

Analysis for Safe Concurrency

Reading: ***Assuring and Evolving Concurrent Programs: Annotations and Policy***

17-654/17-754: Analysis of Software Artifacts

Jonathan Aldrich



Example: java.util.logging.Logger

[Source: Aaron
Greenhouse]



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

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

```
}
```

Consider `setFilter()` in isolation

Example: java.util.logging.Logger

[Source: Aaron
Greenhouse]



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

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

Consider `log()` in isolation

Example: java.util.logging.Logger

[Source: Aaron
Greenhouse]



```
/** ... 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 class `Logger` in it's entirety!

Example: java.util.logging.Logger

[Source: Aaron
Greenhouse]



```
/** ... 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;
        } ...
    } ...
}
```

1

2

3

Class Logger has a *race condition*.

Example: java.util.logging.Logger

[Source: Aaron
Greenhouse]



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

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

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

Correction: synchronize setFilter()



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`

- Not all race conditions result in errors
- Error results when invariant is violated
 - `Logger` invariant
 - filter is not null at call following null test
 - Race-related error
 - race between write and dereference of filter
 - if the write wins the race, filter is null at the call



Example: Summary 4

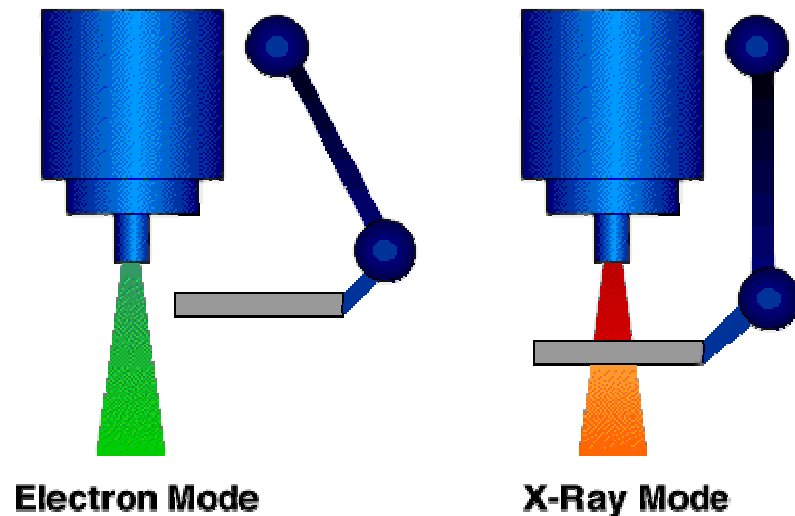
Problem: Race condition in class `Logger`

- 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`

Software Disasters: Therac-25



- Delivered radiation treatment
- 2 modes
 - Electron: low power electrons
 - X-Ray: high power electrons converted to x-rays with shield
- Race condition
 - Operator specifies x-ray, then quickly corrects to electron mode
 - Dosage process doesn't see the update, delivers x-ray dose
 - Mode process sees update, removes shield
- Consequences
 - 3 deaths, 3 serious injuries from radiation overdose



from <http://www.netcomp.monash.edu.au/cpe9001/assets/readings/HumanErrorTalk6.gif>

source: Leveson and Turner, An Investigation of the Therac-25 Accidents, *IEEE Computer*, Vol. 26, No. 7, July 1993.

Fluid: Tool Support for Safe Concurrency



Models are Missing

[Source: Aaron
Greenhouse]



- **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.”
 - Annotations in tools like 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.”
 - Tool design (e.g. Fluid): Payoffs early and often
 - Direct programmer utility — **negative marginal cost**
 - Increments of payoff for increments of effort

Capturing Design Intent

[Source: Aaron
Greenhouse]



- *What data is shared by multiple threads?*
- *What locks are used to protect it?*
 - Annotate class: `@lock FL is this protects filter`

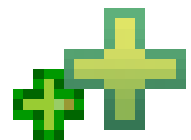
Reporting Code–Model Consistency

[Source: Aaron
Greenhouse]

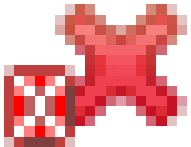


- Tool analyzes consistency
 - No annotations \Rightarrow no assurance
 - Identify likely model sites

- Three classes of results



Code–model consistency



Code–model inconsistency



Informative — Request for annotation

Fluid Demonstration: Locks



Incremental Assurance

[Source: Aaron
Greenhouse]



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



Analysis Issues: Aliasing

- Other pointers can invalidate reasoning
 - `@singlethreaded` – can other threads access through an alias?
 - `@aggregate ... into Instance` – can the field be accessed through an alias that is not protected by the lock?
- Similar issues in other analyses, e.g. `Typestate`

```
FileInputStream a = ...  
FileInputStream b = ...  
a.close()           // what if a and b alias?  
b.read(...)        // may read a closed file
```

- Solution from Fugue (Microsoft Research)
 - `@NotAliased` annotation indicates that b has no aliases
 - Therefore closing a does not affect b
 - Requires alias analysis to verify
 - Can sometimes be inferred by analysis
 - e.g. see Fink et al., ISSTA '06

Capturing Design Intent

[Source: Aaron
Greenhouse]



- *What data is shared by multiple threads?*
- *What locks are used to protect it?*
 - Annotate class: `@lock FL is this protects filter`
- *Is this delegate object owned by its referring object?*
 - Annotate field: `@aggregate ... into Instance`
- *Can this object be accessed by multiple threads?*
 - Annotate method: `@singleThreaded`
- *Can this argument escape to the heap?*
 - Annotate method: `@borrowed this`

Analysis Issues: Constructors, Inheritance



- Constructors
 - Often special cases for assurance
 - Fluid: can't protect with "this" lock
 - But OK since usually not multithreaded yet
 - Others
 - Invariants may not hold until end of constructor
- Subtyping
 - Subclass must inherit specification of superclass
 - Example: `@singlethreaded` for `Formatter`
 - Sometimes subclass extends specification
 - e.g. to be multi-threaded safe
 - requires care in inheriting or overriding superclass methods
- Inheritance
 - Representation of superclass may have different invariants than subclass
 - super calls must obey superclass specs
 - e.g. call to `Formatter` constructor

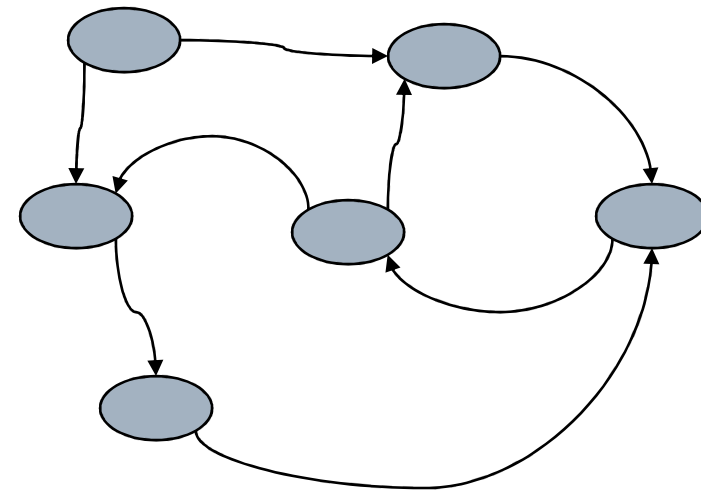
How Incrementality Works 1

[Source: Aaron
Greenhouse]



- How can one provide incremental benefit with mutual dependencies?

Call Graph of Program

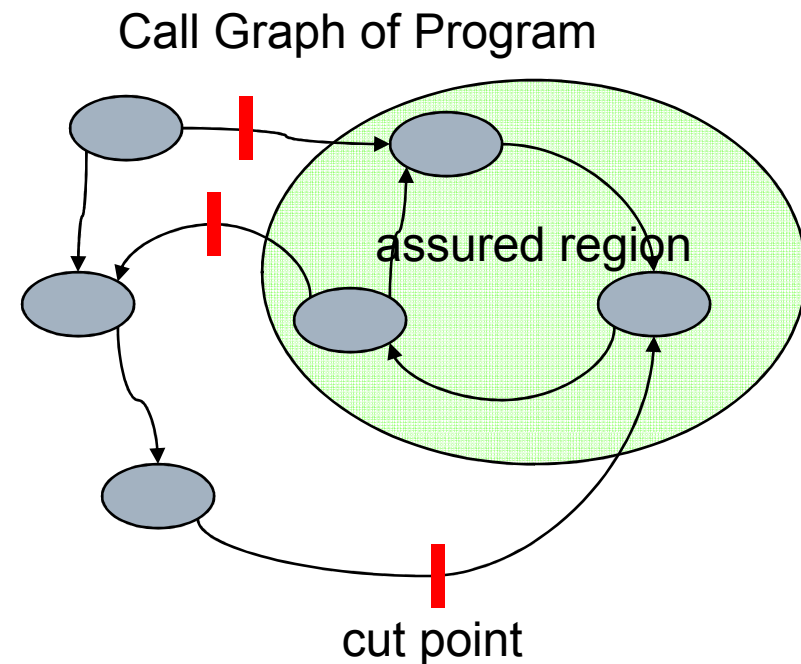


How Incrementality Works 2

[Source: Aaron
Greenhouse]



- 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` [Source: Aaron Greenhouse]



- 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.



Capturing Design Intent

[Source: Aaron
Greenhouse]



- *What data is shared by multiple threads?*
- *What locks are used to protect it?*
 - Annotate class: `@lock FL is this protects filter`
- *Is this delegate object owned by its referring object?*
 - Annotate field: `@aggregate ... into Instance`
- *Whose responsibility is it to acquire the lock?*
 - Annotate method: `@requiresLock FL`



Concurrency: Summary

- Many ways to make concurrency safe
 - Single-threaded data
 - Locks
 - Disabled interrupts
 - Analysis of interleavings (simple settings)
 - Transactions (future)
- Design intent useful
 - Document assumptions for team
 - Aids in manual analysis
 - Enables (eventual) automated analysis

Questions?

