The Evolution of Data Races

Caitlin Sadowski Jaeheon Yi
University of California at Santa Cruz
{*supertri, jaeheon*}*@cs.ucsc.edu*

Sunghun Kim
Hong Kong University of Science and Technology
hunkim@cse.ust.hk

***Abstract*—Concurrency bugs are notoriously difficult to find and fix. Several prior empirical studies have identified the prevalence and challenges of concurrency bugs in open source projects, and several existing tools can be used to identify concurrency errors such as data races. However, little is known about how concurrency bugs evolve over time. In this paper, we examine the evolution of data races by analyzing samples of the committed code in two open source projects over a multi-year period. Specifically, we identify how the data races in these programs change over time.**

I. Introduction

Concurrency bugs are both prevalent and difficult to identify and fix [8]. Even when these bugs are fixed, the fixes may introduce new concurrency bugs [12]. The difficulty of working with concurrency has made this area the focus of many research projects and analysis tools. However, little is known about how concurrency errors and bugs evolve over time. This paper begins to address this gap by examining the history of data races in two open source projects. We find that many races exist in these projects throughout most of their history, and that variables go in and out of being racy.

II. Approach

In order to investigate the evolution of data races, we must first identify data races in a particular version of a program. There are two classes of data race analyses: static and dynamic. Static detectors take as input the source code of a program, but often report many false alarms. In contrast, dynamic detectors analyze a running program. Precise dynamic analyses produce no false alarms but are limited by the particular schedule encountered. Because they are scheduling-dependent, dynamic analyses may be non-deterministic.

We used a precise dynamic analysis to ensure that all races identified were actual races. We ran this race detector on the compiled bytecode from a sampling of the revisions of two open source Java projects. We ran the detector one hundred times per revision, and calculated the union of those one hundred runs as the set of races found at a particular revision. Then, we visualized these race data to show the number of race conditions discovered at each revision and which variables are racing at each revision.

A. Experimental Setup

We used two multithreaded Java programs that are each about 180,000 lines of code (Table I). `jEdit` [4] is a general purpose GUI editor. We used all compilable revisions from 6502 to 20033 related to bug-fixing, as identified by keywords in the change log. These 237 revisions capture over five years' worth of open source development (Figure 1).

Table I
The sampling ranges for jEdit and Columba.

Name	LOC	# Samples	Revision Range	Revision Dates
`jEdit`	175k	237	6502 – 20033	July 2006 – September 2011
`Columba`	190k	214	26 – 437	July 2006 – December 2009

Figure 1. The 237 revisions sampled for `jEdit`. Vertical lines represent sampled revisions.

`Columba` [2] is a GUI email client. We used all 214 compilable versions from over 400 versions that capture over three years' worth of open-source development (Figure 2).

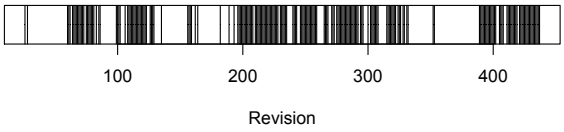

Figure 2. The 214 revisions sampled for `Columba`. Vertical lines represent sampled revisions.

We used the RoadRunner dynamic analysis framework [3], which instruments Java bytecode and provides a simple API to pass the event stream to custom analysis tools. Analysis tools only need to define methods to handle events of interest (e.g. synchronization events and variable accesses). All classes loaded by the benchmarks were instrumented, with the exception of the Java standard libraries. Since `Columba` and `jEdit` are reactive programs, we tested them using a very simple script that just opened then closed

978-1-4673-1761-0/12/$31.00 © 2012 IEEE

each GUI program. This simple test was enough to capture a variety of interesting races; investigating the evolution of data races on more complicated benchmarks over a larger variety of programs remains an area for future work.

To detect data races, we used two standard happens-before race detectors that are included with the RoadRunner framework. Each object field and array element is tracked with a distinct shadow object. These detectors are precise, in that they do not produce false positives; all reported races represent actual racy variables. Some of these reported races may be "benign", or not associated with an actual bug, although it is debatable whether any race can rightly be classified as benign [1]. Because we are running a dynamic race detector, we have the potential for false negatives; variables may be racy even though this is not reported by our detector, even after we merged results from one hundred runs.

III. Results

We looked at two aspects of how data races change over time: the total count and the specific racy variables.

A. Counting Races

Finding 1: The number of racy variables remains high, and may even increase, over time.

Figures 3 and 4 show the number of races at each revision for `Columba` and `jEdit`, respectively. For `jEdit`, the number of races at the end of the sampling period is much higher than the number of races at the beginning. `Columba` does not have such a proportionally dramatic upward trend, although the number of races at the last sample point is higher than the number at most previous sampling points. Interestingly, both projects exhibit flat periods where repository commits do not affect the number of races. This suggests that fixing races may not be a priority for developers or they may not be aware of racy variables. Perhaps developers do not consider many races to be "bugs" [5].

We further investigated the commit logs at the points in `Columba` where the number of races changes. We found that most revisions which resulted in a decrease in the number of races seemed like either refactoring changes (e.g. *cleanup of toolbar* in revision 106) or bug fixes (e.g. *fixed contextual search nullpointer exception* in revision 61). In contrast, the sharpest increases in the number of races occurred when adding new features (e.g. *hide tag feature* in revision 267). Several revisions marked as bug fixes also resulted in an increase of races.

B. Racy Variables

Finding 2: Over the course of a project, variables may go in and out of being racy.

Figures 5 and 6 show which variables are racy at each sampled revision point for `Columba` and `jEdit`, respectively. These graphs contain a list of all variables which are racy at any revision on the y-axis. On the x-axis are the revision

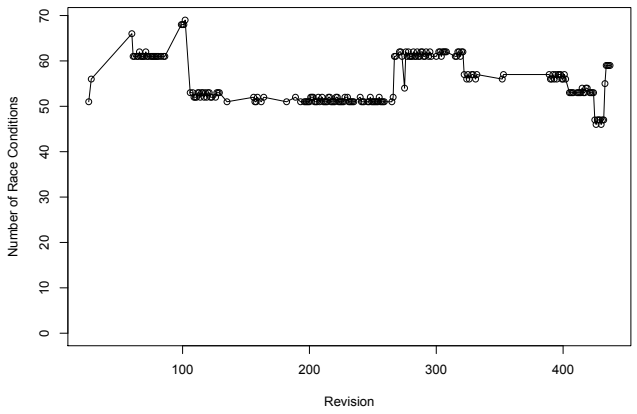

Figure 3. The number of races over time for `Columba`.

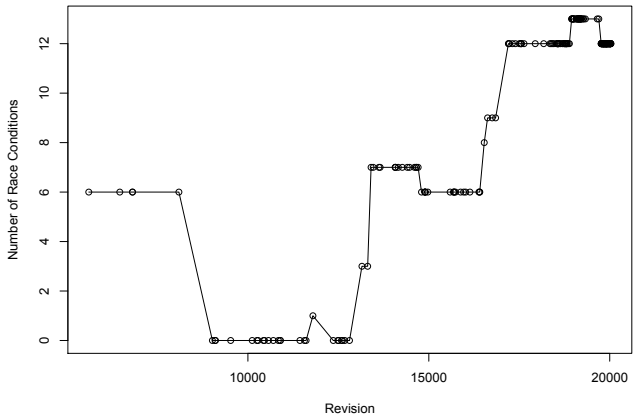

Figure 4. The number of races over time for `jEdit`.

samples; note that two consecutive samples may have several revisions in between. We use samples for the units so that all gaps for a variable represent periods when we did not detect the variable racing. Each point on these graphs represents the fact that the corresponding variable (on the y-axis) was found to be racy at that particular sample (x-axis). Each variable is assigned a particular colour to distinguish it from other variables in the graph. For example, in Figure 6, `DynamicMenuChanged.name` is light blue and is racy from about the 70th sampled revision.

Although many variables become racy and then persist that way, other variables move in and out of racy status. For example, the variable `propertyManager` in `jEdit` is briefly racy before the 50th sample point, then is not racy for a spell, then is racy again. Note that we do not track variable renaming in our analysis, so some gaps could be the result of a variable that is renamed and later reverted

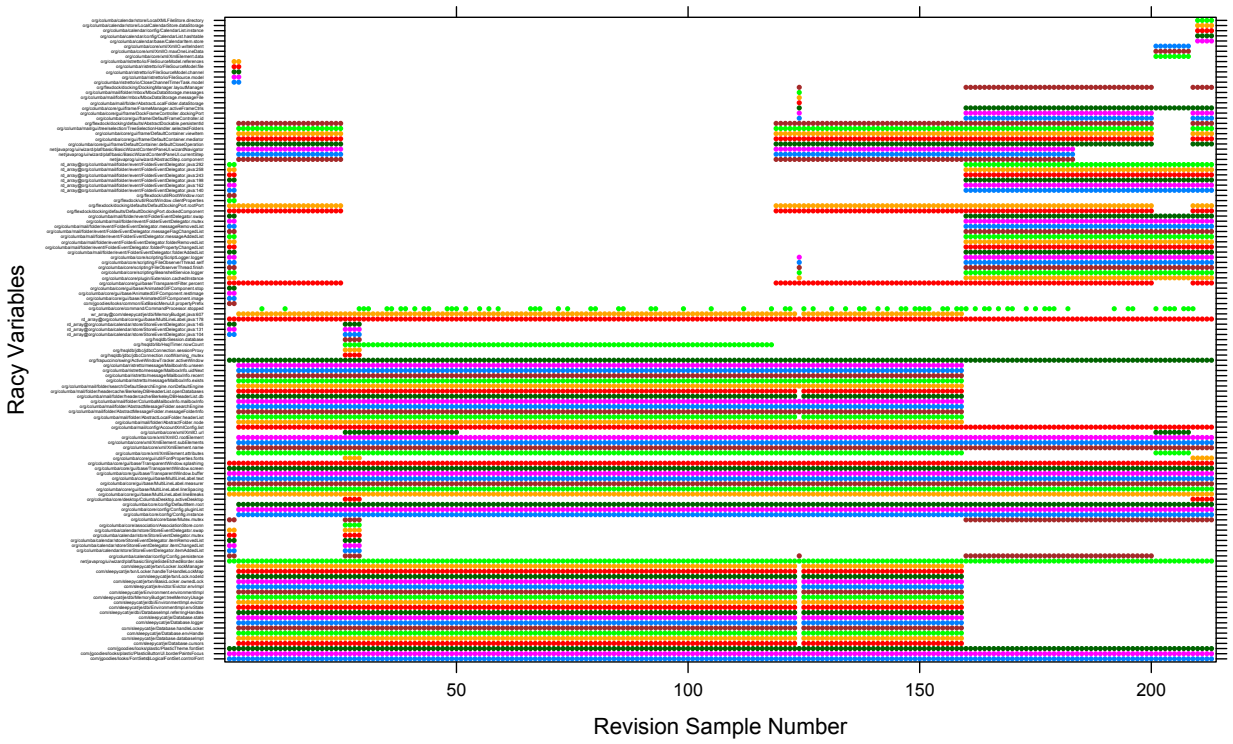

Figure 5. `Columba`: racy variables over revisions. The y axis lists all variables which were found to be racing on any revision for `Columba`. The x axis lists each revision. The graph contains a point for each revision in which a variable is racing.

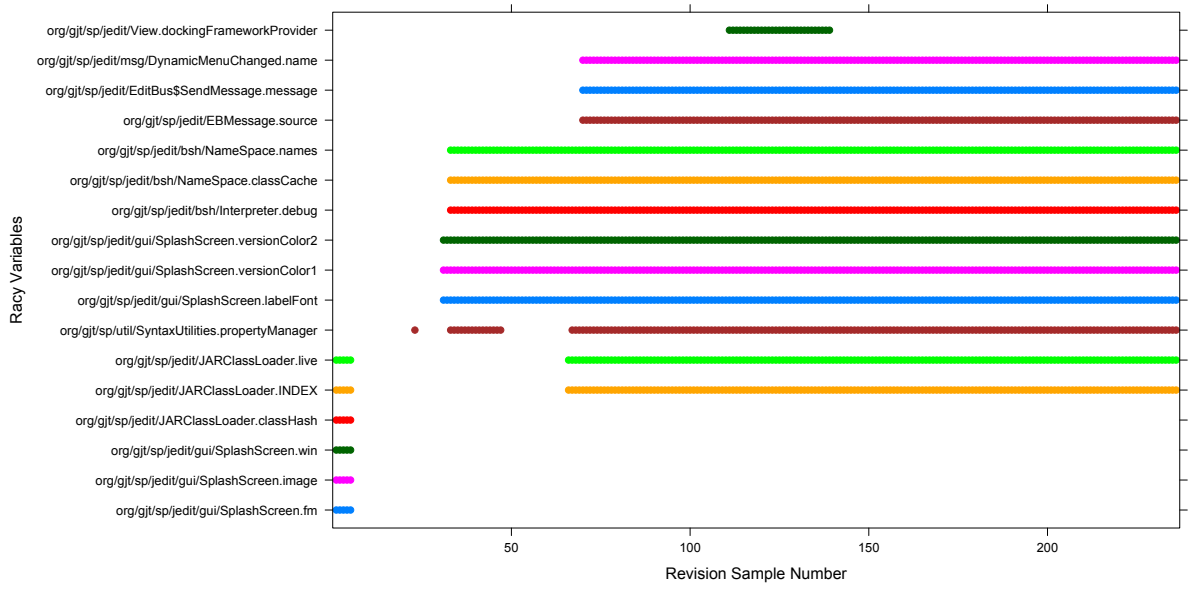

Figure 6. `jEdit`: racy variables over revisions. The y axis lists all variables which were found to be racing on any revision for `jEdit`. The x axis lists each revision. The graph contains a point for each revision in which a variable is racing.

to the original name. Although it is also possible that our detector is reporting a false negative when we encounter an on-again off-again pattern, this pattern (represented by gaps in the horizontal lines in these figures) happens many times (e.g. in `Columba`).

IV. Related Work

To the best of our knowledge, no prior work has looked at the evolution or history of warnings produced by concurrency-related dynamic analysis tools. DynaMine [7] was the first tool which combined dynamic analysis and repository mining. However, DynaMine is focused on identifying correct usage patterns and leveraging them to predict future errors.

A couple papers have examined the evolution of warnings produced by static analysis tools. One of them analyzed the decay likelihood for various types of vulnerabilities [9]. The other paper investigated whether particular warnings given by static analysis tools were associated with bug-fixing commits [6]. In this paper, the warning history, gleaned by running the static analysis tool at each revision, was leveraged to improve warning prioritization.

One prior paper examined the evolution of concurrency in open source software projects [11]. This paper found that most projects involve concurrency, the use of concurrency is rising, and programmers do not always use the higher-level data structures they have available to them. However, this paper was focused on the use of concurrency constructs (such as `synchronized` blocks), not the presence of concurrency-related errors.

V. Conclusion and Future Work

Overall, we find that there may be many racy variables across the history of a project, and that some variables go in and out of being racy. These results emphasize the fact that working with concurrency is difficult, and that data races may frequently plague programs. Developers need to work carefully to ensure that subsequent changes do not reintroduce a fixed concurrency bug. It would be interesting to further study how often concurrency bugs are introduced that are the same or similar to prior bugs.

The dynamic analysis we ran has no false positives, and so gives a lower bound to the number of races that exist at a particular revision. It would be interesting to combine the experimental data collected with the results of a static race detector that gives no false negatives, thus providing an upper bound to the number of races that exist at a particular revision. Also, some of the races identified may be benign. It would be interesting to explicitly relate the results of this study to races that are implicated in real bugs.

In a prior study [10] we found that little is known about how other structures related to concurrent program correctness, such as atomic blocks, evolve over time. We would like to investigate the evolution of other concurrency errors and constructs. As a first step, we wish to identify which portions of a program are deterministic or atomic across many revisions. By better understanding what code changes break or expand these code sections, we may be able to identify buggy or tricky code changes.

We would also like to run our analyses on more revisions for more programs, and expand the results accordingly. Previous studies [6] have leveraged the fault history when predicting new bugs. Perhaps future concurrency bug detection algorithms could incorporate data from changelists that introduce or eliminate concurrency bugs.

References

[1] S. Adve. Data races are evil with no exceptions: Technical perspective. *Communications of the ACM*, 53:84–84, Nov. 2010.

[2] Columba. http://columba.svn.sourceforge.net.

[3] C. Flanagan and S. N. Freund. The RoadRunner dynamic analysis framework for concurrent programs. In *Program Analysis for Software Tools and Engineering (PASTE)*, 2010.

[4] jEdit. http://jedit.svn.sourceforge.net.

[5] B. Kasikci, C. Zamfir, and G. Candea. Data races vs. data race bugs: telling the difference with portend. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[6] S. Kim and M. Ernst. Which warnings should I fix first? In *International Symposium on Foundations of Software Engineering (FSE)*, 2007.

[7] B. Livshits and T. Zimmermann. DynaMine: Finding common error patterns by mining software revision histories. *ACM SIGSOFT Software Engineering Notes*, 30(5):296–305, 2005.

[8] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. *SIGPLAN Notices*, 43(3):329–339, 2008.

[9] M. D. Penta, L. Cerulo, and L. Aversano. The life and death of statically detected vulnerabilities: An empirical study. *Information and Software Technology*, 51(10):1469 – 1484, 2009.

[10] C. Sadowski and S. Kurniawan. Heuristic evaluation of programming language features. In *Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU)*, 2011.

[11] W. Torres, G. Pinto, B. Fernandes, J. P. Oliveira, F. Ximenes, and F. Castor. Are Java programmers transitioning to multicore? A large scale study of Java FLOSS. In *Transitioning to MultiCore Workshop (TMC)*, 2011.

[12] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? In *International Symposium on Foundations of Software Engineering (FSE)*, 2011.