Finding Concurrency Bugs Using Graph-Based Anomaly Detection in Big Code

Andrew Habib

Department of Computer Science
TU Darmstadt, Germany
andrew.a.habib@gmail.com

Abstract

Concurrency bugs are very difficult and subtle to discover, reproduce, and fix. Many techniques have been devised by the academic as well as the industry communities to find these bugs. However, most of the effective techniques tend to focus on a subset of the various concurrency bugs types. We propose a new generic concurrency bug detection technique that leverages "Big Code": millions of lines of code freely available on code repositories. Our approach tries to learn the properties of what constitutes a good and a bad synchronization pattern from hundreds of concurrent software using graph-based anomaly detection.

Categories and Subject Descriptors D.2.5 [*Software and its engineering*]: Software testing and debugging

General Terms Concurrency, Anomaly, Big Code

Keywords Concurrency Bugs, Synchronization

1. Introduction & Related Work

Writing correct and efficient concurrent software is notoriously difficult. Most programmers tend to think sequentially and reasoning about concurrent code is not straightforward. Due to the non-deterministic nature of concurrency bugs they do not manifest easily, are not easily reproducible, and fixing them is a difficult task.

There are different classes of concurrency bugs. Data races are among the most studied types of concurrency bugs. They occur when two threads access a shared memory location without being properly synchronized (for example, protected by locks) and one of the accesses is a write. Another major class of concurrency bugs are deadlocks which happen when two (or more) threads circularly wait on each other to release acquired resources. Other types include atomicity violations and order violations. Atomicity violations occur when a thread executing a particular code block which is intended to be executed as one unit is interrupted by another thread; and order violations happen when the expected order of specific memory accesses is violated.

Most of the work on concurrency bug detection focuses on data races and deadlocks. Slightly less studied bugs include atomicity and order violations. It is noticeable that existing techniques focus on finding particular type(s) or a subset of concurrency bugs. Moreover, some approaches such as the work by Hammer et. al. [4] makes assumptions about correct code. Therefore, more research is still needed to produce generic tools capable of finding different types of concurrency bugs. Another observation made by Lu et. al [1] is that there is no single general fixing strategy that could be applied to fix synchronization problems. For instance, only a fraction of non-deadlock concurrency bugs are mitigated by adding or changing locks and their acquisition and release policy. This motivates more research in the direction of fixing of concurrency bugs.

Now, the increased amount of freely available code induces the idea of using this "big code" to perform statistical learning and reasoning to solve difficult programming problems. For instance, researchers recently applied various machine learning techniques to infer different program properties and it has shown promising results. Raychev et. al. [2] used Conditional Random Fields to infer names and types of identifiers of minified JavaScript programs. Similarly, Allamanis et. al [3] utilized neural networks to develop a model called Logbilinear Context Models of Code to predict precise names for Java methods and classes.

We propose leveraging large code corpora of Java programs (for example, the Qualitas Corpus [5]) to learn different synchronization patterns used by developers of concurrent software. Then using anomaly detection techniques, we find anomalous synchronization patterns among supposedly correct patterns.

SPLASH Companion'16, October 30 – November 4, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4437-1/16/10...$15.00
http://dx.doi.org/10.1145/2984043.2998542

```
public class Z {
  private volatile String n;
  void writeDefault() {
    if (isNull()) {
      setName("default");
    }
  }
  void setName(String s) {
    n = s;
  }
  boolean isNull() {
    return (n == null);
  }
}
```

```
public class A {
  private volatile boolean b;
  synchronized void init() {
    if (!isInit()) {
      setInit(true);
    }
  }
  boolean isInit() {
    return this.b;
  }
  void setInit(boolean x) {
    this.b = x;
  }
}
```

Figure 1: Examples of two classes: `Z` is not thread-safe whereas `A` is thread-safe.

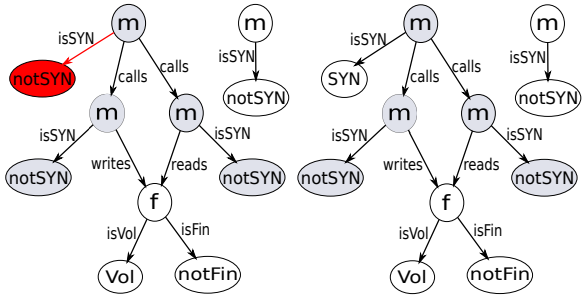

Figure 2: Generated graphs corresponding to classes in Figure 1 + detected anomaly

2. Approach

If we assume that the majority of concurrent software is properly synchronized, then the small portion of incorrectly synchronized programs should be outliers to dominant correctly synchronized programs. To analyze and detect such anomalies, we utilize a two step-approach:

2.1 Static Analysis of Source Code

We perform simple static analysis using the Soot Framework [6] to construct a graph for each program class. Each graph contains information obtained from call graph, and information about which methods read and write which fields. Moreover, we augment our graph with coarse-grained information related to synchronization. We add nodes to the graph to indicate whether each class field is `volatile` or `final`; and whether each method is synchronized (or contains a synchronized block) or not.

To perform statistical reasoning on an entire corpus, we canonicalize the graphs constructed during the static analysis by removing identifiers names and types of methods and fields. For each field and each method, we insert a fresh field node and a fresh method node respectively. Figure 2 shows the corresponding graphs of the two classes in Figure 1.

2.2 Anomaly Detection

We use graph-based anomaly detection algorithms to analyze the set of graphs generated in the previous step for potential irregularities. For this we utilize GBAD [7] which uses a variety of graph mining algorithms with a number of adjustable parameters to discover structural anomalies in data represented as graphs.

Consider the example in Figure 1. `class Z` is not thread-safe because of the missing synchronization on method `writeDefault()` whereas `class A` is thread-safe. If the corpus has enough properly synchronized classes like `class A`, then the anomaly detection tool is able to find that `class Z` is missing synchronization on method `writeDefault()` as indicated by the red (dark) node in Figure 2.

3. Preliminary Results & Ongoing work

As shown in our example, GBAD successfully discovers a simple anomaly: a missing synchronization on some method; if the analyzed corpus contains enough examples of the corresponding proper synchronization pattern. Motivated by this initial finding, we are working on extending our approach to a larger code base.

To achieve this, we address a number of issues. Our initial experiments showed that GBAD does not scale well with the number and size of analyzed graphs. We are also investigating that some data extracted during the static analysis phase might act as noise to the anomaly detection. Finally, we are considering that finer-grained information related to synchronization and control flow such (as acquisition and release of locks) might be more useful to the approach rather than the current coarser-grained synchronization abstraction.

Acknowledgments

This research is supported by the German Federal Ministry of Education and Research and by the Hessian Ministry of Science and the Arts within “CRISP”, and by the German Research Foundation within the Emmy Noether project “ConcSys”.

References

[1] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *ASPLOS ’08*, pages 329–339, 2008.

[2] V. Raychev, M. Vechev, and A. Krause. Predicting program properties from ”big code”. In *POPL ’15*, pages 111–124, 2015.

[3] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Suggesting accurate method and class names. In *ESEC/FSE ’15*, pages 38–49, 2015.

[4] C. Hammer, J. Dolby, M. Vaziri, and F. Tip. Dynamic detection of atomic-set-serializability violations. In *ICSE ’08*, pages 231–240, May 2008.

[5] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. Qualitas corpus: A curated collection of java code for empirical studies. In *APSEC ’10*, pages 336–345, December 2010.

[6] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *CASCON ’99*, pages 13–. 1999.

[7] W. Eberle and L. Holder. Discovering structural anomalies in graph-based data. In *ICDMW ’07*, pages 393–398, Oct 2007.