CalFuzzer: An Extensible Active Testing Framework for Concurrent Programs

Pallavi Joshi[1], Mayur Naik[2], Chang-Seo Park[1], and Koushik Sen[1]

[1] University of California, Berkeley, USA
{pallavi,parkcs,ksen}@eecs.berkeley.edu
[2] Intel Research
mayur.naik@intel.com

Abstract. Active testing has recently been introduced to effectively test concurrent programs. Active testing works in two phases. It first uses imprecise off-the-shelf static or dynamic program analyses to identify potential concurrency bugs, such as data races, deadlocks, and atomicity violations. In the second phase, active testing uses the reports from these imprecise analyses to explicitly control the underlying scheduler of the concurrent program to accurately and quickly discover real concurrency bugs, if any, with very high probability and little overhead. In this paper, we present an extensible framework for active testing of Java programs. The framework currently implements four active testers based on data races, atomic blocks, deadlocks, and user-specified breakpoints.

1 Introduction

Multi-threaded programs often exhibit wrong behaviors due to unintended interference between executing threads. Such concurrency bugs—such as data races and deadlocks—are often difficult to find because they typically happen under very specific interleavings of the executing threads. A traditional method of testing concurrent programs is to repeatedly execute the program with the hope that different test executions will result in different interleavings. There are a few problems with this approach. First, testing being carried out in a particular environment often fails to come up with interleavings that can potentially happen in other environments, such as under different system loads. Second, testing depends on the underlying operating system or the virtual machine for thread scheduling—it does not try to explicitly control the thread schedules; therefore, it often ends up executing the same interleaving many times.

Numerous program analysis techniques [2, 5, 1, 4] have been developed to *predict* concurrency bugs in multi-threaded programs by detecting violations of commonly used synchronization idioms. For instance, accesses to a memory location without holding a common lock are used to predict data races on the location, and cycles in the program's lock order graph are used to predict deadlocks. These techniques are very effective at finding concurrency bugs because they can predict bugs that could potentially happen during a real execution—for such a prediction, they do not need to see actual executions (in the case of static analysis) or they merely need to see an execution that exhibits a violation of the idiom without necessarily exhibiting the bug itself (in the case of dynamic analysis). However, despite recent advances, these techniques often report many false warnings. Going through all of these warnings and reasoning about them manually often turns out to be time consuming.

Recently, we have proposed a new technique for finding *real* bugs in concurrent programs, called *active testing* [8, 6, 3]. Active testing uses a randomized thread scheduler to verify if the warnings reported by an imprecise static or dynamic program analysis are real bugs. The technique works as follows. Active testing first uses an existing imprecise off-the-shelf static or dynamic analysis technique, such as Lockset [7, 5], Atomizer [1], or Goodlock [2], to compute a set of potential concurrency bugs. Each potential concurrency bug is typically identified by a set of program statements. For example, in the case of a data race, the set contains two program statements that could potentially race with each other in some concurrent execution. For each potential concurrency bug, active testing runs the concurrent program under test under random schedules. Further, active testing biases the random scheduling by pausing the execution of any thread when the thread reaches a statement involved in the potential concurrency bug. After pausing a thread, active testing also checks if a set of paused threads could have created a real concurrency bug. For example, in the case of a data race, active testing checks if two paused threads are about to access the same memory location and at least one of them is a write. Thus, active testing attempts to force the program to take a schedule in which the concurrency bug actually occurs. In our previous work, we have developed active testing algorithms for detecting real data races, atomicity violations, and deadlocks.

In this paper, we describe an extensible tool for active testing of concurrent Java programs, called CalFuzzer. CalFuzzer provides a framework for implementing custom schedulers for active testing. We call these custom schedulers *active checkers*. The tool also provides a framework and libraries for implementing various imprecise dynamic analysis techniques, such as Lockset, Atomizer, and Goodlock. A set of program statements involved in a bug could be computed using these dynamic analysis techniques. CalFuzzer also allows the user to manually specify a set of program statements where an active checker should pause. Such statements may be thought of as "concurrent breakpoints".

We have implemented three active checkers in CalFuzzer for detecting real data races, atomicity violations, and deadlocks. We have also implemented three dynamic analysis techniques in CalFuzzer. They are a hybrid race detector [5], Atomizer [1] for finding potential atomicity violations, and iGoodlock [3] for detecting potential deadlocks. CalFuzzer provides a method `ActiveChecker.Check()` that the user can use in her concurrent program to specify arbitrary "pause" points (or "breakpoints") in the program.

We have applied CalFuzzer to several concurrent Java benchmark programs containing a total of 600K lines of code and have detected both previously known and unknown data races, atomicity violations, and deadlocks. CalFuzzer could easily be extended to detect other kinds of concurrency bugs, such as missed notifications and atomic set violations.

Algorithm 1 CalFuzzer with user defined `analysis()` and `check()` methods

```
Inputs: the initial state $s_0$ and a set of transitions $breakpoints$
$paused := \emptyset$
$s := s_0$
while $\texttt{Enabled}(s) \neq \emptyset$ do
    $t :=$ a random transition in $\texttt{Enabled}(s) \setminus paused$
    $\texttt{analysis}(t)$
    if $t \in breakpoints$ then
        $paused := \texttt{check}(t, paused)$
    end if
    if $t \notin paused$ then
        $s := \texttt{Execute}(s, t)$
    end if
    if $paused = \texttt{Enabled}(s)$ then
        remove a random element from $paused$
    end if
end while
if $\texttt{Alive}(s) \neq \emptyset$ then
    print "ERROR: system stall"
end if
```

2 The Active Testing Framework

In this section, we give a high-level description of our active testing framework. We consider a concurrent system composed of a finite set of threads. Given a concurrent state s, let $\texttt{Enabled}(s)$ denote the set of transitions that are enabled in the state s. Each thread executes a sequence of transitions and communicates with other threads through shared objects. We assume that each thread terminates after the execution of a finite number of transitions. A concurrent system evolves by transitioning from one state to another state. If s is a concurrent state and t is a transition, then $\texttt{Execute}(s, t)$ executes the transition t in state s and returns the updated state.

The pseudo-code in Algorithm 1 describes the CalFuzzer algorithm. The algorithm takes an initial state s_0 and a set of transitions (denoting a potential concurrency bug), called *breakpoints*, as input. The set of transitions *paused* is initialized to the empty set. Starting from the initial state s_0, at every state, CalFuzzer randomly picks a transition enabled at the state and not present in the set *paused*. It then calls the user defined method `analysis` to perform a user defined dynamic analysis, such as Lockset, Atomizer, or Goodlock. The `analysis` method can maintain its own local state; for example, the local state could maintain lockset and vector clocks in the case of hybrid race detection. If transition t is in the set *breakpoints*, CalFuzzer invokes the user provided method `check`, which takes t and the *paused* set as input and returns an updated *paused* set. The `check` method could be used to implement various active checkers. A typical implementation of the `check` method could add t to the *paused* set

Algorithm 2 The `check` method for active testing of data races

```
1: Inputs: transition $t$ and a set of transitions $paused$
2: if $\exists t' \in paused$ such that $t$ and $t'$ access same location and one of them is write
   then
3:     print "Real data race between $t$ and $t'$" (* next resolve race randomly to check
       if something could go wrong due to the race *)
4:     if random boolean then
5:         add $t$ to $paused$ and remove $t'$ from $paused$
6:     end if
7: else
8:     add $t$ to $paused$
9: end if
10: return $paused$
```

and remove some transitions from the *paused* set. After the invocation of `check`, CalFuzzer executes the transition t if it has not been added to the *paused* set by the `check` method. At the end of each iteration, CalFuzzer removes a random transition from the *paused* set if all the enabled transitions have been paused. The algorithm terminates when the system reaches a state that has no enabled transitions. At termination, if there is at least one thread that is alive, the algorithm reports a system stall.

CalFuzzer thus takes two user defined methods: `analysis` and `check`. In order to implement an active testing technique, one needs to define these two methods. For example, an active testing technique for data races [8] would require us to implement hybrid race detection in the `analysis` method and a `check` method as shown in Algorithm 2.

3 Implementation Details

We have implemented the CalFuzzer active testing framework for Java. CalFuzzer (available from `http://srl.cs.berkeley.edu/~ksen/calfuzzer/`) instruments Java bytecode to insert the following callback functions before or after various synchronization operations and shared memory accesses: `startBefore`, `joinAfter`, `waitAfter`, `notifyBefore`, `notifyAllBefore`, `lockBefore`, `unlockAfter`, `readBefore`, `writeBefore`. These callback functions are implemented to invoke various dynamic analyses and active checkers. An imprecise dynamic analysis is typically implemented by implementing an interface called `Analysis`. The interface contains the callback functions described above.

An active checker is implemented by extending the class `ActiveChecker` declared below.

```
public class ActiveChecker {
    final public static Object lock = new Object();
    final protected void block(int milliSeconds) { ... }
    final protected void unblock(int milliSeconds) { ... }
    final public static void blockIfRequired() { ... }
    public void check(Collection<ActiveChecker> checkers) {
        block(0);
    }
    final public void check() { ... }
}
```

An instance of a subclass of `ActiveChecker` (e.g. `RaceChecker`) is equivalent to a transition in Algorithm 1. The `check()` method defined by this class is equivalent to the `check` method used in Algorithm 1. This method in turn calls method `check(Collection<ActiveChecker> checkers)`. A subclass of `ActiveChecker` should override the `check(Collection<ActiveChecker> checkers)` method. The default implementation blocks (i.e. pauses) the current thread (i.e. the transition denoted by the `ActiveChecker` object). The user can use `ActiveChecker.Check()` in the code under analysis to indicate a "breakpoint". The class also provides other methods, such as `block(int milliSeconds)` to pause the current thread and `unblock(int milliSeconds)` to continue a paused thread. These methods should be used in a custom implementation of the `check(Collection<ActiveChecker> checkers)` method.

The framework also provides various utility classes, such as `VectorClockTracker` and `LocksetTracker` to compute vector clocks and locksets at runtime. Methods of these classes are invoked in the various callback functions described above. These utility classes are used in the hybrid race detection [5] and iGoodlock [3] algorithms. Other user defined dynamic analyses could also use these utility classes.

The instrumentor of CalFuzzer modifies all bytecode associated with a Java program including the libraries it uses, except for the classes that are used to implement CalFuzzer. This is because CalFuzzer runs in the same memory space as the program under analysis. CalFuzzer cannot track lock acquires and releases by native code. As such, there is a possibility that CalFuzzer can go into a deadlock if there are synchronization operations inside uninstrumented classes or native code. To avoid such scenarios, CalFuzzer runs a low-priority monitor thread that periodically polls to check if there is any deadlock. If the monitor discovers a deadlock, then it removes one random transition from the *paused* set.

CalFuzzer can also go into livelocks. Livelocks happen when all threads of the program end up in the *paused* set, except for one thread that does something in a loop without synchronizing with other threads. We observed such livelocks in a couple of our benchmarks including `moldyn`. In the presence of livelocks, these benchmarks work correctly because the correctness of these benchmarks assumes that the underlying Java thread scheduler is fair. In order to avoid livelocks, CalFuzzer creates a monitor thread that periodically removes those transitions from the *paused* set that are waiting for a long time.

4 Results

Table 1 summarizes some of the results of running active testing on several real-world Java programs. Further details are available in [8, 6, 3]. Note that the bugs reported by the active checkers (e.g. RaceFuzzer, AtomFuzzer, and DeadlockFuzzer) are real concurrency bugs, whereas the bugs reported by the dynamic analyses (e.g. hybrid race detection, iGoodlock, and Atomizer) could be false warnings.

		Number of reported bugs					
Benchmark	LoC	HRD	RaceFuzzer	iGoodlock	DeadlockFuzzer	Atomizer	AtomFuzzer
jspider	10,252	29	0	0	0	28	4
sor	17,718	8	0	0	0	0	0
hedc	25,024	9	1	0	0	3	0
jigsaw	160,388	547	36	283	29	60	2
Java Swing	337,291	-	-	1	1	-	-

Table 1. Results for benchmarks: LoC is Lines of Code; HRD is Hybrid Race Detection

CalFuzzer provides a framework for writing custom randomized schedulers that could quickly find real bugs. We have currently implemented three active checkers in this framework and we believe that CalFuzzer provides a simple extensible framework to experiment with other active checkers and dynamic analysis techniques.

References

1. C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multi-threaded programs. In *31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–267, 2004.
2. K. Havelund. Using runtime analysis to guide model checking of java programs. In *7th International SPIN Workshop on Model Checking and Software Verification*, pages 245–264, 2000.
3. P. Joshi, C.-S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09) (to appear)*, 2009.
4. M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 308–319, 2006.
5. R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 167–178. ACM, 2003.
6. C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 135–145, New York, NY, USA, 2008. ACM.
7. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.
8. K. Sen. Race directed random testing of concurrent programs. In *Programming Language Design and Implementation (PLDI'08)*, 2008.