Concurrency Assurance in Fluid

Reading: Assuring and Evolving Concurrent Programs: Annotations and Policy

17-654/17-765
Analysis of Software Artifacts

Guest Lecturer: Aaron Greenhouse
Software Engineering Institute

Example: java.util.logging.Logger

```
public class Logger { ...
    private Filter filter;
    ...
    public void setFilter(Filter newFilter) {...
        if (!anonymous) manager.checkAccess();
        filter = newFilter;
    }
```
Example: java.util.logging.Logger

```java
public class Logger {
    private Filter filter;

    public void setFilter(Filter newFilter) {
        if (!anonymous) manager.checkAccess();
        filter = newFilter;
    }

    public void log(LogRecord record) {
        synchronized (this) {
            if (filter != null && !filter.isLoggable(record)) return;
        }
    }
}
```

Example: java.util.logging.Logger

```java
/** ... All methods on Logger are multi-thread safe. */
public class Logger {
    private Filter filter;

    /** ...
     * @param newFilter a filter object (may be null)
     */
    public void setFilter(Filter newFilter) {
        if (!anonymous) manager.checkAccess();
        filter = newFilter;
    }

    public void log(LogRecord record) {
        synchronized (this) {
            if (filter != null && !filter.isLoggable(record)) return;
        }
    }
}
```

Consider log() in isolation

Consider class Logger in its entirety!
Example: java.util.logging.Logger

```java
/** ... All methods on Logger are multi-thread safe. */
public class Logger {
    private Filter filter;

    /** ...
     * @param newFilter a filter object (may be null)
     */
    public void setFilter(Filter newFilter) {
        if (!anonymous) manager.checkAccess();
        filter = newFilter;
    }

    public synchronized void log(LogRecord record) {
        synchronized (this) {
            if (filter != null && !filter.isLoggable(record)) return;
        }
    }
}
```

**Correction:** synchronize `setFilter()`

Class Logger has a **race condition.**
Example: Summary 1

**Problem:** Race condition in class `Logger`

- **Race condition** defined:
  
  (From Savage et al., *Eraser: A Dynamic Data Race Detector for Multithreaded Programs*)
  
  - Two threads access the same variable
  - At least one access is a write
  - No explicit mechanism prevents the accesses from being simultaneous

Example: Summary 2

**Problem:** Race condition in class `Logger`

- Non-local error
  
  - Had to inspect whole class
    
    • Bad code invalidates good code
  
  - Could have to inspect all clients of class

- Hard to test
  
  - Problem occurs non-deterministically
    
    • Depends on how threads interleave
Example: Summary 3

Problem: Race condition in class Logger

• Need to know design intent
  – Should instances be used across threads?
  – If so, how should access be coordinated?
    • Assumed log was correct: synchronize on this
    • Could be caller’s responsibility to acquire lock
      ⇒ log is incorrect
      ⇒ Need to check call sites of log and setFilter

The Fluid Project

Assure code is consistent with programmer-specified design intent.
  – Assure critical dependability attributes
    • Tend to defy testing and inspection
    • Provide direct static assurance
  – Express dependability-related models
    • Incrementally capture design intent
  – Emphasize adoptability and scalability
    • Ease of use by practicing developers
    • Composability and components
    • Incrementality and early rewards
    • Partiality and contingency
Topics

• Capturing design intent

• Assuring code–model consistency
  – Cutpoints and incrementality
  – Overview of the lock analysis
  – Real world complications

• Fluid Demo
  – Annotations
  – Incrementality

• Case study experiences

Fluid: Models are Missing

• Programmer design intent is missing
  – Not explicit in Java, C, C++, etc
    • What lock protects this object?
      – “This lock protects that state”
    • What is the actual extent of shared state of this object?
      – “This object is ‘part of’ that object”

• Adoptability
  – Programmers: “Too difficult to express this stuff.”
  – Fluid: Minimal effort — concise expression
    • Capture what programmers are already thinking about
    • No full specification

• Incrementality
  – Programmers: “I’m too busy; maybe after the deadline.”
  – Fluid: Payoffs early and often
    • Direct programmer utility — negative marginal cost
    • Increments of payoff for increments of effort
Capturing Design Intent

- **What data is shared by multiple threads?**
  - Annotate class: `@lock FL is this protects filter`

- **What locks are used to protect it?**
  - Annotate method: `@requiresLock FL`

- **Whose responsibility is it to acquire the lock?**
  - Annotate field: `@aggregate ... into Instance`

Categories of Design Intent

- **Safe concurrency**
  - Race conditions
  - Lock management
  - Single thread concurrency control
  - Lock ordering and deadlocks

- **Policy compliance**
  - API policy compliance
  - Framework compliance
  - Object references and aliasing
  - Patterns, uses, structure

- **Code safety**
  - Ignored exceptions
  - Appropriate typing

- **Real time**
  - Real-time thread/memory policies

*Hard to Test — Hard to Inspect*
Races and security

Source: Bugtraq vulnerabilities list
- 15-11-2003: monopd Race Condition Denial of Service Vulnerability
- 10-10-2003: Microsoft Windows RPCSS Multi-thread Race Condition Vulnerability
- 23-08-2003: Glibc Malloc Routine Race Condition Vulnerability
- 26-06-2003: Linux 2.4 Kernel execlp() System Call Race Condition Vulnerability
- 29-04-2003: Worker Filemanager Directory Creation Race Condition Vulnerability
- 23-04-2003: SAP Database SDINST Race Condition Vulnerability
- 20-04-2003: Microsoft Windows Service Control Manager Race Condition Vulnerability
- 15-03-2003: Samba REG File Writing Race Condition Vulnerability
- 27-02-2003: Hypermail Local Temporary File Race Condition Vulnerability
- 11-02-2003: Sun Microsystems Solaris Mail Reading Local Race Condition Vulnerability
- 27-01-2003: Sun Solaris AT Command Race Condition Vulnerability
- 12-01-2003: BitMover BitKeeper Local Temporary File Race Condition Vulnerability
- 20-12-2002: Tmpwatch Race Condition Vulnerability
- 20-12-2002: STMPClean Race Condition Vulnerability
- 29-07-2002: Multiple Vendor BSD pppd Arbitrary File Permission Modification Race Condition Vulnerability
- 29-07-2002: Util-linux File Locking Race Condition Vulnerability
- 04-07-2002: BEA Systems WebLogic Server and Express Race Condition Denial of Service Vulnerability

Reporting Code–Model Consistency

- Tool analyzes consistency
  - No annotations ⇒ no assurance
  - Identify likely model sites

- Three classes of results
  Code–model consistency
  Code–model inconsistency
  Informative — Request for annotation
Incremental Assurance

Payoffs early and often to reward use

• Reassure after every save
  – Maintain model-code consistency
  – Find errors as soon as they are introduced

• Focus on interesting code
  – Heavily annotate critical code
  – Revisit other code when it becomes critical

• Doesn’t require full annotation to be useful

How Incrementality Works

• How can one provide incremental benefit with mutual dependencies?

• Cut points
  – Method annotations
    • partition call graph
  – Can assure property of a subgraph
  – Assurance is contingent on accuracy of trusted method annotations

Call Graph of Program
How Incrementality Works 2

- How can one provide incremental benefit with mutual dependencies?
- Cut points
  - Method annotations partition call graph
  - Can assure property of a subgraph
  - Assurance is contingent on accuracy of trusted cut point method annotations

Cutpoint Example: @requiresLock

- Analysis normally assumes a method acquires and releases all the locks it needs.
  - Prevents caller’s correctness from depending on internals of called method.

- Method can require the caller to already hold a certain lock: @requiresLock FilterLock
  - Analysis of method gets to assume the lock is held.
    - Doesn’t need to know about caller(s).
  - Analysis of caller checks for lock acquisition.
    - Still ignores internals of called method.
Assuring Concurrency

• Annotations are turned into tables
  – `@lock` → Lock–state associations
  – `@requiresLock` → Method preconditions

• Lock analysis is syntax-directed
  – Single pass over the syntax tree
  – (Sample rules follow)

Caveat: Lock analysis is a consumer of other analyses—some of which are data flow analyses.

Assuring Field Access

Checking field access `e.fd`

• `P` — The program
• `E` — Current stack frame
• `C` — Currently held locks

\[ P; E; C ⊢ e : c \quad \langle c'.fd, t \rangle \in C \quad \text{lockFor}(P, e, c'.fd, e) \subseteq C \]

\[ P; E; C ⊢ e.fd : t \]
Synchronized Blocks

Expression $e_1$ could be any of the locks in set $L$.

Analyze the body of the block ($e_2$) holding the locks $L$.

\[
P; E; C \vdash_{\text{final}} e_1 : c \text{ as } L \quad P; E; (C \cup L) \vdash e_2 : t
\]

\[
P; E; C \vdash \text{synchronized}(e_1) \{e_2\} : t
\]

Semantically Rich Models

Expressing lock policy
- Object protects itself:
  \text{@lock BufLock is this protects Instance}
- Caller of method must acquire lock:
  \text{@requiresLock BufLock}

Aggregating state
- Only references to arrays are protected, not the arrays themselves
- Aggregate unaliased arrays:
  \text{@unshared}
  \text{@aggregate [] into Instance}

Constructors
- Cannot be \text{synchronized}.
- But most are single-threaded:
  \text{@singleThreaded}
  \text{@borrowed this}
Annotations, Analyses & Assurances

Lock Model Annotations
@lock
@requiresLock
@returnsLock
@singleThreaded

Policy Annotations
@selfProtected
@policyLock

Region Annotations
@region
@mapInto
@aggregate

Effect Annotations
@reads
@writes

Lock Analysis

Uniqueness Annotations
@unshared
@unique
@borrowed

Effects Analysis

MayEqual

Uniqueness Analysis

Lightening the Annotation Burden

Problem:
Class has 8 constructors; tens of methods. They all should have similar annotations. Annoying to repeat annotations.

```java
/**
 * @lock L is this protects Instance
 * @promise "@singleThreaded" for new(**)
 * @promise "@borrowed this"
 */
public class DateFormatManager {
    public DateFormatManager(TimeZone timeZone) {
        super();
        _timeZone = timeZone;
        configure();
    }
    ...
    private synchronized void configure() {...}
}
```

“All constructors are single threaded.”
“No method/constructor retains reference to the receiver.”
Missing Code

• “Missing code” is a problem
  – Libraries
  – Unwritten code
  – Code that is “not my problem”

• But assurance may be impossible without assuring missing code
  – E.g., will the called method create aliases?

Missing Code: Assumptions

• Allow local assumptions:
  – `@assume "... " for ...`
  – `Trusted locally` within their scope
  – `Create obligations` for the missing code
  – `Assurance is conditional` on their truth

"Red Dot" marks conditional assurances.
The Fluid Eclipse Plug-in

Eclipse IDE

Fluid Tool View
Fluid Summary: Towards Safer Code

Realities
• Code is the as-built reality
  – But, we don’t understand code
  – Non-local properties are (often) known but not expressed
  – Thus, loss of intellectual control

• Models are necessary
  – Code and design evolve separately
  – We assure consistency

• Adoption barriers exist for present semantic assurance techniques

The Fluid approach
• Incrementality
  – Capture & express critical properties
  – New ways to model and express diverse mechanical properties
  – Create assurance: chains of evidence
    • Couple models/annotations, analysis
    • Are we in the framework? Are we compliant with the API?
  – Build semantic links between code and design
  – Integrate into programmer practice
    • Build on existing practice (e.g., open source, Eclipse, etc.)
      • Seek invisible or incremental interventions
      • Instant gratification principle

http://www.fluid.cs.cmu.edu