A Programmer-Oriented Approach to
Software Assurance and Evolution

Bill Scherlis, as told by Dean Sutherland
CMU School of Computer Science
scherlis@cmu.edu
412-268-8741

The Fluid Project
www.fluid.cs.cmu.edu

Jan 05

Carnegie Mellon

Fluid — Fluid dependability attributes (*examples*)

- Engineering properties for safety, dependability, security
  - **Safe concurrency**
    - Race conditions
    - Lock management
    - Single thread concurrency control
    - Lock ordering and deadlocks
  - **Code safety**
    - Ignored exceptions
    - Appropriate typing
  - **Policy compliance**
    - API policy compliance
    - Framework compliance
    - Object references and aliasing
    - Patterns, uses, structure
  - **Real time**
    - Real-time thread/memory policies

- Hard to **test**
  - Nondeterminism
- Hard to **inspect**
  - Non-local
  - Model-based

Carnegie Mellon

Fluid — Direct measures

We treat our software as if it were a phenomenon of nature

— Sir Tony Hoare, 2004

Indirect Measures
- Process
- People
- Bug counts
- KLOC counts

➡

Direct Measures
- Model coverage
  - By attribute kind
  - By code coverage
- Code/model consistency

Carnegie Mellon

Fluid — IT supply chain barriers

Interface barriers exist between producers and consumers at all stages of IT supply chains

Five barriers
- Contractor qualification
- Requirements definition
- "Second" sourcing
- Risk allocation
- Engineering acceptance

➡

Mitigation (today's best)
- CMM / CMMI
- Close relationships
- API conventionalization
- Asymmetry
- **Testing and inspection**

Producers:
Internal development groups
Subcontractors
Outsources
Offshore
Off-the-shelf
Open Source

Carnegie Mellon

Fluid — Quality stakeholders

At each supply chain interface:

- Developers
  - Immediate code guidance
  - Basis for dependability claims
  - Incremental progress
- Managers
  - Direct evidence / measurement
  - Design intent capture
- CIO organization
  - Standards (e.g., framework enforcement)
  - Organizational memory
- Acceptance evaluators
  - Proxy elimination
  - Direct artifact evaluation

Carnegie Mellon

Fluid — Software and code

Code
- The ground truth of software
  - We create it, but we do not understand it

Challenges
- Poor quality measures
  - Weak proxies: People, process, bug counts, KLOC
  - Impact: Difficulty of ROI case
- Design intent is missing
  - Code embodies insufficient information about itself
  - Huge information loss
- There is no escape
  - Generation and abstraction: program at higher level

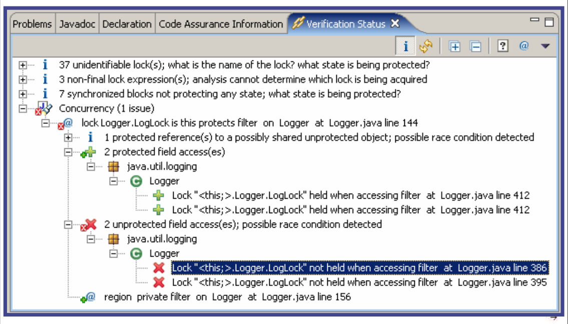
Carnegie Mellon

Fluid — The Fluid Project

- Create and maintain safe, dependable, secure code
  - Directly assure critical **dependability** attributes
    - Attributes tend to defy testing and inspection
      - {Dependability, safety, security}
    - Direct static assurance
  - Express dependability-related **models**
    - Incrementally capture design intent
  - Provide **direct assurance** and **measurement**
    1. Inventory of fault-relevant sites
    2. Modeling progress
    3. Analysis progress: assurance, potential faults
  - **Adoptability** and **scalability** are paramount
    - Ease of use by practicing developers
    - Management value – metrics and process support
    - Composability and components
    - Incrementality and early rewards
    - Partiality and contingency

Carnegie Mellon

Fluid — The Fluid Eclipse Plug-in



Carnegie Mellon

Fluid — Reporting Code–Model Consistency

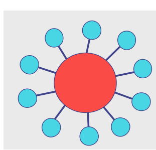
Tool analyzes model/code consistency
- No model ⇒ no assurance
- Identify likely model sites

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

Carnegie Mellon

Fluid — Assured Development: Hub and spokes

- Hub – Fluid core infrastructure
  - Representations, core analyses, etc.
  - Interactive online, build-based offline
  - Verification support
    - Proof management, Assertion propagation
    - Permissions
    - Effects, aliasing, regions
- Spokes – attribute-specific analyses (*examples*):
  - Assurance:
    - Races (lock)
    - Races (non-lock)
    - Modular non-lock
    - Real time
  - Indicators
    - Appropriate typing
    - Exceptions ignored
    - Concurrency finder
    - Thread effects



Carnegie Mellon

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

Reporting Assurance Results

Carnegie Mellon — Fluid

Assurance results
- Model – programmer provided design intent
- Assured – design intent is consistent with code
- Not Assured – design intent is inconsistent with code
  - Relative to design intent

Inferred results
- Possible problems, next steps, reasonable defaults

Metric results (recent work)

How much have I done?
- Model building
- Assurance development

Assurance locator
- Identifies *where* models and assurance *exist* within the system's structure
- *Incrementality* allows assurance of focused "islands" within a large software system
  - Cut points allow programmer selected modularization of assurance efforts

7	1	29	org.apache.log4j
		14	org.apache.log4j.chainsaw
			org.apache.log4j.config
1	68		org.apache.log4j.helpers

Fluid Tool Capabilities (for Java)

Carnegie Mellon — Fluid

- Lock-based concurrency
  - Region model
- Non-lock concurrency
  - Color model
- Real-time thread policy compliance
  - Color model
- Code quality analysis
  - Appropriate types
  - Ignored exceptions
- Facets of API compliance

Apache Log4j BoundedFIFO: Model semantics

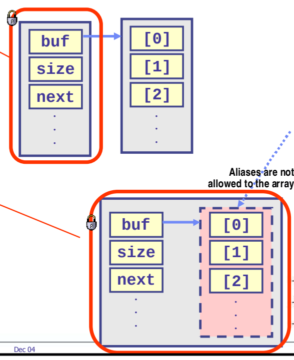
Carnegie Mellon — Fluid

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

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

Constructors
- Cannot be `synchronized.`
- But most are single-threaded:
  @singleThreaded
  @borrowed this

Verification and assurance
- Access to shared data
- Correct lock used
- Lock held by callers
- Unshared access



Races and security

Carnegie Mellon — Fluid

Examples of security-related race conditions:

- 15-11-2003: monopd Race Condition Denial of Service Vulnerability
- 15-10-2003: Sun Solaris Pipe Function Unspecified Kernel Race Condition 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 execve() System Call Race Condition Vulnerability
- 29-04-2003: Worker Filemanager Directory Creation Race Condition Vulnerability
- 23-04-2003: SAP Database SDBINST 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
- 16-05-2002: SuSE AAA_Base_Clean_Core Script RM Race Condition Vulnerability
- 09-05-2002: Multiple Vendor exec C Library Standard I/O File Descriptor Race Condition Vulnerability
- 11-03-2002: GNU Fileutils Directory Removal Race Condition Vulnerability
- 27-02-2002: FSLint Temporary File Race Condition Vulnerability
- 06-02-2002: FreeBSD FStatFS Syscall Race Condition Vulnerability
- 30-01-2002: Compaq Tru64 Kernel Race Condition Vulnerability
- 26-01-2002: Tarantella Enterprise 3 gunzip Race Condition Vulnerability
- 16-01-2002: BSD exec() Race Condition Vulnerability
- 05-12-2001: XTel XTel-User Temporary File Race Condition Vulnerability
- 20-11-2001: IBM AIX Bellmail Race Condition Vulnerability
- 17-08-2001: Multiple BSD FTS Directory Traversal Race Condition Vulnerability

(Source: Bugtraq vulnerabilities list)

Fluid: published results

Carnegie Mellon — Fluid

Annotation, analysis, and tool publications

- POPL '05
- CSJP '04
- OOPSLA '03 Eclipse Tech eXchange
- Greenhouse thesis '03
- PASTE '02
- ICSE '02
- *Software—Practice and Experience* '01
- ECOOP '99
- ICSE '98

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


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

Model intent that all constructors are single threaded. Model intent that no method retains reference to the receiver.

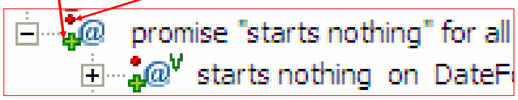
Now the locking model can be assured (deeply)…as the tool displays

lock DateFormatManager.DateFormatManager is this protects Instance
29 protected field access(es)
region public Instance on Object

Carnegie Mellon

Fluid — The "red dot" – exploit partial results

Consistency of model and code is contingent on a "trusted" result



Carnegie Mellon

Fluid — Fluid summary: towards safer code

Realities

- **Code is the as-built reality**
  - Nonetheless, we don't understand code
  - Non-local properties are (often) known but not expressed
  - Loss of intellectual control
- **Models are necessary**
  - Code and design evolve separately
  - We assure consistency
- **Adoption barriers exist for present semantic assurance techniques**

Our approach

- **Incrementality**
  - Capture and 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
    - Accept coding constraint to facilitate this
- **Integrate directly into programmer practice**
  - Build on existing practice (e.g., open source, Eclipse, *etc.*)
    - Seek invisible or incremental interventions
    - Instant gratification principle

Carnegie Mellon

Fluid — Case Study - Scope

- The assurance evaluation we are presently offering for case study purposes focuses on **race conditions**, including both lock-based and non-lock concurrency.
- **Questions**
  - What are the sizes and complexity of the candidate systems and the major subsystems and components of interest?
  - What are your most challenging concurrency-related assurance issues? Where is the greatest complexity of threading and locks? Is there significant exploitation of thread-locality or time-sharing of state?
  - Are there known races and other anomalies?
- **Focus of effort**
  - We prefer to work on the **most challenging** concurrency issues in your code, where you are having the most vexing and costly problems
  - We expect to provide some immediate improvement in the overall quality of your software system. All design intent annotation will remain after we leave.
  - CMU values the experience gained from exercising the FTT technology in a live, production environment.

Carnegie Mellon

Fluid — Case Study - Agenda

- Day 1
  - We work together in a room with a digital projector, though we will likely break into 1-3 person teams after the initial session.
  - Morning -- Meet and greet
    - *Fluid team*: Tool intro
    - *Host team*: Software system overview and issues
  - Afternoon: Load tool with the code base and do a local build.
    - Start analysis
    - Obtain preliminary results
- Day 2
  - Tool use by both teams and collaboration
  - Mid-way assessment
- Day 3
  - Tool use by both teams and collaboration
  - Assessment
  - Outbrief of overall results and discussion

Carnegie Mellon

Fluid — Case Study - Staffing

FTT Team

- The team includes technical principals who have considerable experience in applying the tool in production settings.
- They are experts in program analysis, Java concurrency, and model/code management for larger systems.
- Our team are all CMU researchers and US citizens.
- We expect to either execute a suitable bilateral NDA or work under informal NDA.

Host Team

- Ideally, we collaborate with developers in identifying (reverse engineering, in some cases) concurrency-related design intent.
- It is therefore important to us to have access to individuals with whom we can address technical questions as modeling and analysis proceed.

Carnegie Mellon

Fluid — Case Study - Preparation

- Advance preparation
  - Informal presentation/discussion regarding concurrency patterns and potential issues in the code base of interest.
  - Additionally, architectural overview information would be helpful.
  - We prefer to bring our own laptops which already have the tools installed. (We have done this at highly secure sites.)
    - We will load/unload code under host supervision.
    - If this is not possible, we will need to have access to high-performance Windows computers with 2GB RAM
    - Our tool is presently based in Eclipse