This project is kindly supported by NSF and TEXAS ATP

Practical Tools for Testing Concurrent Programs

Corky Cartwright, Mathias Ricken
{cork | mgricken}@rice.edu

Unpredictable Behavior of Concurrent Unit Tests

Unit Tests Are Effective...

- Help find and prevent bugs in sequential code
- Tests must pass before code changes are committed
  → Fewer bugs enter the repository
- Tests exhibiting fixed bugs are added to test suite
  → Prevent old bugs from reappearing
- Current tools: JUnit, TestNG, Ant

... But Not for Concurrent Programs

- Thread switching is non-deterministic and machine-specific
- Success of a unit test does not imply correct behavior under all possible schedules
- Most programs are concurrent
  → GUI: separate thread for display
  → Multi-core: programs must be concurrent to benefit
- Current tools not effective or easy to use on large projects

Practical Tools for Concurrency

(1) **A concurrency-aware extension of JUnit**

(2) **Lightweight dynamic and static checking of concurrency invariants using Java annotations**

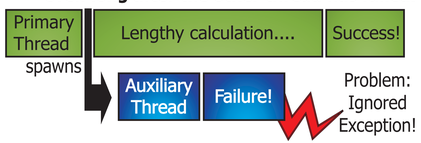
(3) **Execution with short delays inserted at critical places to test different execution schedules**

(4) **Recording of program execution and replaying according to a predetermined schedule**

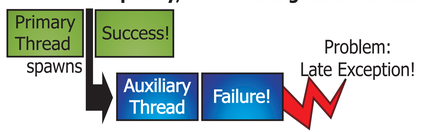
Concurrency-Aware Unit Testing Framework

Current Frameworks JUnit, TestNG Are Broken!

- JUnit, TestNG, etc. were not designed with concurrency in mind
  → The criteria for success and failure are incorrect for concurrent programs
  → The design promotes writing tests that succeed by default even when they should fail
- Writing good concurrent unit tests is as hard as writing good concurrent programs
- Problem 1: Exceptions are not automatically detected and considered a test failure.
  → The program fails in an auxiliary thread, but the unit test nonetheless succeeds, even long after the other thread has failed.



- Solution: Use a default exception handler
- Problem 2: The test does not have to wait for all auxiliary threads to terminate.
  → The program fails in an auxiliary thread, but the primary thread completes the unit test so quickly, that failure goes unnoticed.



- Solution: Wait for all spawned auxiliary threads

Inserting Delays and Scheduled Replay

- Custom class loader rewrites class files as they are loaded
- Modified class files will run on any Java VM
  → Portable and open-source
- Bytecode can be dynamically compiled by an embedded JIT compiler
  → Faster than interpretation
- When an atomic block ends, a delay may be inserted, forcing the program into a different schedule

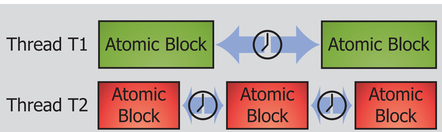


Figure: Random delays inserted between atomic blocks

- Same technique can be used for record and reply by schedule
- At the end of an atomic block, all threads are put on hald, and only the scheduled one resumes
- Difficult to implemenent a Java scheduler in the same Java VM
  → Scheduler itself may create and destroy values, influencing garbage collection and schedule

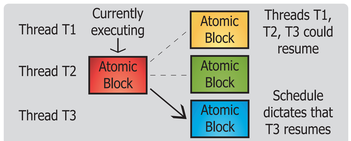


Figure: Transfer of control at the end of an atomic block

Lightweight Checking of Concurrency Invariants Using Java Annotations

Java Annotations to Specify Invariants

- Declare which classes and methods should not be or may only be executed by certain threads
- Simple to introduce into existing large projects
  → Just annotate parts of the projects, no need for a complete rewrite
- Dynamic checks to maintain invariants are automatically inserted
  → Some checks are also performed statically

A Way to Specify API Threading Discipline

- Maintaining a threading discipline is problematic for current APIs
  → APIs are often extensible, so API designers do not have full control
  → Invariants are written as comments or white papers at best
- Annotations record violations: They are not just comments anymore!
- Utility can apply annotations to existing libraries using external XML specification files
  → No need to access source code or even recompile
  → XML files written for a library can be reused and openly shared
  → We have started writing a specification file for the Java Swing API

DrJava Case Study

- DrJava recently exhibited a number of concurrency-related problems
  → We believe we have diligently fixed most problems by manually
- To test the utility and to make future development of DrJava easier, we annotated a current version and several old versions of DrJava
- The most recent version had a lower percentage of thread checks failing
  → Had the annotations not been done after the fact, they probably would have identified even more problems
  → Still identified several potential bugs in the current version

Augmenting Type Systems Using Annotations

- By introducing annotations describing permitted threading behavior, we have in effect augmented Java's type system.
  → Subtyping problems: C <: D according to Java, but not by our checking
- Formulation of the utility's subtyping relation based on Featherweight Java
- Annotations can be used to augment Java's type system to get additional guarantees; material for future research.