 735 false positive warnings across all benchmarks, or about 1 for every 122 lines of code. The most common reasons (responsible for 57% of false positive warnings) were:

1) (24%) An algorithm is used that does not depend on the ordering of its input, but the Determinism Checker cannot verify this. For instance, the elements of an `@OrderNonDet` list during iteration are `@NonDet`, but some computations (sum, max, searching, etc.) are `@Det`. Other instances of order insensitive operations include merging collections and mutating all elements of an `@OrderNonDet` collection deterministically.
2) (12%) All classes that implement an interface define `toString` to return `@Det String`, but the `toString` method of the interface is not so annotated. This is the case for the `java.lang.reflect.Type` interface. Some of the false positives in this category were due to calling `Object.toString` in contexts where we could not establish

whether the invoked `toString` method was deterministic. We counted these as false positives, but the code is error-prone: changes anywhere in the code could change which values flow to the invocations, making them nondeterministic. As part of future work, we could enhance the Determinism Checker with an analysis to track which expressions have a run-time class that overrides `toString` deterministically. This will eliminate these false positives, or it will show them to be true positives.

3) (6%) The Determinism Checker should relax conservative rules when it is safe to do so. For example, it should be legal to pass a `@Det List` to a method that expects an `@OrderNonDet List`, if the method never mutates its input.
4) (3%) Uses of caches. Even if a cache is populated with nondeterministic keys, so long as the key–value mapping is deterministic, looking up a Det key yields a Det value.
5) (2%) The Determinism Checker cannot verify a method that iterates over all the elements of an `@OrderNonDet` collection to create another `@OrderNonDet` collection.
6) (2%) Array sorting can type-refine an array from `@OrderNonDet` to `@Det`, but only if there are no aliases whose type is not refined. Our type system does not incorporate an alias analysis, so it forbids the type refinement to avoid a type loophole.
7) (1%) Iterating over a `@PolyDet` collection to create or modify another `@PolyDet` collection. For example, the following code is safe, but the call to `add` does not type check because variable `elt` has type `@PolyDet("up")`.

```
void m(@PolyDet List<@PolyDet String> input) {
    @PolyDet List<@PolyDet String> output =
    new @PolyDet ArrayList<>();
    for (String elt : input) {
        output.add(elt);
    }
}
```

8) (1%) A class type parameter has upper bound `@PolyDet`, but the Determinism Checker does not always instantiate it with the most precise type. For example, if a method has a `@Det` receiver, inside that method the upper bound can be treated as `@Det`.
9) (1%) The Determinism Checker should treat `@PolyDet("up")` as equivalent to `@PolyDet` for non-collection types.
10) (1%) If a class is declared as `@Det`, then any instance with `@PolyDet` type should also be treated as `@Det` rather than as `@PolyDet`.
11) (1%) An object has a `toString` method that returns `@Det` or `@PolyDet`, but the Determinism Checker's analysis loses track of this fact before the call to `toString`, so the Determinism Checker issues a warning.
12) (1%) A method iterates over an `@OrderNonDet` collection and calls a log method that uses a `SortedSet` in its implementation.
13) (1%) the Determinism Checker flagged a code pattern that is illegal in general — assigning a `Det` value to an `OrderNonDet` variable — but is safe in these specific instances because the value is immutable or there is no aliasing.

2% of the false positives are caused by a bug in our implementation (https://github.com/t-rasmud/checker-framework/issues/219.)

[15] discusses how to improve the Determinism Checker to address some of these false positive.

F. Annotation effort

The number of annotations — one per 17 lines of code — is much higher than we would prefer. Nonetheless, it compares favorably to the extensive effort by the Randoop developers (section IV-A). Moreover, the Determinism Checker found issues in large software (Randoop, Checkstyle, and CF Dataflow) that the developers did not. As another point of comparison, the code contains fewer total determinism type qualifiers than Java generic type arguments. In other words, Java generics cause more clutter than determinism types do.

V. Comparison to NonDex

The state of the art in flaky test detection is NonDex [2]. Section IX explains how NonDex works. This section compares the errors reported by NonDex and the Determinism Checker.

A. Case study with NonDex

We ran NonDex on versions of the projects that contain all 87 nondeterminism bugs that the Determinism Checker found. NonDex found none of those bugs. It did find two flaky tests, both in Checkstyle. In each case the nondeterministic code was in the test, not in Checkstyle proper. Our case study did not detect them because we ran the Determinism Checker on each project's source code but not its tests.

Many of the bugs are not detectable by NonDex. For example, in Randoop, only **HashSet bug** and **Classpath bug** are covered by test cases; apparently the Randoop developers had already found most of the nondeterminism problems that are covered by a test case. The reason for nondeterminism in **Classpath bug** was a call to the `System.getProperty()` method, which is not modeled by NonDex.

B. The Determinism Checker on NonDex benchmarks

Section V-A shows that the Determinism Checker finds errors that NonDex does not. This section determines whether NonDex finds errors that the Determinism Checker does not. Its authors ran NonDex on 195 open-source projects, and NonDex found flaky tests in 21 of them [2]. The flakiness that NonDex found was due to 7 methods (`getDeclaredFields`, `getDeclaredMethods`, `getFields`, `getZoneStrings`, `entrySet`, `keySet`, `values`) in 3 classes (`Class`, `DateFormatSymbols`, `HashMap`).

For every compilable project where the NonDex paper stated a source of flakiness, we ran the Determinism Checker on the part of the project that the NonDex authors determined as flaky. In every case, the Determinism Checker issued a warning on the nondeterministic code. In other words, the Determinism Checker's recall was 100%.

```
static public FieldAccess get(Class type) {
    ...
    while (nextClass != Object.class) {
    Field[] declaredFields = nextClass.getDeclaredFields();
    ...
```

Fig. 3: Nondeterministic code from reflectasm. `getDeclaredFields` returns its result in arbitrary order.

Figure 3 shows a sample of nondeterministic code from these benchmarks. We annotated the type of field `declaredFields` as `@Det Field @Det []`. That type means a deterministic array of deterministic `Field`s, analogously to `@Det List<@Det Field>`. The Determinism Checker issued a warning at the assignment, because `getDeclaredMethods` returns `@Det Field @OrderNonDet []`, which is an order-nondeterministic array of deterministic `Field`s.

The NonDex authors state "we found that manually inspecting these failures was rather challenging, and we leave it as future work to automate debugging test failures." The Determinism Checker reports source locations which makes it easier for the programmer to fix issues, and the annotation effort serves as valuable documentation and prevents regressions.

VI. Comparison to DeFlaker

DeFlaker [3], like NonDex, reports tests that could be flaky. We were unable to run DeFlaker on any of our case studies (other than Checkstyle which we chose from DeFlaker's experiments), because DeFlaker works with projects built with Maven, but the projects other than Checkstyle use Gradle as their build system. The Determinism Checker found 13 bugs in Checkstyle (section IV-B) whereas DeFlaker found 1 [3].

A. The Determinism Checker on DeFlaker benchmarks

DeFlaker found 87 previously unknown flaky tests in 93 projects that were being actively developed at the time the paper was written. The authors reported 19 of these tests, out of which 7 were addressed by the maintainers of those projects [3]. We ran the Determinism Checker on the part of each of these codebases where the reported bug was fixed, as in section V-B. The Determinism Checker reports errors at the source of nondeterminism whereas DeFlaker reports the test case where this nondeterminism manifests. The rationale for choosing these 7 tests is that we could perform a fair comparison between the output of the Determinism Checker and the root cause reported by the developers in the respective issue trackers.

Four of the seven flaky tests (two in `achilles`, one each in `jackrabbit-oak` and `togglz`) were caused by a race condition, which the Determinism Checker cannot detect. (This is a strength of DeFlaker over the Determinism Checker.) The Determinism Checker also found the source of flakiness in `checkstyle`. This bug was in a test case that treated an array returned by `Class.getDeclaredConstructors()` as deterministic. This is erroneous because `getDeclaredConstructors` returns an order-nondeterministic array. We were unable to build `togglz` and `nutz` which had one flakiness issue each. However, we extracted the source code causing the flakiness in these repositories into test cases after looking at the corresponding issues on GitHub. The Determinism Checker detected the errors.

VII. Discussion

While the overhead of annotation for our approach is high, the benefits are also high. Ours is the only approach that discovers all determinism errors and guarantees that no more remain. The trade-off may not be worthwhile for every programmer and for every program. When determinism is important, our approach is easier and more effective than testing-based approaches.

Future work, such as type inference, can further improve our approach, making it more attractive to programmers. Type inference can reveal what the program's behavior is, but not whether that behavior is desired. To find bugs requires comparing the program's behavior to a specification. In our specification-and-verification approach, the programmer provides the specification, and the tool does the verification. The programmer's specification may permit nondeterminism in some parts of the program. An alternative would be for a tool to guess a specification and report wherever the program deviates from the guessed specification. Such an approach would be easier for programmers to use. However, this approach is inherently unsound and incomplete, so it does not meet our design goal of soundness. In addition, such an approach requires access to the whole program (including any libraries or clients it might be linked against), and it often has poor scalability. Future work could compare such an approach to ours. We also see great value in specifying some parts of the program and using inference on the rest, and future work could explore such a combination.

As an alternate design strategy, one could imagine providing a different deterministic implementation of the collection library methods. However, determinism is not necessary or desirable in all parts of a program. For example, a map that is not iterated over has no need for deterministic order. A deterministic version of map iteration would be less performant and would be incompatible with the assumptions of existing programs. This approach does not address other types of nondeterminism, such as coin-flipping, dates and times, system properties, etc. This approach also does not address nondeterminism in the user program.

VIII. Threats to validity

Our type system does not capture nondeterminism from concurrency. It could be combined with a type system for concurrency (see section IX).

In our case study, we disabled two checks in the Determinism Checker because they led to many false positives. One check gives all caught exceptions `NonDet` type, to account for the fact that unchecked libraries might use nondeterministic values in thrown exceptions. The other check requires conditional expressions to be `Det`, to prevent "implicit flows". Implicit flows are a well-known challenge for dataflow analysis, and standard approaches [32], [33] lead to imprecise abstract values (e.g., in a taint analysis, most of the program state becomes tainted). A programmer can work around the problem by declaring more types to be `Det` rather than `PolyDet`, but that reduces the contexts in which a library can be used.

The Determinism Checker only examines the code it is run on. Unchecked libraries with incorrect specifications might introduce nondeterminism even if the Determinism Checker issues no warnings. The Determinism Checker is sound with respect to reflection.

The case studies found important previously unknown errors, but their results might not generalize to other programs. We mitigated this problem by showing that the Determinism Checker finds a superset of the non-concurrency nondeterminism identified by other tools.

IX. Related work

The state of the art for detecting nondeterministic tests is NonDex [2]. NonDex uses a hand-crafted list of 47 methods (25 unique method names) in 13 classes as potential sources of flakiness. For each of the identified methods, the authors built models that return different results when called consecutively. A modified JVM then runs a given test multiple times and reports the test as being flaky if it observes diverging test output. While this approach produces precise results, it requires manual inspection and considerable debugging effort to locate and fix the source of flakiness. The Determinism Checker, in contrast, reports the cause of nondeterminism (a line of code) at compile time requiring little debugging effort. However, the Determinism Checker requires much more upfront effort, and it produces false positive warnings. NonDex's approach of identifying and modeling methods with nondeterministic specifications is analogous to our library specifications. So far, we have annotated 1034 methods across 59 classes in the JDK and JUnit.

DeFlaker [3] is another approach for flaky test detection. It relies on a version control history. It computes a diff of the code covered in the current version and the previous one. If there exists a test case whose code coverage does not include this diff but still produces different results on the two versions being compared, the test case is flagged as being flaky. This approach does not require JVM modifications and integrates easily with production software. DeFlaker reported 19 previously unknown bugs in open source projects, 7 of which were addressed by the developers of these projects. DeFlaker is agnostic to the code under test and can therefore report flakiness arising out of concurrency, which the Determinism Checker cannot.

Nondeterminism in tests is of interest to both researchers and software developers alike [34], [5]. Empirical analysis [1] suggests that most of the flakiness in tests is caused by async await, concurrency, or test order dependency. Our approach is complementary to such techniques and aims to prevent nondeterminism from causing harmful effects.

Eilers et al. [35] propose constructing product programs to help verify hyperproperties (i.e properties that reason about multiple program executions). This dynamic approach checks hyperproperties over k execution traces by comparing the program state after executing the product program with that of the original program. Specifying properties over collections would require quantification over every element in the array. In [36], the authors study the effect of nondeterminism in MapReduce programs with a specific focus on nondeterminism caused by non-commutative reducers.

Several techniques have been proposed to test whether a deterministic implementation conforms to its nondeterministic finite state machine [37], [38], [39], [40]. [41] presents an approach that can automatically verify properties in branching time temporal logic systems that are inherently nondeterministic. Bocchino et al. [42], [43] present a type-and-effect system that provides compile-time determinism guarantees for parallelism. They ignore other sources of nondeterminism. Our work is complementary and addresses a previously overlooked problem.

Failing tests that are unrelated to code changes can be expensive in monetary costs and in developer effort. [44] proposes techniques to classify tests as false alarms if they are known to be caused by testing infrastructure or other environment issues. [45] presents an approach that detects brittle assertions in tests by performing a taint analysis on inputs classified as controlled and uncontrolled. [46] investigates the effects of the test independence assumption on other techniques such as test prioritization, selection, etc. Other approaches [47], [48] analyze test dependencies and either prevent them or use this information for other optimizations. The approaches in [49], [50] focus on differentiating bugs due to tests from those caused by source code.

X. Conclusion

We designed a type system that expresses determinism specifications for sequential programs. To the best of our knowledge, ours is the first compile-time verification approach addressing the problem of nondeterminism in sequential programs. We implemented our type system in Java and applied it to real world software. Our tool, the Determinism Checker, found errors that the developers had missed, despite spending extensive effort on the problem of nondeterminism. In experiments, The Determinism Checker found a superset of the nondeterminism bugs in sequential programs that were found by the state of the art flaky test detectors, NonDex [2] and DeFlaker [3].

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] M. T. Rahman and P. C. Rigby, "The impact of failing, flaky, and high failure tests on the number of crash reports associated with Firefox builds," in *ESEC/FSE 2018: The ACM 26th joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Lake Buena Vista, FL, USA, November 2018, pp. 857–862.

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

[6] 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.

[7] 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.

[8] R. Tene, https://blogs.dropbox.com/tech/2018/05/how-were-winning-the-battle-against-flaky-tests/, 2018.

[9] M. Shah, https://docs.microsoft.com/en-us/azure/devops/learn/devops-at-microsoft/eliminating-flaky-tests, 2017.

[10] Y. Chen, S. Zhang, Q. Guo, L. Li, R. Wu, and T. Chen, "Deterministic replay: A survey," *ACM Computing Surveys*, vol. 48, no. 2, September 2015.

[11] 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.

[12] P. Cousot, "Types as abstract interpretations," in *POPL '97: Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, France, January 1997, pp. 316–331.

[13] J. S. Foster, R. Johnson, J. Kodumal, and A. Aiken, "Flow-insensitive type qualifiers," *ACM Transactions on Programming Languages and Systems*, vol. 28, no. 6, pp. 1035–1087, November 2006.

[14] A. Igarashi, B. C. Pierce, and P. Wadler, "Featherweight Java: a minimal core calculus for Java and GJ," *ACM Transactions on Programming Languages and Systems*, vol. 23, no. 3, pp. 396–450, May 2001.

[15] R. Mudduluru, J. Waataja, S. Millstein, and M. D. Ernst, "Verifying determinism in sequential programs (extended version)," University of Washington, Tech. Rep. TR-UW-CSE-2021-02-01, 2021.

[16] D. Ancona, G. Lagorio, and E. Zucca, "Type inference for polymorphic methods in Java-like languages," in *Theoretical Computer Science: Proceedings of the 10th Italian Conference on ICTCS '07*, Rome, Italy, October 2007, pp. 118–129.

[17] N. Cameron, S. Drossopoulou, and E. Ernst, "A model for Java with wildcards," in *ECOOP 2008 — Object-Oriented Programming, 22nd European Conference*, Paphos, Cyprus, July 2008, pp. 2–26.

[18] M. Abadi, B. Pierce, and G. Plotkin, "Faithful ideal models for recursive polymorphic types," in *LICS '89: Proceedings of the Fourth Annual IEEE Symposium on Logic in Computer Science*, Pacific Grove, CA, USA, June 1989, pp. 216–225.

[19] G. D. Plotkin and M. Abadi, "A logic for parametric polymorphism," in *TLCA 2003: International Conference on Typed Lambda Calculi and Applications*, Utrecht, The Netherlands, March 1993, pp. 361–375.

[20] *The Checker Framework Manual: Custom pluggable types for Java*, http://CheckerFramework.org/.

[21] S. Hunt and D. Sands, "On flow-sensitive security types," in *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '06. New York, NY, USA: ACM, 2006, pp. 79–90. [Online]. Available: http://doi.acm.org/10.1145/1111037.1111045

[22] M. D. Adams, A. W. Keep, J. Midtgaard, M. Might, A. Chauhan, and R. K. Dybvig, "Flow-sensitive type recovery in linear-log time," in *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '11. New York, NY, USA: ACM, 2011, pp. 483–498. [Online]. Available: http://doi.acm.org/10.1145/2048066.2048105

[23] Y. Sui and J. Xue, "On-demand strong update analysis via value-flow refinement," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 460–473. [Online]. Available: http://doi.acm.org/10.1145/2950290.2950296

[24] R. Mudduluru, J. Waataja, S. Millstein, and M. D. Ernst, "subject programs and tool for the paper: Verifying Determinism in Sequential Programs," Feb. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.4536285

[25] https://github.com/randoop/randoop.

[26] https://github.com/randoop/randoop/issues, 2010–2020.

[27] https://groups.google.com/forum/#!forum/randoop-developers and https://groups.google.com/forum/#!forum/randoop-discuss, 2010–2020.

[28] https://github.com/checkstyle/checkstyle.

[29] https://github.com/typetools/checker-framework/tree/master/dataflow.

[30] https://github.com/plume-lib.

[31] https://randoop.github.io/randoop/manual/index.html, March 2020, version 4.2.3.

[32] M. G. Kang, S. McCamant, P. Poosankam, and D. X. Song, "Dta++: Dynamic taint analysis with targeted control-flow propagation," in *NDSS*, 2011.

[33] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," in *PLDI 2014: Proceedings of the ACM SIGPLAN 2014 Conference on Programming Language Design and Implementation*, Edinburgh, UK, June 2014, pp. 259–269.

[34] M. Fowler, https://martinfowler.com/articles/nonDeterminism.html, 2011.

[35] M. Eilers, P. Müller, and S. Hitz, "Modular product programs," *ACM Trans. Program. Lang. Syst.*, vol. 42, no. 1, Nov. 2019. [Online]. Available: https://doi.org/10.1145/3324783

[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 *Companion Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion 2014. New York, NY, USA: ACM, 2014, pp. 44–53. [Online]. Available: http://doi.acm.org/10.1145/2591062.2591177

[37] A. Petrenko, N. Yevtushenko, and G. v. Bochmann, "Testing deterministic implementations from nondeterministic fsm specifications," in *Testing of Communicating Systems: IFIP TC6 9th International Workshop on Testing of Communicating Systems Darmstadt, Germany 9–11 September 1996*. Boston, MA: Springer US, 1996, pp. 125–140. [Online]. Available: https://doi.org/10.1007/978-0-387-35062-2_10

[38] A. Petrenko, N. Yevtushenko, A. Lebedev, and A. Das, "Nondeterministic state machines in protocol conformance testing," in *Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test Systems VI*. Amsterdam, The Netherlands, The Netherlands: North-Holland Publishing Co., 1994, pp. 363–378. [Online]. Available: http://dl.acm.org/citation.cfm?id=648128.761244

[39] T. Savor and R. E. Seviora, "Supervisors for testing non-deterministically specified systems," in *Proceedings International Test Conference 1997*, Nov 1997, pp. 948–953.

[40] R. M. Hierons and M. Harman, "Testing conformance of a deterministic implementation against a non-deterministic stream x-machine," *Theor. Comput. Sci.*, vol. 323, no. 1-3, pp. 191–233, Sep. 2004. [Online]. Available: http://dx.doi.org/10.1016/j.tcs.2004.04.002

[41] B. Cook and E. Koskinen, "Reasoning about nondeterminism in programs," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: ACM, 2013, pp. 219–230. [Online]. Available: http://doi.acm.org/10.1145/2491956.2491969

[42] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian, "A type and effect system for deterministic parallel java," in *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '09. New York, NY, USA: ACM, 2009, pp. 97–116. [Online]. Available: http://doi.acm.org/10.1145/1640089.1640097

[43] R. L. Bocchino, Jr., S. Heumann, N. Honarmand, S. V. Adve, V. S. Adve, A. Welc, and T. Shpeisman, "Safe nondeterminism in a deterministic-by-default parallel language," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '11. New York, NY, USA: ACM, 2011, pp. 535–548. [Online]. Available: http://doi.acm.org/10.1145/1926385.1926447

[44] K. Herzig and N. Nagappan, "Empirically detecting false test alarms using association rules," in *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 39–48. [Online]. Available: http://dl.acm.org/citation.cfm?id=2819009.2819018

[45] C. Huo and J. Clause, “Improving oracle quality by detecting brittle assertions and unused inputs in tests,” in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 621–631. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635917

[46] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin, “Empirically revisiting the test independence assumption,” in *ISSTA 2014, Proceedings of the 2014 International Symposium on Software Testing and Analysis*, San Jose, CA, USA, July 2014, pp. 385–396.

[47] J. Bell, G. Kaiser, E. Melski, and M. Dattatreya, “Efficient dependency detection for safe Java test acceleration,” in *ESEC/FSE 2015: The 10th joint meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Bergamo, Italy, September 2015, pp. 770–781.

[48] A. Gyori, A. Shi, F. Hariri, and D. Marinov, “Reliable testing: Detecting state-polluting tests to prevent test dependency,” in *ISSTA 2015, Proceedings of the 2015 International Symposium on Software Testing and Analysis*, Baltimore, MD, USA, July 2015, pp. 223–233.

[49] D. Hao, T. Lan, H. Zhang, C. Guo, and L. Zhang, “Is this a bug or an obsolete test?” in *ECOOP 2013 – Object-Oriented Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 602–628.

[50] A. Vahabzadeh, A. M. Fard, and A. Mesbah, “An empirical study of bugs in test code,” in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2015, pp. 101–110.