Making Sequential Programs Deterministic

ANONYMOUS AUTHOR(S)

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 if the test depends on iteration order for asserting correctness.

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 the typings for collections; and enhancements to polymorphism that improve precision when analyzing collection operations. While state-of-the-art nondeterminism detection tools rely on observing runtime output, our approach aims to verify determinism at compile time, thereby providing stronger soundness guarantees.

We implemented our type system for Java. Our type checker, Garçon, 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. In an experiment, it found all of the non-concurrency-related nondeterminism that was found by state-of-the-art dynamic approaches to detecting flaky tests.

Additional Key Words and Phrases: nondeterminism, type system, verification, specification, hash table

1 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 [Bell et al. 2018; Luo et al. 2014; Rahman and Rigby 2018; Shi et al. 2016; Sudarshan 2012] that sometimes pass and sometimes fail. 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 [Yu et al. 2012; Zeller 1999].
- Nondeterminism reduces users' and developers' **trust** in a program's output [Shah 2017; Tene 2018].

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.

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. If our analysis issues no warnings, then the program produces the same output when executed twice on the same inputs. This guarantee is modulo the limitations of the analysis, notably concurrency, implicit control flow,

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

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, Garçon, confirmed that 25 other uses of `new HashSet` were acceptable, as were 15 uses of `new HashMap`.

and unanalyzed libraries (see section 7). 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. There is no need for a custom runtime system nor rerunning a program multiple times—nor even running it once. 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.

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. 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 [Shi et al. 2016] 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 can reveal unwarranted dependence on those methods. DeFlaker [Bell et al. 2018] 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 [Briski et al. 2008].

Our analysis uses three main abstractions:

`NonDet` represents values that might differ from run to run.

`OrderNonDet` represents collections that are guaranteed to contain the same elements but whose iteration order is nondeterministic.

`Det` represents values that will be the same across executions.

Programmers can write these to specify their program's behavior. Our full analysis contains other features that increase expressiveness, permitting it to scale to programs of practical interest.

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 Garçon.
(3) In a case study, we annotated several libraries, including the collection classes and other parts of the JDK (Java's standard library). This provides a formal, machine-readable specification for the libraries, and it demonstrates the expressiveness of our type system.
(4) In another 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. Garçon discovered 5

$$
\begin{array}{lcll}
C & ::= & \texttt{Int} \mid \texttt{String} \mid \texttt{Collection} \mid \texttt{List} \mid \texttt{Set} \mid \texttt{Map} & \text{class name} \\
\kappa & ::= & \texttt{NonDet} \mid \texttt{OrderNonDet} \mid \texttt{Det} & \text{type qualifier} \\
\tau & ::= & \kappa\ \beta & \text{type} \\
\beta & ::= & C \mid C\langle\overline{\tau}\rangle & \text{unqualified type}
\end{array}
$$

Fig. 2. Grammar of types

instances of nondeterminism that the developers had overlooked. The developers fixed all these issues when we reported them. Figures 1 and 14 show two examples.

(5) We compared our tool, Garçon, against state-of-the-art flaky test detectors, on their benchmarks. Garçon found all the non-concurrency nondeterminism found by the other tools.

2 A TYPE SYSTEM FOR DETERMINISM

This section presents the key aspects of our type system in the context of a core calculus for an object-oriented language.

2.1 Preliminaries and notation

One way to view a type is as a set of values. A type abstracts or restricts the set of possible run-time values that an expression may evaluate to and the operations that may be performed. A programming language provides *basetypes*, such as `Int`. A *type qualifier* on a basetype adds additional constraints; that is, it reduces the size of the set of values. An example type qualifier is `Positive`, and a type (which combines a qualifier and a basetype) is `Positive Int`. A polymorphic type abstraction such as `List` can be instantiated by a type argument, as in `List`⟨`Positive Int`⟩.

Figure 2 formalizes these notions. For simplicity of exposition, fig. 2's β is not a basetype as described above: β lacks a top-level type, but any type arguments are full types τ. In our core calculus, a type τ always has a qualifier. Our implementation, Garçon, uses inference (section 3.5) and defaults (section 3.4) to permit users to omit type qualifiers.

A type checker verifies the types written in a program. Our type system checks all the standard typing rules including the ones shown in fig. 3, which shows a sample of standard typing rules for an object-oriented programming language.

A type qualifier constrains the set of possible run-time values, that is, $\kappa\ \beta \sqsubseteq \beta$. 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.

Type qualifier systems are sometimes defined independently of the underlying type system. In our type system, there are interactions between the basetypes and the type qualifiers. Defining them together improves precision, which is important in practice.

2.2 Determinism types

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.

Figures 4 and 5 show the subtyping relationship among the qualifiers.

$$\frac{\tau_1 \sqsubseteq \tau_2 \quad \tau_2 \sqsubseteq \tau_3}{\tau_1 \sqsubseteq \tau_3} \qquad \frac{\Gamma \vdash x : \tau_1 \quad \tau_1 \sqsubseteq \tau_2}{\Gamma \vdash x : \tau_2} \text{ SUBTYPING}$$

$$\frac{\Gamma \vdash f : \overline{\tau_i} \longrightarrow \tau_r \quad \Gamma \vdash \overline{a_i : \tau_i}}{\Gamma \vdash f(\overline{a_i}) : \tau_r} \text{ CALL}$$

$$\frac{C_1 \sqsubseteq C_2}{C_1\langle\overline{\tau_i}\rangle \sqsubseteq C_2\langle\overline{\tau_i}\rangle} \text{ INVARIANT TYPE ARG}$$

Fig. 3. A sampling of standard typing rules. We omit other standard rules, for brevity.

$$\frac{}{\texttt{Det} \sqsubseteq \texttt{OrderNonDet}} \qquad \frac{}{\texttt{OrderNonDet} \sqsubseteq \texttt{NonDet}}$$

$$\frac{\kappa_1 \sqsubseteq \kappa_2}{\kappa_1\beta \sqsubseteq \kappa_2\beta} \qquad \frac{\beta_1 \sqsubseteq \beta_2}{\kappa\ \beta_1 \sqsubseteq \kappa\ \beta_2}$$

Fig. 4. Subtyping rules for our type system's type qualifiers. $\sqsubseteq$ is overloaded for classes, type qualifiers, types, and unqualified types β.

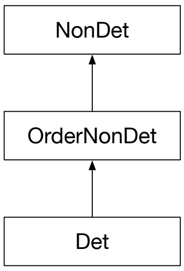

Fig. 5. Determinism type qualifier hierarchy

`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. The next section discusses type well-formedness in detail.

2.3 Collection types

2.3.1 Type well-formedness. Figure 6 shows the type well-formedness, or type validity, rules of our type system.

The element type of a `NonDet` collection must be `NonDet`. For other collections, the type qualifier on a collection's element must be a subtype or equal to the collection's type qualifier (COLLECTION rules of fig. 6). Note that the ORDERNONDET rules also permit `Det` collections, because `Det` $\sqsubseteq$ `OrderNonDet`. Figure 7 restates the COLLECTION rules of fig. 6.

$$\frac{\kappa \in \{\texttt{Det}, \texttt{NonDet}\} \quad \beta \not\sqsubseteq \texttt{Collection} \quad \beta \not\sqsubseteq \texttt{Map}}{\vdash: \kappa\ \beta}\ \text{NONCOLLECTION}$$

$$\frac{\kappa_c \sqsubseteq \texttt{OrderNonDet} \quad C \sqsubseteq \texttt{Collection} \quad \kappa_e \sqsubseteq \kappa_c \quad \vdash: \kappa_e\ \beta_e}{\vdash: \kappa_c\ C\langle \kappa_e\ \beta_e \rangle}\ \text{ORDERNONDET COLLECTION}$$

$$\frac{C \sqsubseteq \texttt{Collection} \quad \vdash: \texttt{NonDet}\ \beta_e}{\vdash: \texttt{NonDet}\ C\langle \texttt{NonDet}\ \beta_e \rangle}\ \text{NONDET COLLECTION}$$

$$\frac{\kappa_m \sqsubseteq \texttt{OrderNonDet} \quad C \sqsubseteq \texttt{Map} \quad \kappa_k \sqsubseteq \kappa_m \quad \kappa_v \sqsubseteq \kappa_m \quad \vdash: \kappa_k\ \beta_k \quad \vdash: \kappa_v\ \beta_v}{\vdash: \kappa_m\ C\langle \kappa_k\ \beta_k, \kappa_v\ \beta_v \rangle}\ \text{ORDERNONDET MAP}$$

$$\frac{C \sqsubseteq \texttt{Map} \quad \vdash: \texttt{NonDet}\ \beta_k \quad \vdash: \texttt{NonDet}\ \beta_v}{\vdash: \texttt{NonDet}\ C\langle \texttt{NonDet}\ \beta_k, \texttt{NonDet}\ \beta_v \rangle}\ \text{NONDET MAP}$$

Fig. 6. Type well-formedness

		Element type		
		NonDet	OrderNonDet	Det
Collection type	NonDet	valid	invalid	invalid
	OrderNonDet	invalid	valid	valid
	Det	invalid	invalid	valid

Fig. 7. Valid Collection declarations. The Collection's type qualifier must be a supertype or equal to the element type.

Some examples of valid types are:

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

These types are invalid:

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

If the type `NonDet List⟨Det String⟩` were permitted, the following type hole would exist:

```
Det    List⟨Det String⟩ ddlist
NonDet List⟨Det String⟩ ndlist = ddlist   // assignment permitted by subtyping rules
NonDet Int ni = ...
ndlist[ni] = "anywhere"
```

In this code snippet, the deterministic string `"anywhere"` is placed in the list at an arbitrary (that is, nondeterministic) index. This must be prohibited, because it unsoundly permits the deterministic list `ddlist` to differ from execution to execution. Our type system does so by forbidding the type `NonDet List⟨Det String⟩`.

2.3.2 *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. (As shown by types of methods GET, ITERATOR, NEXT in fig. 8)
(2) Size-related operations return a deterministic result. This also affects queries of whether an iterator has more elements.

$$\text{GET}: \forall \tau\ \kappa.\ \texttt{OrderNonDet}\ C\langle\tau\rangle\ \times \kappa\ \texttt{Int} \longrightarrow \texttt{NonDet Int}$$

$$\text{SET}: \forall \kappa\ \beta.\ \texttt{OrderNonDet}\ C\langle\kappa\ \beta\rangle\ \times \texttt{Det Int} \times \kappa\ \beta \longrightarrow \texttt{NonDet}\ \beta$$

$$\text{SIZE}: \forall \tau.\ \texttt{OrderNonDet}\ C\langle\tau\rangle \longrightarrow \texttt{Det int}$$

$$\text{ITERATOR}: \forall \tau.\ \texttt{OrderNonDet}\ C\langle\tau\rangle \longrightarrow \texttt{OrderNonDet Iterator}\langle\tau\rangle$$

$$\text{NEXT}: \forall \kappa\ \beta.\ \texttt{OrderNonDet Iterator}\langle\kappa\ \beta\rangle \longrightarrow \texttt{NonDet}\ \beta$$

$$\text{HASNEXT}: \forall \tau.\ \texttt{OrderNonDet Iterator}\langle\tau\rangle \longrightarrow \texttt{Det boolean}$$

Fig. 8. `OrderNonDet` Collection rules. C is a collection class. These are partial types indicating the methods' behavior on `OrderNonDet` arguments.

(3) If the collection is sorted, or its elements are placed in a collection that does sorting, the result is deterministic.

To state the first point more formally, the typical type for a list access operation, such as `get`, is

$$\forall \tau.\ \texttt{List}\langle\tau\rangle \times \texttt{Int} \longrightarrow \tau$$

but this type is incorrect for our type system. Each of the following is a correct (but overly restrictive) type for `get`:

$$\begin{array}{rl}
\forall \kappa\ \beta.\ \texttt{NonDet List}\langle\kappa\ \beta\rangle \times \texttt{Det Int} \longrightarrow \texttt{NonDet}\ \beta & \text{in this case } \kappa = \texttt{NonDet} \\
\forall \kappa\ \beta.\ \texttt{OrderNonDet List}\langle\kappa\ \beta\rangle \times \texttt{Det Int} \longrightarrow \texttt{NonDet}\ \beta & \text{in this case } \kappa \in \{\texttt{OrderNonDet}, \texttt{Det}\} \\
\forall \kappa\ \beta.\ \texttt{Det List}\langle\kappa\ \beta\rangle \times \texttt{Det Int} \longrightarrow \texttt{Det}\ \beta & \text{in this case } \kappa = \texttt{Det}
\end{array}$$

and no type introduced so far can express this polymorphism. (The typing is actually even more complex, because if *either* argument to `get` is `OrderNonDet` or `NonDet`, then the result is `NonDet`. The above examples only show the case where the index is deterministic.) The typical type for the list access operation only applies if both arguments are `Det`, and our typing must have that as a special case, which is the last line above.

2.4 Polymorphism

This section first describes basic polymorphism over type qualifiers and over basetypes (section 2.4.1). Then, it describes two extensions. One extension resolves `OrderNonDet` to `NonDet` or `Det` where needed (section 2.4.2). The other extension permits use of polymorphic (type) variables without affecting binding of those variables — that is, it affects which instantiation of a (method) abstraction is chosen at a use site (section 2.4.3).

2.4.1 Qualifier and basetype polymorphism. Our type system supports parametric polymorphism [Abadi et al. 1989; Plotkin and Abadi 1993]. A polymorphic abstraction (a class or method) is written and typechecked once. 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. When both are applied, it gives the effect of type polymorphism.

- Type polymorphism has the usual semantics. For example, the type of the identity function is $\forall \tau.\ \tau \longrightarrow \tau$, which can be equivalently written as $\forall \kappa\ \beta.\ \kappa\ \beta \longrightarrow \kappa\ \beta$.

$$\frac{}{\texttt{NonDet}\uparrow = \texttt{NonDet}} \quad \frac{}{\texttt{OrderNonDet}\uparrow = \texttt{NonDet}} \quad \frac{}{\texttt{Det}\uparrow = \texttt{Det}}\ \text{POLYUP}$$

$$\frac{}{\texttt{NonDet}\downarrow = \texttt{NonDet}} \quad \frac{}{\texttt{OrderNonDet}\downarrow = \texttt{Det}} \quad \frac{}{\texttt{Det}\downarrow = \texttt{Det}}\ \text{POLYDOWN}$$

Fig. 9. Polymorphic resolution of $\uparrow$ and $\downarrow$ operators. $\uparrow$ is overloaded for types and qualifiers.

- Basetype polymorphism lets the basetype vary independently of the qualifier, which might be fixed or might be polymorphic. An example is a nondeterministic choose function, which might have type
  choose : $\forall\beta.$ `Set`$\langle$`Det` $\beta\rangle \longrightarrow$ `NonDet` β
  Basetype polymorphism is rarely used on its own. Full type polymorphism is more common, even if the function type decomposes the type parameter and uses the parts independently, as in this more general type for the choose function:
  choose : $\forall\kappa\ \beta.$ `Set`$\langle\kappa\ \beta\rangle \longrightarrow$ `NonDet` β
- Qualifier polymorphism is common. For example, here are types for the `length` method on strings and the addition operation:
  `length :` $\forall\kappa.\ \kappa$ `String` $\longrightarrow \kappa$ `Int`
  `plus :` $\forall\kappa.\ \kappa$ `Int` $\times\ \kappa$ `Int` $\longrightarrow \kappa$ `Int`

Informally, the `length` method above acts as if it had multiple (overloaded) definitions
`length` : `NonDet String` $\longrightarrow$ `NonDet Int`
`length` : `Det String` $\longrightarrow$ `Det Int`
The body of `length` must typecheck at every possible instantiation of its polymorphic function type. At a use site of `length`, the most specific applicable instantiation is chosen.

For example, suppose we have two variables `ni` : `NonDet Int` and `di` : `Det Int`. This table shows, for several method invocations, the most specific applicable instantiation for that invocation and the type of the call expression:

Expression	instantiation	type
`plus(ni, ni)`	`NonDet String` $\longrightarrow$ `NonDet Int`	`NonDet`
`plus(di, di)`	`Det String` $\longrightarrow$ `Det Int`	`Det`
`plus(ni, di)`	`NonDet String` $\longrightarrow$ `NonDet Int`	`NonDet`
`plus(di, ni)`	`NonDet String` $\longrightarrow$ `NonDet Int`	`NonDet`

We adopt the convention that polymorphism is not instantiated in ways that would create invalid types. For example, the `length` and `plus` polymorphic functions would not be instantiated at κ = `OrderNonDet`. This makes no difference for these particular functions: such an instantiation would never be the most specific applicable one. The general rule about instantiations and type validity is important for some of the type rules given in this paper, which would otherwise need complex preconditions that recapitulate the type well-formedness rules.

2.4.2 Polymorphism rules for collections. As described so far, polymorphism cannot express the collection behaviors of section 2.3.2. Consider this potential typing for the `size` method in class `Collection`$\langle\tau_e\rangle$:
`size :` $\forall\kappa.\ \kappa$ `Collection`$\langle\tau_e\rangle \longrightarrow \kappa$ `Int`
It cannot be instantiated at κ = `OrderNonDet` as
`size : OrderNonDet Collection`$\langle\tau_e\rangle \longrightarrow$ `OrderNonDet Int`
because such an instantiation would include the invalid return type `OrderNonDet Int`. This means that the only two instantiations are at κ = `NonDet` and κ = `Det`. An invocation on an `OrderNonDet` collection would choose the `NonDet` instantiation, and the return type at that call site would be `NonDet Int` rather than the desired `Det Int`.

Our type system resolves this problem by introducing two operators over polymorhic (type) variables, ↑ and ↓. The ↑ operator converts `OrderNonDet` to `NonDet` and leaves the other qualifiers unchanged. The upward-pointing arrow is mnemonic for replacing `OrderNonDet` by something higher in the type hierarchy. The ↓ operator is analogous, but it converts `OrderNonDet` to `Det`, which is lower in the type hierarchy. Figure 9 formalizes their behavior.

The correct type for `size` is

`size` : $\forall\kappa.\ \kappa$ `Collection`$\langle\tau_e\rangle \longrightarrow \kappa\!\downarrow$ `Int`

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

`size : NonDet Collection`$\langle\tau_e\rangle \longrightarrow$ `NonDet Int`
`size : OrderNonDet Collection`$\langle\tau_e\rangle \longrightarrow$ `Det Int`
`size : Det Collection`$\langle\tau_e\rangle \longrightarrow$ `Det Int`

The above instantiations implement the semantics of section 2.3.2.

An example use of ↑ is in the `first` method in class `List`$\langle\kappa_e\ \beta_e\rangle$ that returns the first element of a list. Its type is

`first` : $\forall\kappa.\ \kappa$ `List`$\langle\kappa_e\ \beta_e\rangle \longrightarrow \kappa\!\uparrow \beta_e$

which can be instantiated as

`first : NonDet List`$\langle\kappa_e\ \beta_e\rangle \longrightarrow$ `NonDet` β_e
`first : OrderNonDet List`$\langle\kappa_e\ \beta_e\rangle \longrightarrow$ `NonDet` β_e
`first : Det List`$\langle\kappa_e\ \beta_e\rangle \longrightarrow$ `Det` β_e

Discussion of the type of `first`. In the type of `first`, the ↑ is not strictly necessary.

Suppose that `myOndStrings : OrderNonDet List⟨Det String⟩`. The type of `first(myOndStrings)` should be `NonDet String`.

If the type of `first` were

`first` : $\forall\kappa.\ \kappa$ `List`$\langle\kappa_e\ \beta_e\rangle \longrightarrow \kappa\ \beta_e$

then the instantiation

`first : OrderNonDet List`$\langle\kappa$ `String`$\rangle \longrightarrow$ `OrderNonDet String`

would be invalid. At the call `first(myOndStrings)`, the most specific instantiation would be

`first : NonDet List`$\langle\kappa$ `String`$\rangle \longrightarrow$ `NonDet String`

making the type of the expression `NonDet String` as desired. We prefer the type

`first` : $\forall\kappa.\ \kappa$ `List`$\langle\kappa_e\ \beta_e\rangle \longrightarrow \kappa\!\uparrow \beta_e$

because it is more explicit about the method's effect and more instantiations, rather than requiring a reader to reason about valid types.

The given type prevents some invocations. Suppose we have the following variable:

`listOfLists : OrderNonDet List⟨Det List⟨Det String⟩⟩`

Then the invocation `first(listOfLists)` is illegal. Its type would be `NonDet List⟨Det String⟩`, which is an invalid type.

This restriction prevents programmers from writing some code, but we never found this to be a problem in our case studies.

2.4.3 *Differentiating binding and use.*

This section introduces another polymorphism feature by contrasting the types of the `get` and `set` operations on lists.

A simplified type for the list `get` method for lists of strings is

`get` : $\forall\kappa.\ \kappa$ `List`$\langle\kappa\!\downarrow$ `String`$\rangle \times \kappa\!\uparrow$ `Int` $\longrightarrow \kappa\!\uparrow$ `String`

(The actual type permits the qualifier on the type argument to differ from the type qualifier on the list, and it better handles order-nondeterminism, but this type is enough for illustration in this section.)

Its instantiations are:

`get : NonDet List⟨NonDet String⟩` $\times$ `NonDet Int` $\longrightarrow$ `NonDet String`

$\texttt{get} : \texttt{OrderNonDet List}\langle\texttt{Det String}\rangle \times \texttt{NonDet Int} \to \texttt{NonDet String}$
$\texttt{get} : \texttt{Det List}\langle\texttt{Det String}\rangle \times \texttt{Det Int} \to \texttt{Det String}$

The type of get ensures that if either the list or the index is (order) nondeterministic, then the result of the call is nondeterministic.

A similar typing does *not* work for the set function that sets an element of a list. Consider the mutative type

$\texttt{set} : \forall\kappa.\ \kappa\ \texttt{List}\langle\kappa{\downarrow}\ \texttt{String}\rangle \times \kappa{\uparrow}\ \texttt{Int} \times \kappa{\downarrow}\ \texttt{String} \to ()$

which has instantiations

$\texttt{set} : \texttt{NonDet List}\langle\texttt{NonDet String}\rangle \times \texttt{NonDet Int} \times \texttt{NonDet String} \to ()$
$\texttt{set} : \texttt{OrderNonDet List}\langle\texttt{Det String}\rangle \times \texttt{NonDet Int} \times \texttt{Det String} \to ()$
$\texttt{set} : \texttt{Det List}\langle\texttt{Det String}\rangle \times \texttt{Det Int} \times \texttt{Det String} \to ()$

Suppose a client calls this method as follows:

```
Det List⟨Det String⟩ dList = ...
NonDet int random = ...
Det String str = ...
set(dList, random, str)  // should be forbidden
```

The call should be forbidden, since on different runs it may make changes at different indices of a deterministic list. However, the call is permitted. The middle instantiation is applicable, since $\texttt{DetList}\langle\texttt{Det String}\rangle \sqsubseteq \texttt{OrderNonDetList}\langle\texttt{Det String}\rangle$ and $\texttt{Det} \sqsubseteq \texttt{NonDet}$.

Our type system must prevent this behavior. It does so via another variant of qualifier variables, *use*(κ), which represents a *use* of κ that does not affect the *instantiation* of κ.

Ordinarily, the 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

$f : \forall\kappa.\ \kappa\ \texttt{Int} \times \texttt{Det Int} \times \kappa\ \texttt{Int} \times \kappa\ \texttt{Int}$

would be 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, or if the other arguments do not conform to the parameter types, the call does not type-check.) By contrast, function

$f : \forall\kappa.\ \kappa\ \texttt{Int} \times \texttt{Det Int} \times \textit{use}(\kappa)\ \texttt{Int} \times \kappa\ \texttt{Int}$

would be instantiated at the least upper bound of the types of its first and fourth arguments, and the third argument must conform to the instantiation. However, the third argument does not affect the instantiation.

This is especially useful in preventing methods from nondeterministically modifying the state of a deterministic receiver. The correct type for set on List⟨String⟩ is

$\texttt{set} : \forall\kappa.\ \kappa\ \texttt{List}\langle\kappa{\downarrow}\ \texttt{String}\rangle \times \textit{use}(\kappa)\ \texttt{Int} \times \kappa{\downarrow}\ \texttt{String} \to ()$

and the undesired call set(dList, random, str) does not type-check.

(For the curious reader, the actual types of the get and set methods in the JDK are the following.

```
@PolyDet("up") E get(@PolyDet List<E> this, @PolyDet int index);
@PolyDet("up") E set(@PolyDet List<E> this, @PolyDet("use") int index, E element);
```

The syntax is explained in section 3.3.)

2.5 Type inference, dataflow analysis, and type refinement

Some method calls refine the types of their arguments. One example is the sort routine, as shown in fig. 10. One notable fact about the SORT rule is that it requires the type argument to be deterministic (type qualifier Det). This is necessary because of two constraints. First, refinement must not change the type argument, because of invariant type argument subtyping. Second, the only possible element type for a Det collection is Det.

$$\frac{\Gamma \vdash x_{\text{pre}} : \texttt{OrderNonDet List}\langle\texttt{Det}\ \beta\rangle \quad \texttt{sort}(x_{\text{pre}})}{\Gamma \vdash x_{\text{post}} : \texttt{Det List}\langle\texttt{Det}\ \beta\rangle}\ \text{SORT}$$

Fig. 10. Sorting type refinement rules. sort is a method that works by side effect and changes the value—and the type—of its receiver. "x_{pre} stands for a value before the call to sort, and "x_{post} stands for the value after the call.

$$\frac{\begin{array}{c}\Gamma \vdash x : \texttt{OrderNonDet}\ \mathit{Set}\langle\kappa_x\ \beta_x\rangle, y : \texttt{OrderNonDet}\ \mathit{Set}\langle\kappa_y\ \beta_y\rangle \\ \kappa_x\ \beta_x \not\sqsubseteq \texttt{OrderNonDet List} \qquad \kappa_y\ \beta_y \not\sqsubseteq \texttt{OrderNonDet List}\end{array}}{\Gamma \vdash x.\mathit{equals}(y) : \texttt{Det}\ \mathit{boolean}}\ \text{SET PRECISION}$$

$$\frac{\Gamma \vdash x : \texttt{OrderNonDet Map}\langle\texttt{Det}\ \beta_k, \kappa_v\ \beta_v\rangle, y : \texttt{Det}\ \beta}{\Gamma \vdash x.\mathit{get}(y) : \texttt{Det}\ \beta_v}\ \text{MAP PRECISION}$$

Fig. 11. Special rules for set equality and map lookup.

For example, if `sort()` is called on a receiver of type `OrderNonDet List⟨Det Set⟨Det String⟩⟩`, it will be type refined to `Det List⟨Det Set⟨Det String⟩⟩`. However, it is not permitted to call `sort` on a receiver of type `OrderNonDet List⟨OrderNonDet Set⟨Det String⟩⟩`. The rule is not applicable, because doing so would instantiate invalid types.

As another example, it is only sometimes permitted to call `shuffle` on a receiver of type `Det List⟨Det String⟩`. It is illegal to call `shuffle` on a receiver of *declared* type `Det List⟨Det String⟩`, because doing so would change the receiver's type to be inconsistent with its declaration. However, it is permitted to call `shuffle` on a receiver of type `Det List⟨Det String⟩` that was declared as `OrderNonDet List⟨Det String⟩` but then refined to `Det List⟨Det String⟩`, for example by calling `sort` or via assignment.

2.6 Maps and sets

A map is a collection of key–value pairs. Like other collections, a map or set can be nondeterministic, order-nondeterministic, or deterministic.

It might seem that the notion of a deterministic set or map is nonsensical, since the set and map specifications make no promises about iteration order. However, some subtypes do. We explain why each of these is possible by considering three common implementation strategies.

- A hash table is order-nondeterministic, because hash codes are nondeterministic in general.
- A hash table can be augmented by a linked list recording insertion order, so that iteration order is deterministic.
- A map or set can be backed by a sorted collection, such as a tree. Such a map is either `NonDet` (if `NonDet` keys or values are inserted into it) or `Det`, but never `OrderNonDet`.

Analogously to set membership, lookups are deterministic in order-nondeterministic and deterministic maps; in addition, the iteration order is fixed in a deterministic map.

2.6.1 Improving precision for equality and map lookup. List equality is dependent on iteration order, but set equality is not. Comparing two objects of type `OrderNonDet List⟨Det String⟩` yields a `NonDet` result: depending on the execution, the lists might or might not be in the same order. However, comparing two objects of type `OrderNonDet Set⟨Det String⟩` yields a `Det` result: always the same on every execution.

More specifically, rule SET PRECISION of fig. 11 expresses that the return type of `equals()` is `Det` if both arguments have type `OrderNonDet Set` and neither argument has `@OrderNonDet List` or a subtype within its type argument. Without this rule, the type would be `NonDet` which is sound but imprecise.

Similarly, the rule MAP PRECISION in fig. 11 improves precision when calling `get` on an `OrderNonDet` receiver.

3 DETERMINISM TYPES FOR JAVA

We have implemented our type system for Java, in a tool named Garçon. The implementation consists of 1200 lines of Java code, plus 1800 lines of tests (all line measurements are non-comment, non-blank lines), a manual, etc. Garçon requires Java version 8. It is publicly available at (URL elided).

3.1 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, as in `@Positive int` or `@NonEmpty List<@NonNull String>`.

Garçon supports the type qualifiers `@NonDet`, `@OrderNonDet`, and `@Det` (and others described below). The meaning of `@Det` is with respect to value equality, not reference equality; that is, values on different executions are the same with respect to `.equals()`, not ==.

Garçon treats arrays analogously to collections: the semantics and rules are similar even though the syntax varies. For simplicity this paper uses the term "collection" and the type `Collection` to represent collections and arrays. Collections include any type (including user-defined classes) that implements the `Iterable` or `Iterator` interfaces; this includes all Java collections such as `Lists` and `Sets`.

3.2 Java collection types

A `Map` is deterministic if its `entrySet` is deterministic. In other words, iterating over this `entrySet` produces the same values in the same order across executions. The determinism qualifier on the return type of `entrySet()` is the same as that on the receiver (fig. 13). That is, its type (ignoring type arguments) is $\forall\kappa.\ \kappa$ `Map` $\longrightarrow \kappa$ `Set` .

We have not yet found any implementation of the `Map` interface in which `entrySet()` `keySet()`, and `values()` behave differently with respect to determinism: either all three methods, or none, are 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 HashMap` to its constructor, as in `new LinkedHashMap(myOndHashMap)`.
- `TreeMap` can either be `@Det` or `@NonDet`. An `@OrderNonDet TreeMap` doesn't exist because the entries are always sorted.

Garçon prohibits the creation of a `@Det HashMap` or an `@OrderNonDet TreeMap`. Figure 12 formalizes the type well-formedness rules for Java `Maps` and `Sets`.

$$\frac{\kappa \in \{\texttt{OrderNonDet}, \texttt{NonDet}\}}{\vdash: \kappa\ \texttt{HashMap}}\ \text{VALID HASHMAP}$$

$$\frac{\kappa \in \{\texttt{Det}, \texttt{NonDet}\}}{\vdash: \kappa\ \texttt{TreeMap}}\ \text{VALID TREEMAP}$$

$$\frac{\kappa \in \{\texttt{OrderNonDet}, \texttt{NonDet}\}}{\vdash: \kappa\ \texttt{HashSet}}\ \text{VALID HASHSET}$$

$$\frac{\kappa \in \{\texttt{Det}, \texttt{NonDet}\}}{\vdash: \kappa\ \texttt{TreeSet}}\ \text{VALID TREESET}$$

Fig. 12. Type well-formedness rules for subclasses of `Map` and `Set`.

3.3 Polymorphism

Our type system supports three types of polymorphism: type polymorphism, basetype polymorphism, and qualifier polymorphism. These apply to both classes and methods.

- In the Garçon implementation, type polymorphism is handled by Java's generics mechanism, which Garçon fully supports, including class and method generics, inference, etc. Given the Java declaration
  `<T> T identity(T param) { return param; }`
  the type of `identity` is $\forall\tau.\ \tau \rightarrow \tau$, and the type of `identity(x)` is the same as the type of `x`. A programmer can also write a type qualifier on a type argument, as in `@NonDet List<@NonDet Integer>`.
- Basetype polymorphism is enabled by writing a type qualifier on a use of a type variable, which overrides the type qualifier at the instantiation site. For example, a `choose` operation on sets could be defined in Java as
  ```
  interface ChooseableSet<T> implements Set<T> {
    // return an arbitrary element from the set
    @NonDet T choose();
  }
  ```
- Java does not provide a syntax that can be used for qualifier polymorphism, so Garçon uses a special type qualifier name, `@PolyDet`. (`@PolyDet` stands for "polymorphic determinism qualifier".) A qualifier-polymorphic method `m` with signature $\forall\kappa.\ \kappa$ `int` $\times$ `Det boolean` $\rightarrow$ κ `String` would be written 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. Garçon currently supports qualifier polymorphism on methods but *not* on classes. Therefore, `@PolyDet` may be written in methods (signatures and bodies) but not on fields.

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) { return a+b; }
```

This can be used in more contexts than

```
@Det int plus(@Det int a, @Det int b) { return a+b; }
```

can be, as was shown in section 2.4.1.

Just as a qualifier variable κ is written as `@PolyDet` in Java source code, $\kappa\uparrow$ is written as `@PolyDet("up")`, and $\kappa\downarrow$ is written as `@PolyDet("down")`. An occurrence of a qualifier variable that does not affect the binding of that variable ($use(\kappa)$ in section 2.4.3) is written `@PolyDet("use")`.

```
// Annotated JDK methods
public interface Map<K,V> {
    @PolyDet Set<Map.Entry<K, V>> entrySet(@PolyDet Map<K,V> this);
}
public interface Iterator<E> {
    @PolyDet("up") E next(@PolyDet Iterator<E> this);
}

// Client code
public class MapUtils {
    public static <K extends @PolyDet Object, V extends @PolyDet Object>
                            @Det String toString(@PolyDet Map<K,V> map) {
        ...
        for (Map. @Det Entry<K,V> entry : map.entrySet()) { ... }
    }
}
[ERROR] MapUtils.java:[20,50] [enhancedfor.type.incompatible]
incompatible types in enhanced for loop.
found : @PolyDet Entry<K extends @PolyDet Object,V extends @PolyDet Object>
required: @Det Entry<K extends @PolyDet Object,V extends @PolyDet Object>
```

Fig. 13. Example: Error detected by Garçon in `scribe-java` [Shi et al. 2016].

All of our syntax is legal Java code that can be compiled with any Java 8 or later compiler.

Figure 13 presents some of the JDK methods that we have annotated with our determinism types and give examples of client code that would produce errors at compile time.

3.4 Defaulting

Garçon 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.

Formal parameter and return types default to `@PolyDet`. That is, a programmer-written method

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

is treated as if the programmer had written

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

and its function type is

$\forall\kappa.\ \kappa\ \texttt{int} \times \kappa\ \texttt{int} \longrightarrow \kappa\ \texttt{int}$

This choice type-checks if the method body does not make calls to any interfaces that require `@Det` arguments or produce `@NonDet` results. Otherwise, the programmer must write explicit type qualifiers in the method signature.

If a formal parameter or return type is a collection, its element type defaults to `@PolyDet`. No other choice would result in a valid type after instantiation.

As an exception to the above rules about return types, if a method's formal parameters (including the receiver) are all `@Det` — that is, there are no unannotated or `@PolyDet` formal parameters — then an unannotated return type defaults to `@Det`. This is particularly useful for methods that take no formal parameters. A type like $\forall\kappa.\ () \rightarrow \kappa$ `int` does not make sense, because there is no basis on which to choose an instantiation for the type argument κ. Treating the type as $() \longrightarrow$ `Det int` permits just as many uses.

Types are inferred for unannotated local variables; see section 3.5.

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.

3.5 Type refinement via dataflow analysis

Our type system is flow-sensitive [Adams et al. 2011; Hunt and Sands 2006; Sui and Xue 2016]. That is, an expression may have a different type on every line of the program, based on assignments and side effects. The current type must always be consistent with the declared type, if any. Consider the example below:

```
@NonDet int x = ...;
                        // The type of x is @NonDet int
x = 42;
                        // The type of x is @Det int
@Det int y = x;
x = random();
                        // The type of x is @NonDet int
y = random();           // Error: y is declared as @Det int
```

Arbitrary expressions can be given more specific types, including fields and pure method calls. A type refinement is lost whenever a side effect might affect the value. For example, type refinements to all fields are lost whenever a non-pure method is called.

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

Although Garçon performs local type inference within method bodies, it does not perform whole-program type inference. This makes separate compilation possible. It forces programmers to write specifications (type qualifiers) on methods, which is good style and valuable documentation.

3.6 The environment

The return type of `System.getenv`, which reads an operating system environment variable, is always `@NonDet`.

The return type of `System.getProperty` is usually `@NonDet`. However, the result is `@Det` if the argument is `"line.separator"`; although it varies by operating system, many programs including `diff` and editors hide those differences from the user. The result is also deterministic for `"path.separator"` and `"file.separator"`, because these lead to the same behavior given corresponding environments. A user of Garçon can specify Java properties that must be passed on the `java` command line and thus act like inputs to the program; Garçon treats these as deterministic.

The inputs to a program are also treated as deterministic. That is, the type of the formal parameter to `main` is `@Det String @Det []`, a deterministic array of deterministic strings.

4 CASE STUDY

To evaluate the usability of Garçon, we applied it to the Randoop test generator [Randoop]. We also wrote specifications for libraries Randoop uses, such as the JDK, JUnit, plume-util, and others. Like the Garçon implementation, all the case study materials are publicly available at (URL elided) for reproducibility.

4.1 Subject program

We chose Randoop because it is frequently used in software engineering experiments, it is actively maintained, and its developers have struggled with nondeterminism [Randoop issue tracker 2019; Randoop mailing lists 2019].

Randoop is intended to be deterministic, when invoked on a deterministic program [Randoop Manual 2019].[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.

The developers took extensive action to detect and mitigate the effect of 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 don't 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 nondeterministing output. Each of these efforts was helpful, but they were not enough to solve the problem.

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 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 a version of Randoop from late February 2019 (commit 0a8e63fb), over 6 months after the most recent determinism bug fix.

4.2 Methodology

In brief, we wrote type qualifiers in the Randoop source code to express its determinism specification, then ran Garçon. 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.

We annotated the core of Randoop (the `src/main/java` directory), which contains 24K non-comment, non-blank lines of code. We did not annotate Randoop's test suite.

[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.

We annotated one package at a time, starting with the most central packages that are most depended upon. Within a package, we followed a similar strategy, annotating supertypes first. If the determinism of classes and methods had been documented, then our annotation effort would have been easy, just converting English into type qualifiers. Unfortunately, this was not the case: we had to reverse-engineer each specification, largely from the methods it calls. If a method *m* calls a method that we had previously annotated as returning a nondeterministic result, we annotated *m* as returning a nondeterministic result. Sometimes after we annotated a subclass, we had to go back and change annotations on the supertype because the subclass implementation was inconsistent with the specification we had guessed for the supertype. After resolving most type-checking warnings by adding or changing annotations, we manually examined each remaining one to determine whether it was a bug or a false positive warning. When the number of `@Det` annotations in a file was overwhelming, we changed the default qualifier for that class to `@Det`. (Users can control defaulting on a file-by-file and method-by-method basis.) 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.

`javac` takes 18 seconds to compile all files of Randoop. While also running Garçon as a compiler plugin, `javac` takes 32 seconds to verify determinism and compile all files of Randoop. These numbers are the median of 5 trials on an 8-core Intel i7-3770 CPU running at 3.40GHz with 32GB of memory.

4.3 Results

Garçon found 5 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

HashSet bug One use of `HashSet` could cause a problem if a type variable's lower or upper bound in the code Randoop is run on has a type parameter that the type variable itself does not have. 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). Garçon 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, shown in fig. 14). Garçon verified that all other uses of system and Java properties did not lead to nondeterministic behavior.

Moderate issues

HashMap output bug While printing diagnostic output, Randoop iterated over a `HashMap` in arbitrary order, making the output difficult to compare across different executions, or even to find information in. The class already implemented `Comparable`, so the developers changed `methodWeights.keySet()` to `new TreeSet<>(methodWeights.keySet())` in a `for` loop (commit f212cc7e).

Minor issues

Hash code output bug Diagnostic output printed a hash code for brevity. The developers changed it to have deterministic output (commit 661a4970). Although the Randoop developers classified this issue as minor, it is similar to ones they fixed during their two-week coding sprint.

In `Minimize.java`:

```
151:-  private static final String SYSTEM_CLASS_PATH = System.getProperty("java.class.path");

913:-    String command = "javac -classpath " + SYSTEM_CLASS_PATH + PATH_SEPARATOR + ".";
913:+    String command = "javac -classpath .";
914:     if (classpath != null) {
915:       // Add specified classpath to command.
916:       command += PATH_SEPARATOR + classpath;
917:     }

948:-    String classpath = SYSTEM_CLASS_PATH + PATH_SEPARATOR + dirPath;
948:+    String classpath = dirPath;
949:     if (userClassPath != null) {
950:       classpath += PATH_SEPARATOR + userClassPath;
951:     }
```

In `MinimizerTests.java`:

```
55:-    String classPath = "";
55:+    String classPath = JUNIT_JAR;
56:     if (dependencies != null) {
57:       for (String s : dependencies) {
58:         Path file = Paths.get(s);
59:         classPath += (pathSeparator + file.toAbsolutePath().toString());
60:       }
61:     }
```

Fig. 14. Fixes made by the Randoop developers in response to our bug report about use of environment variables. Lines starting with "-" were removed and those starting with "+" were added. Garçon verified all other uses of system and Java properties.

Timestamp output bug Diagnostic output printed a timestamp. The developers fixed it by making it obey an option about whether to print timestamps (commit a460df97). The developers may not have noticed this because their log-postprocessing tools removed timestamps from the log when doing comparisons.

We reported another suspicious case of order-nondeterminism, in the `SpecificationCollection-.findOverridden` method. The Randoop developers explained it was acceptable, after tracing the flow through the program. (One said, "This looks OK to me … it took me a little while to decide that.") The fact depended on subtle, undocumented invariants about Randoop that we had not been able to reverse-engineer on our own.

4.4 False positive warnings

Garçon issued 37 false positive warnings, or 1 for every 650 lines of code.

The most common reasons (responsible for 67% of false positive warnings) were:

- (22%) When every instance of a class is `@Det`, `@PolyDet` is equivalent to `@Det`, but Garçon does not yet recognize this fact.
- (16%) For a type that cannot be `@OrderNonDet`, `@PolyDet("up")` is equivalent to `@PolyDet`.
- (14%) Java does not have syntax for specifying the receiver type in anonymous class methods.
- (11%) A variable is declared as `Object` which has nondeterministic `toString`, but at run time the subtype of `Object` stored always has deterministic `toString`. For example, the run-time value is always an `Integer` or a `String`.

- (5%) Iteration over an `@OrderNonDet` collection in order to make another `@OrderNonDet` collection, but the extracted elements have type `@NonDet`. For example, the following code is safe, but Garçon issues a warning. Variable `elt` has type `@NonDet`, so the call to `add` does not type check.

```
void m(@OrderNonDet List<@Det String> input) {
  @OrderNonDet List<@Det String> output = new List<>();
  for (String elt : input) {  // elt has type @NonDet String
    output.add(elt);          // error: illegal argument
  }
}
```

The first two issues can be resolved by improvements to the Garçon implementation. The third and fourth issues are limitations of the Java language, but a local refactoring in the Randoop codebase can make the code type check. Refactoring can also eliminate the last warning, for example by using the higher-order `map()` function instead of a loop; alternately, Garçon could be enhanced to pattern-match some loops.

4.5 Annotation effort

In Randoop's 24K lines of code, we wrote 98 `@NonDet`, 64 `@OrderNonDet`, 1083 `@Det`, and 103 `@PolyDet` annotations.

The number of `@Det` annotations — one per 22 lines of code — is much higher than we would prefer. Nonetheless, it compares favorably to the extensive effort by the Randoop developers (section 4.1), especially given that Garçon found issues that the developers did not. As another point of comparison, there are fewer type qualifiers than Java generic type arguments.

The large number of `@Det` annotations has two main reasons.

One is that we annotated some core components of Randoop to require all Boolean expressions used in conditionals to be deterministic. (This is an option in the Garçon implementation, which increases soundness; see section 7.) This forced their clients — transitively, much of the Randoop codebase — to be annotated as deterministic. These annotations are not incorrect, but they are stricter than necessary because some method types could otherwise be qualifier-polymorphic. This would make them usable in more contexts. It would also reduce the number of necessary annotations, since qualifier-polymorphism (the `@PolyDet` annotation) is the default for formal parameter and return types.

The second reason is a limitation in the Garçon implementation, which is inherited from the Checker Framework [Checker Framework 2019] upon which Garçon is built. The Checker Framework does not support qualifier polymorphism over classes. This means that fields must be annotated as deterministic. `@Det` annotations must be added to methods and constructors that modify these fields, as well as any methods that call these. Our annotations in Randoop use qualifier polymorphism extensively on methods but cannot use it on classes, and this leads to many more `@Det` annotations than would be needed otherwise.

In future work, we plan to implement qualifier polymorphism for classes, then re-do the case study. We estimate this will reduce the number of `@Det` annotations needed by about an order of magnitude.

5 COMPARISON TO NONDEX

The state of the art in flaky test detection is NonDex [Shi et al. 2016]. Section 8 explains how NonDex works. This section compares the errors reported by NonDex and Garçon.

`Class`	: `getDeclaredFields`, `getDeclaredMethods`, `getFields`
`DateFormatSymbols`	: `getZoneStrings`
`HashMap`	: `entrySet`, `keySet`, `values`

Fig. 15. Souces of flakiness known to NonDex [Shi et al. 2016].

5.1 Case study with NonDex

We ran NonDex on the same commit of Randoop as we used for the case study of section 4. This version contains all 5 nondeterminism bugs that Garçon found.

We modified Randoop by deleting tests that were skipped by its buildfile, because the NonDex Gradle plugin does not respect those settings. After that, NonDex ran without problems on Randoop. NonDex reported no flaky tests.

We investigated the tests covering code that Garçon reported as nondeterministic. **HashSet bug** and **Classpath bug** have corresponding test cases whereas the remainder of the bugs do not. As NonDex is a *flaky test* detection tool, detecting bugs that don't have corresponding tests is outside of its scope. The reason for nondeterminism in **Classpath bug** was a call to the `System.getProperty()` method which is not modeled by NonDex.

5.2 Garçon on NonDex benchmarks

Section 5.1 shows that Garçon finds errors that NonDex does not. This section determines whether NonDex finds errors that Garçon does not. Its authors ran NonDex on 195 open-source projects, and NonDex found flaky tests in 21 of them [Shi et al. 2016]. The authors also reported the sources of flakiness after manually inspecting these tests. The flakiness that NonDex found was due to 7 methods in 3 classes (fig. 15).

We tried to run Garçon on all these benchmarks, at the commit given in the NonDex paper. Some of the projects had moved, or did not build because their dependencies had moved or were no longer available. We repaired all these issues and were able to compile all but two projects, handlebars and oryx. Some of the remaining projects did not pass their tests, but that did not hinder us since Garçon works at compile time.

For three of the projects, we could not find the flakiness reported in the NonDex paper. The reported root cause of flakiness in `easy-batch` and `vraptor` was a call to `Class.getDeclaredFields`. We couldn't find a call to this method in any of the source files of these two repositories. Similarly, the flakiness in `visualee` was attributed to an invocation of `Class.getDeclaredMethods` which we didn't find in the source code.

This left 16 projects. We ran Garçon on the part of the project that the NonDex authors determined as flaky. In every case, Garçon issued a warning on the nondeterministic code. In other words, Garçon's recall was 100%.

All these benchmarks and instructions to reproduce our results can be found at (URL elided).

Figure 16 shows samples of nondeterministic code from these benchmarks. We annotated the benchmarks based on assumptions made downstream of the shown code.

In reflectasm (fig. 16a), we changed the type of field `declaredFields` from `Field[]` to `@Det Field @Det []`. That type means a deterministic array of deterministic `Field`, analogously to `@Det List<@Det Field>`. Then, Garçon issued a warning at the assignment, because `getDeclaredMethods` returns `@Det Field @OrderNonDet []`, which is an order-nondeterministic array of deterministic `Fields`.

The ActionGenerator code (fig. 16b) is similar. Other code assumes that `entry` is deterministic, but annotating it as `@Det` leads to a warning from Garçon that iterating over `fields.entrySet()` (which is itself `@OrderNonDet`) yields `@NonDet Entry` values.

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

(a) Nondeterministic source code from reflectasm.

```
protected SimpleDataEvent createNextEvent() {
    for (Entry<String, FieldType> entry : fields.entrySet()) {
        ...
    }
    ...
}
```

(b) Nondeterministic source code from ActionGenerator.

Fig. 16. Errors detected by Garçon in NonDex benchmarks.

Nondex found 14 flaky tests in Apache Commons Lang due to calls to `Class.getDeclaredFields`. There are only 5 invocations of `Class.getDeclaredFields`, so annotating 5 lines of source code would have been sufficient to identify all that nondeterminism. Having said that, we admit that there could be significant programmer effort involved in annotating the whole program. On the other hand, 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." Garçon reports source locations which makes it easier for the programmer to fix issues, and the annotation effort serves as valuable documentation and prevents regressions.

Garçon is capable of detecting nondetermism in test cases, as some of the NonDex examples are. We wrote specifications for the JUnit library (in addition to the JDK and some other libraries). The specifications are the `@Det` type qualifier on the formal parameters to the `assert*` methods. Garçon reports any `junit` test that passes nondeterministic arguments to `assert*`, even if the test case is not executed.

6 COMPARISON TO DEFLAKER

Deflaker [Bell et al. 2018], like NonDex, reports tests that could be flaky. We were unable to run DeFlaker on Randoop, because Deflaker works with projects built with Maven, but Randoop uses Gradle for its build system.

6.1 Garçon 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 [Bell et al. 2018]. We ran Garçon on the part of each of these codebases where the reported bug was fixed, as in section 5.2.

Four of the seven flaky tests (two in `achilles`, one each in `jackrabbit-oak` and `togglz`) were caused by a race condition, which Garçon cannot detect. (This is a strength of DeFlaker over

Garçon.) 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 bug in `togglz` was caused by a copy method that iterated over an `OrderNonDet Set` and expected it to be deterministic. Garçon correctly flagged an error in the loop that iterated over the `Set`. The flakiness in `nutz` was a result of printing a response received over http. Since network operations are nondeterministic, we annotated the method in `nutz` that returns this response as `NonDet` which led Garçon to report an error at the print statement. Garçon 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.

7 THREATS TO VALIDITY

Our type system and implementation suffer some unsoundness. The most important is that they do not capture all sources of nondeterminism. One important such source is concurrency. In addition, our annotations of the JDK (the Java standard library) are incomplete. Some of the unannotated parts might introduce nondeterminism without the verification tool being aware of it.

The type system presented in this paper does not account for the fact that a nondeterministic conditional expression (in an `if` or `switch` statement or a `for` or `while` loop, for example) may behave differently from execution to execution. These "implicit flows" are a well-known challenge for taint analysis. Suppose that an expression used in a conditional is tainted. A standard approach [Arzt et al. 2014; Kang et al. 2011] for a dynamic taint analysis is to taint any value that may be assigned in either branch of the conditional statement. This tends to rapidly lead to most of the program state being tainted. In a static analysis, every possibly-assigned lvalue would be considered possibly tainted; again, this tends to snowball through the program.

Our implementation, Garçon, can enforce that conditional expressions are deterministic, but we found this to be too restrictive: it causes Garçon to issue large numbers of false positive warnings, or it forces types to be `Det` unnecessarily. We disabled this check in most of our case study.

It our case study, we also disabled a check that required all caught exceptions to have `NonDet` type. This type is a safe approximation to an exception that can be thrown from arbitrary code, including libraries. Enabling this check led to many unhelpful false positive warnings, which are only relevant if unverified libraries throw nondeterministic exceptions.

Garçon only examines the code it is run on. Unchecked libraries and native code with incorrect specifications might introduce nondeterminism even if Garçon issues no warnings.

We have performed only one case study. This case study found important previously-unknown errors, but its results might not carry over to other programs or to programs written in other programming lanugages. We mitigated this problem by showing that Garçon finds many of the sources of nondeterminism identified by other tools.

8 RELATED WORK

The state of the art for detecting nondeterministic tests is NonDex [Shi et al. 2016]. It is a highly effective tool. 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. Garçon, in contrast, reports the cause of nondeterminism at compile time requiring little to no debugging effort. However, Garçon 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 928 methods across 41 classes in the JDK, in addition to other libraries.

DeFlaker [Bell et al. 2018] 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, it is flagged as being flaky. This approach doesn't 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 Garçon cannot.

Nondeterminism in tests is of interest to both researchers and software developers alike [Fowler 2011; Sudarshan 2012]. Luo et al. [Luo et al. 2014] studied 51 open source projects and analyzed the most common root causes of test flakiness. The findings suggest that most of the flakiness in tests is caused by the use of async await, concurrency, or due to test order dependency. The authors of [Plotkin and Abadi 1993] perform an empirical study on open source benchmarks, the results of which concur with those of [Luo et al. 2014]. Additionally, [Plotkin and Abadi 1993] studies the effects of removing these test smells. The results suggest that refactoring the test code smells fixed more than half of the identified flaky tests. Our approach is complementary to these techniques and aims to prevent nondeterminism from causing harmful effects.

The problem of inconsistencies between specifications and implementations is quite prevalent. Other researchers have explored ways of reducing the gap between implicit specifications in APIs and the assumptions made by their clients in a variety of problem domains. The findings in [Jin et al. 2012] suggest that incorrect understanding of performance specifications of APIs is a very common root cause for performance bugs. [Wang et al. 2013] analyzes the APIs related to security such as authorization and authentication. These APIs which come as a part of the SDKs provided by online providers are used by application developers to build secure apps. The paper discovers that the security guarantees provided by these SDKs are violated due to the implicit assumptions made by the APIs that the users are unaware of. In [Xiao et al. 2014], the authors study the effect of nondeterminism in MapReduce programs with a specific focus on nondeterminism caused by non-commutative reducers. While they found bugs that violated correctness due to this bug pattern, the authors reported that several of these were harmless as they relied on an implicit assumption on data which ensured correctness.

Several techniques have been proposed to test whether a deterministic implementation conforms to its nondeterministic finite state machine [Hierons and Harman 2004; Petrenko et al. 1994, 1996; Savor and Seviora 1997]. [Cook and Koskinen 2013] presents an approach that can automatically verify properties in branching time temporal logic systems that are inherently nondeterministic. Bocchino et al. [Bocchino et al. 2009, 2011] present a type-and-effect system that provides compile-time determinism guarantees for parallel programs, with a focus on barrier removal, reasoning about interference and thread interleavings. We believe our work to be the first verification approach aimed at ensuring determinism in sequential programs.

Failing tests that are unrelated to code changes can be expensive both in terms of monetary costs and developer effort. [Herzig and Nagappan 2015] proposes techniques to classify tests as false alarms if they are known to be caused by testing infrastructure or other environment issues. [Huo and Clause 2014] presents an approach that detects brittle assertions in tests by performing a taint analysis on inputs classified as controlled and uncontrolled. [Zhang et al. 2014] investigates the effects of the test independence assumption on other techniques such as test prioritization, selection, etc. Other approaches [Bell et al. 2015; Gyori et al. 2015] analyze test dependencies and either prevent them or use this information for other optimizations. The approaches in [Hao et al.

2013; Vahabzadeh et al. 2015] focus on differentiating bugs due to tests from those caused by source code. While we haven't fully accounted for all nondeterministic sources due to the environment, our technique is easily extensible by adding annotations to the corresponding classes.

9 CONCLUSION

We designed a type system that helps programmers express determinism specifications in sequential programs. We implemented our type system in Java, applied it to real world software, and found it to be effective in finding previously unknown determinism bugs. Our tool Garçon found errors that the Randoop developers had missed, despite spending extensive effort on the problem of nondeterminism. We compared our tool with existing state of the art flaky test detectors and found that Garçon found all the bugs arising due to nondeterminism in sequential programs that were used for evaluation in NonDex [Shi et al. 2016] and DeFlaker [Bell et al. 2018]. To the best of our knowledge, ours is the first compile-time verification approach addressing the problem of nondeterminism in sequential programs.

REFERENCES

M. Abadi, B. Pierce, and G. Plotkin. 1989. Faithful Ideal Models for Recursive Polymorphic Types. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*. IEEE Press, Piscataway, NJ, USA, 216–225. http://dl.acm.org/citation.cfm?id=77350.77373

Michael D. Adams, Andrew W. Keep, Jan Midtgaard, Matthew Might, Arun Chauhan, and R. Kent Dybvig. 2011. Flow-sensitive Type Recovery in Linear-log Time. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '11)*. ACM, New York, NY, USA, 483–498. https://doi.org/10.1145/2048066.2048105

Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 259–269. https://doi.org/10.1145/2594291.2594299

Jonathan Bell, Gail Kaiser, Eric Melski, and Mohan Dattatreya. 2015. 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, 770–781.

Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. DeFlaker: Automatically Detecting Flaky Tests. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 433–444. https://doi.org/10.1145/3180155.3180164

Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. 2009. 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 (OOPSLA '09)*. ACM, New York, NY, USA, 97–116. https://doi.org/10.1145/1640089.1640097

Robert L. Bocchino, Jr., Stephen Heumann, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Adam Welc, and Tatiana Shpeisman. 2011. Safe Nondeterminism in a Deterministic-by-default Parallel Language. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 535–548. https://doi.org/10.1145/1926385.1926447

KA Briski, Poonam Chitale, Valerie Hamilton, Allan Pratt, B Starr, J Veroulis, and B Villard. 2008. Minimizing code defects to improve software quality and lower development costs. *Development Solutions. IBM. Crawford, B., Soto, R., de la Barra, CL* (2008).

Checker Framework 2019. https://checkerframework.org/.

Byron Cook and Eric Koskinen. 2013. Reasoning About Nondeterminism in Programs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 219–230. https://doi.org/10.1145/2491956.2491969

Martin Fowler. 2011. https://martinfowler.com/articles/nonDeterminism.html.

Alex Gyori, August Shi, Farah Hariri, and Darko Marinov. 2015. Reliable Testing: Detecting State-polluting Tests to Prevent Test Dependency. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, New York, NY, USA, 223–233. https://doi.org/10.1145/2771783.2771793

Dan Hao, Tian Lan, Hongyu Zhang, Chao Guo, and Lu Zhang. 2013. Is This a Bug or an Obsolete Test?. In *ECOOP 2013 – Object-Oriented Programming*, Giuseppe Castagna (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 602–628.

Kim Herzig and Nachiappan Nagappan. 2015. Empirically Detecting False Test Alarms Using Association Rules. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 39–48. http://dl.acm.org/citation.cfm?id=2819009.2819018

R. M. Hierons and M. Harman. 2004. Testing Conformance of a Deterministic Implementation Against a Non-deterministic Stream X-machine. *Theor. Comput. Sci.* 323, 1-3 (Sept. 2004), 191–233. https://doi.org/10.1016/j.tcs.2004.04.002

Sebastian Hunt and David Sands. 2006. On Flow-sensitive Security Types. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06)*. ACM, New York, NY, USA, 79–90. https://doi.org/10.1145/1111037.1111045

Chen Huo and James Clause. 2014. 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 (FSE 2014)*. ACM, New York, NY, USA, 621–631. https://doi.org/10.1145/2635868.2635917

Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and Detecting Real-world Performance Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 77–88. https://doi.org/10.1145/2254064.2254075

Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Xiaodong Song. 2011. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. In *NDSS*.

Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *FSE 2014: Proceedings of the ACM SIGSOFT 22nd Symposium on the Foundations of Software Engineering*. Hong Kong, 643–653.

Alexandre Petrenko, Nina Yevtushenko, Alexandre Lebedev, and Anindya Das. 1994. Nondeterministic State Machines in Protocol Conformance Testing. In *Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test Systems VI*. North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands, 363–378. http://dl.acm.org/citation.cfm?id=648128.761244

A. Petrenko, N. Yevtushenko, and G. v. Bochmann. 1996. 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*, Bernd Baumgarten, Heinz-Jürgen Burkhardt, and Alfred Giessler (Eds.). Springer US, Boston, MA, 125–140. https://doi.org/10.1007/978-0-387-35062-2_10

Gordon D. Plotkin and Martín Abadi. 1993. A Logic for Parametric Polymorphism. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications (TLCA '93)*. Springer-Verlag, London, UK, UK, 361–375. http://dl.acm.org/citation.cfm?id=645891.671433

Md Tajmilur Rahman and Peter C. Rigby. 2018. The Impact of Failing, Flaky, and High Failure Tests on the Number of Crash Reports Associated with Firefox Builds. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. ACM, New York, NY, USA, 857–862. https://doi.org/10.1145/3236024.3275529

Randoop 2019. https://github.com/randoop/randoop.

Randoop issue tracker 2010–2019. https://github.com/randoop/randoop/issues.

Randoop mailing lists 2010–2019. https://groups.google.com/forum/#!forum/randoop-developers and https://groups.google.com/forum/#!forum/randoop-discuss.

Randoop Manual 2019. https://randoop.github.io/randoop/manual/index.html. Version 4.1.1.

T. Savor and R. E. Seviora. 1997. Supervisors for testing non-deterministically specified systems. In *Proceedings International Test Conference 1997*. 948–953. https://doi.org/10.1109/TEST.1997.639710

Munil Shah. 2017. https://docs.microsoft.com/en-us/azure/devops/learn/devops-at-microsoft/eliminating-flaky-tests.

August Shi, Alex Gyori, Owolabi Legunsen, and Darko Marinov. 2016. Detecting Assumptions on Deterministic Implementations of Non-deterministic Specifications. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 80–90. https://doi.org/10.1109/ICST.2016.40

Pavan Sudarshan. 2012. https://www.thoughtworks.com/insights/blog/no-more-flaky-tests-go-team.

Yulei Sui and Jingling Xue. 2016. On-demand Strong Update Analysis via Value-flow Refinement. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 460–473. https://doi.org/10.1145/2950290.2950296

Ran Tene. 2018. https://blogs.dropbox.com/tech/2018/05/how-were-winning-the-battle-against-flaky-tests/.

A. Vahabzadeh, A. M. Fard, and A. Mesbah. 2015. An empirical study of bugs in test code. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 101–110. https://doi.org/10.1109/ICSM.2015.7332456

Rui Wang, Yuchen Zhou, Shuo Chen, Shaz Qadeer, David Evans, and Yuri Gurevich. 2013. Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. USENIX, Washington, D.C., 399–314. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/wang_rui

Tian Xiao, Jiaxing Zhang, Hucheng Zhou, Zhenyu Guo, Sean McDirmid, Wei Lin, Wenguang Chen, and Lidong Zhou. 2014. 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 (ICSE Companion 2014)*. ACM, New York, NY, USA, 44–53. https://doi.org/10.1145/2591062.2591177

Kai Yu, Mengxiang Lin, Jin Chen, and Xiangyu Zhang. 2012. Towards automated debugging in software evolution: Evaluating delta debugging on real regression bugs from the developers' perspectives. *J. Syst. Softw.* 85, 10 (October 2012), 2305–2317.

Andreas Zeller. 1999. 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, 253–267.

Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muşlu, Wing Lam, Michael D. Ernst, and David Notkin. 2014. Empirically revisiting the test independence assumption. In *ISSTA 2014, Proceedings of the 2014 International Symposium on Software Testing and Analysis*. San Jose, CA, USA, 385–396.