Does It Always Do That? Practical Lightweight Nondeterminism Detection and Debugging

Alex Groce
School of Informatics, Computing & Cyber Systems
Northern Arizona University
Email: agroce@gmail.com

Josie Holmes
School of Informatics, Computing & Cyber Systems
Northern Arizona University
Email: josie.holmes@nau.edu

***Abstract*—A critically important, but surprisingly neglected in the software testing literature, aspect of system reliability is system predictability. Many software systems are implemented using mechanisms (unsafe languages, mulitple threads, complex caching, stochastic algorithms, environmental dependencies) that can introduce behavioral nondeterminism. Users of software systems, especially other software systems using library calls in a single-threaded context, often expect that systems will behave deterministically in the sense that they have predictable results from the same sequential series of operations. Equally importantly, even when it does not impact correctness or user experience, nondeterminism must be understood and managed for effective (especially automated) testing. Nondeterministic behavior that is not either controlled or accounted for can result in flaky tests as well as causing problems for test reduction, differential testing, and automated regression test generation. We show that lightweight techniques, requiring (almost) no effort on the part of a developer, can be used to extend an existing test generation system to allow checking for problematic nondeterminism. We introduce a set of lightweight nondeterinism properties, inspired by real faults, and tailor these notions to the practical, automatic, checking of Python library code.**

I. Introduction

For ourselves, we might prefer to think (and act as if) we had free will; however, we generally prefer our software systems to be as constrained in their actions as possible: in other words, we wish them to be largely deterministic, from our perspective.

WHY DETERMINISM: Bugs, flaky tests, pain debugging.

A. What is Determinism?

A system is deterministic if, given its complete state at a point in time, it is possible, in principle, to predict its future behavior perfectly. We say "in principle" because in the real world, prediction may be possible but hopelessly impractical. We write complex software systems in many cases because we cannot predict their behavior (if we could perform the calculations in advance, ourselves, we would just do so). As a consequence, in software, rather than defining determinism in terms of prediction, we usually therefore simply say that a system is deterministic if, given the same state and inputs, it always produces the same outputs[1]. The change of definition, we will see, is central to our approach, and comes with a number of limitations.

[1]Of course, this shift of definition is often adopted in arguments for physical or human behavioral determinism, due to the even greater difficulties of prediction in those arenas.

Technically, many "nondeterministic" systems are not nondeterministic in a strong sense, at all. Given the complete state of the system (which includes the entire state of the underlying hardware, the operating system, storage devices, etc.), ignoring quantum effects, and treating outside interventions such as network traffic, human activity at an input device, etc. properly as inputs, most software *is* completely predictable. What we actually mean, usually, is that, given a certain limited abstraction of state and of inputs, observable behavior is repeatable. This abstraction, for most software, is not expected to include many elements outside of the software system itself.

Unexpected nondeterminism usually arises when there is an element of state or input that *is* critical to the produced behavior, but the programmer has not anticipated. E.g., it is beleived that the behavior of a thread scheduler will not matter, but a race condition in the code means that it does matter, after all.

B. The High Cost of Unexpected Nondeterminism

Unexpected nondeterminism is, unfortunately, usually only discovered in a context that makes it very hard to debug. The most common such contexts are occasional rare failures of a system in deployment, and flaky tests.

II. Practical Lightweight Nondeterminism Detection and Debugging

A. Horizontal Determinism

1) Visible Value Determinism:

2) Final State Determinism:

3) Process-Level Determinism:

B. Vertical Determinism

Apple bugs.

1) Failure Nondeterminism:

C. Nondeterminism and Delta-Debugging

In *monotonic* cases, removing a part of a test cannot increase the probability of a predicate holding. This is fairly common. In these cases, when use a reduction algorithm to reduce t to r, $P(p(r) \leq P(p(t))$. In *non-monotonic* cases, however, there is no such upper bound. Removing a step in R

The probabilistic predicate of interest for determinism is always of the form: **the test will exhibit at least two different**

behaviors in S runs, with probability P. This predicate may be monotonic or non-monotonic, depending on the causes of the nondeterminism detected.

III. Implementation

We implemented our approach in the TSTL [2] system.

IV. Experimental Evaluation

A. Case Study: Redis-Py

B. Case Study: pyfakefs and OS X

C. Case Study: Datarray Inference Algorithms

https://github.com/BIDS/datarray

V. Related Work

Shi et. al [3] examined what might be seen as a related, though in a sense the opposite, problem: they detect code that assumes a deterministic implementation of a non-deterministic specification. E.g., they detect instances when iteration through items of a set is explicitly not guaranteed to provide any particular order of items, but code depends on the order produced by a given implementation.

Other efforts [1] have aimed to avoid nondeterminism in parallel implementations, by design, indicating the importance of avoiding nondeterministic behavior, even at the expense of adopting novel programming models, where the goal is not simply (as in, say, Rust) to avoid classic concurrency errors, but to enforce genuinely deterministic behavior.

VI. Conclusions and Future Work

References

[1] R. L. Bocchino, Jr., V. S. Adve, S. V. Adve, and M. Snir. Parallel programming must be deterministic by default. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, HotPar'09, pages 4–4, Berkeley, CA, USA, 2009. USENIX Association.

[2] A. Groce and J. Pinto. A little language for testing. In *NASA Formal Methods Symposium*, pages 204–218, 2015.

[3] A. Shi, A. Gyori, O. Legunsen, and D. Marinov. Detecting assumptions on deterministic implementations of non-deterministic specifications. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 80–90, April 2016.

```
@import inference_algs
@import datarray

<@
def flatten_and_sort(v):
    return (sorted(map(flatten_and_sort,v),key=repr) if type(v) in [list,tuple] else
                (flatten_and_sort(list(v.items())) if type(v) == dict else v))
def psplit(P):
    return ([P,1.0-P])
@>

pool: <P> 3
pool: <cpts> 3
pool: <evidence> 3 OPAQUE
pool: <ename> 3
pool: <event> 3

<P> := 0.01 * <[0..100]>

<ename> := "E" + str(<[1..5]>)

{Exception} <event> := [datarray.DataArray(psplit(<P>), axes = [<ename>])]
{Exception} <ename,1>!=<ename,2> -> <event> := [datarray.DataArray([[psplit(<P>)],psplit(<P>)],[<ename>,<ename>])]

<cpts> := []
~<cpts>.append(<event>[0])

<evidence> := {}
~<evidence>.update([(<ename>,0)])

{Exception} print(flatten_and_sort(inference_algs.calc_marginals_simple(<cpts>,<evidence>)))
{Exception} print(flatten_and_sort(inference_algs.calc_marginals_sumproduct(<cpts>,<evidence>)))
{Exception} print(flatten_and_sort(inference_algs.calc_marginals_jtree(<cpts>,<evidence>)))
```

Fig. 1: Complete TSTL harness for finding the hash-order bug in the datarray inference algorithms.