Testing Concurrent Programs

Sri Teja Basava
Arpit Sud
CSCI 5535: Fundamentals of Programming Languages
University of Colorado at Boulder
Spring 2010

Problem Statement

• Survey tools for testing concurrent programs, with an emphasis on
  • CHESS
  • CTrigger
  • CalFuzzer
• Use CalFuzzer to discover real data races in Scala’s concurrent programs that use Scala’s Actor concurrency model
Motivation

• Debugging concurrent programs is hard!
• Concurrent programs exhibit bugs not present in sequential programs, such as data races, deadlocks, and atomicity violations
• Traditional approach to testing concurrent programs is to repeatedly execute the program
• Problems with traditional approach
  • Testing carried out in a particular environment fails to come up with interleavings that could happen in other environments
  • Testing depends on the underlying scheduler and often ends up executing the same interleaving many times

Concurrency Bugs

• Data Race: A data race occurs when two threads access the same memory location without any intervening synchronization operations, and at least one of the accesses is a write
• Deadlock: A deadlock occurs when two or more threads cannot make progress because each thread is waiting for another thread to release a resource
• Atomicity Violation: An atomicity violation occurs when a particular code region intended to be atomic is violated, causing unintended results
CHESS

• A tool to discover “Heisenbugs”
• Uses model checking to generate all possible interleavings and then bounds the number of preemptive context switches to limit possible interleavings
• Provides wrappers for OS concurrency APIs and dynamically intercepts calls to these APIs during runtime
• Can successfully reproduce erroneous execution

CTrigger

• Detects atomicity violations by focusing on “unserializable interleavings,” i.e. interleavings which are not equivalent to any sequential execution
• Works in two phases
  • Phase I: Identify unserializable interleavings
    • Step 1: Profiling and identifying unserializable interleavings
    • Step 2: Pruning infeasible interleavings
    • Step 3: Ranking low-probability interleavings
  • Phase II: Controlled testing of unserializable interleavings
CalFuzzer

• An extensible tool for active testing of concurrent Java programs

• **Active testing** is a dynamic analysis technique that works in two phases
  - *Phase 1*: It uses a predictive static or dynamic analysis to compute potential concurrency bugs
  - *Phase 2*: It determines whether each reported bug is a real bug or a false positive by attempting to create an interleaving that exhibits the bug

• Active testing can find real data races, deadlocks, and atomicity violations

RaceFuzzer

• Active testing applied to finding real data races in concurrent programs

• The two phases
  - *Phase 1*: It first uses an imprecise race detection technique, such as hybrid race detection, to compute a set of pairs of statements that could potentially race in some interleavings
  - *Phase 2*: For each pair, it executes the program as follows:
    • It picks a random thread to execute at each state
    • If the next statement of the thread is one of the statements in the pair, it pauses the thread
    • If it comes across another statement also in the pair, it reports a real race if this statement races with the paused thread
Scala’s Actor Model

- Actors are active objects
  - Objects: Actors encapsulate state and behavior
  - Active: Actors run in separate threads

- Actors communicate through asynchronous message passing
  - An actor normally sends a message to another actor by invoking the receiving actor’s `!` method, which queues the message in the receiving actor’s mailbox and returns immediately (i.e., it is non-blocking)
  - An actor dequeues messages from its mailbox by calling its `receive` method, which is blocking

RaceFuzzer on Scala Programs

- Two sources of race conditions
  - One of the tenets of the Actor model is that actors shouldn’t share state, but Scala does not enforce this, so Scala’s Actor model suffers from the same problems as the traditional Thread model
  - The order in which messages are delivered to an actor may be non-deterministic, and an unexpected order may lead to a failure
  - Unfortunately, we haven’t made much progress with using RaceFuzzer to detect data races in Scala programs :(}
Questions?