Analysis-based verification:

A programmer-oriented approach to the assurance of mechanical program properties

Tim Halloran

HCSS 6 May 2011

tim.halloran@surelogic.com
SureLogic, Inc.

Overview

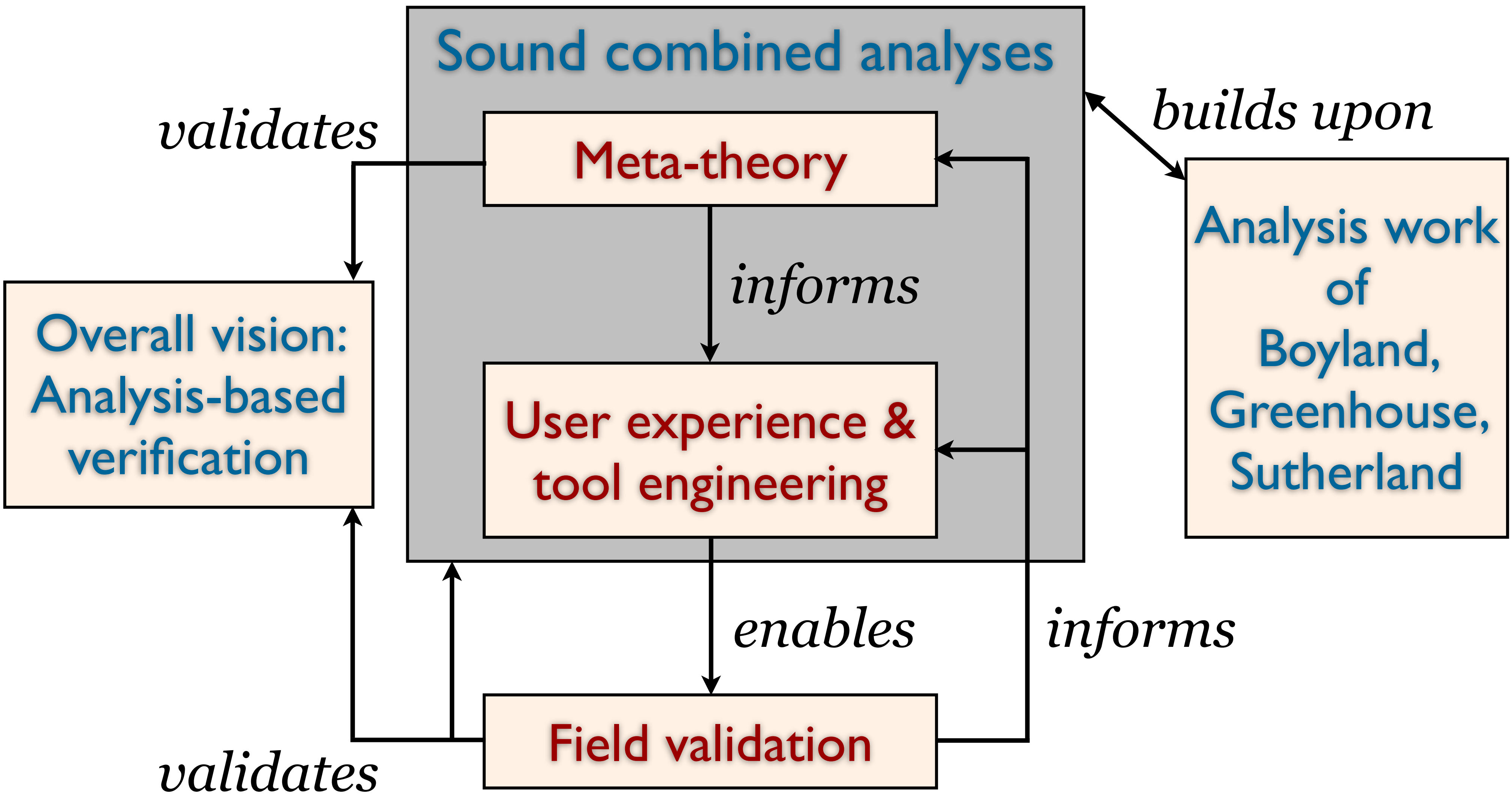
- ***Vision***: Create focused *analysis-based verification* for software quality attributes[1] as a scalable[2] and adoptable[3] approach to verifying[4] consistency of code with its design intent[5]

1. Quality attributes: Including safe concurrency with locks, data confinement to thread roles, static layer structure, many others
    - Each has its combination of contributing constituent analyses — e.g., effects upper bounds, mostly-unique references, and binding context
2. Scalable: Significantly adapt constituent analyses to enable composition
3. Adoptable: Before-lunch test (incremental reward principle)
4. Verification: No false negatives from analysis targeted to an attribute and a model
5. Design intent: Fragmentary models/specifications focused on quality attributes

Overview

- The focus of this talk is concept of *sound combined analyses*, an enabling component of analysis-based verification, including
  - **Meta-theory** to establish soundness of the approach of combining multiple constituent static analyses into an aggregate developer-focused analysis
    - Reminiscent, with respect to goals — not particulars, of Nelson-Oppen cooperating decision procedures
  - **User experience** and **tool engineering** approach designed to address adoption and usability criteria of professional development teams
    - Developer ROI, including before-lunch test
  - **Field validation** in collaboration with professional engineers on diverse commercial and open source code bases
    - Deployed major systems including vendor application server code, library and framework code, and MapReduce infrastructure

Overview

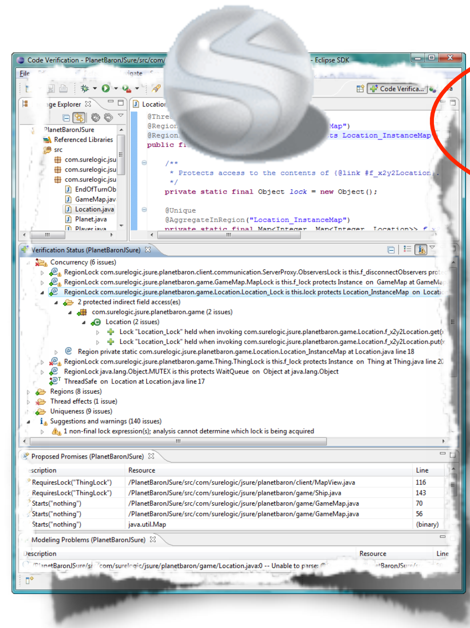


Sound combined analyses
Meta-theory
informs
User experience & tool engineering
builds upon
Analysis work of Boyland, Greenhouse, Sutherland
validates
Overall vision: Analysis-based verification
enables
informs
Field validation
validates

This work in context

Carnegie Mellon

isr institute for SOFTWARE RESEARCH



JSure

An analysis-based verification tool

Flashlight

A concurrency-focused dynamic analysis tool

Sierra

A platform for the management of results from multiple heuristic-based static analysis tools

surelogic®

Fluid Research Project

Scherlis, Boyland, Chan, Greenhouse, Halloran, Sutherland

→

Commercialization

Java* Analysis Capability

Prototype Tools, Technology, People

Related work

- Fluid project at Carnegie Mellon – sound static analysis, promises
  - Scherlis, Boyland, Greenhouse, Chan (formative)
  - Sutherland (evaluative)
- Heuristics-based static analysis tools
  - FindBugs [Hovemeyer, Pugh]; MC [Engler, Chelf, Chou, *et al.*]
- Spec# – practicable verification of real-world code
  - *Specification*: preconditions, postconditions, invariants
  - *Tool verification*: Boogie verifier
  - Microsoft Research [Leino, Barnett, *et al.*]
  - Builds upon the work ESC/Modula and ESC/Java (Larch)
- JML [Leavens, *et al.*]
  - Verifiers: LOOP (PVS), KeY, Jive – automation, language subset
- Languages that support specification
  - SPARC Ada – up front commitment

Outline

- Design intent and heuristics-based static analysis tools
- What is analysis-based verification?
- Sound combined analyses
  - Supporting verification
  - An aside on the meta-theory
  - Supporting model expression
  - Supporting contingencies
- Evaluation
  - Field trials
- JSure Modeling Language
- Summary

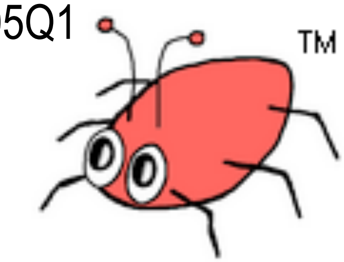
Heuristics-based static analysis tools

- Examples: FindBugs, PMD, MC (lots more...)
- Scan code and report violations of “bug patterns”
- Successful finding defects in real-world code

```
/**
 * java.lang.annotation.AnnotationTypeMismatchException
 * @author Josh Bloch
 */
private final String foundType;
public String foundType() {
  return this.foundtype();
}
```

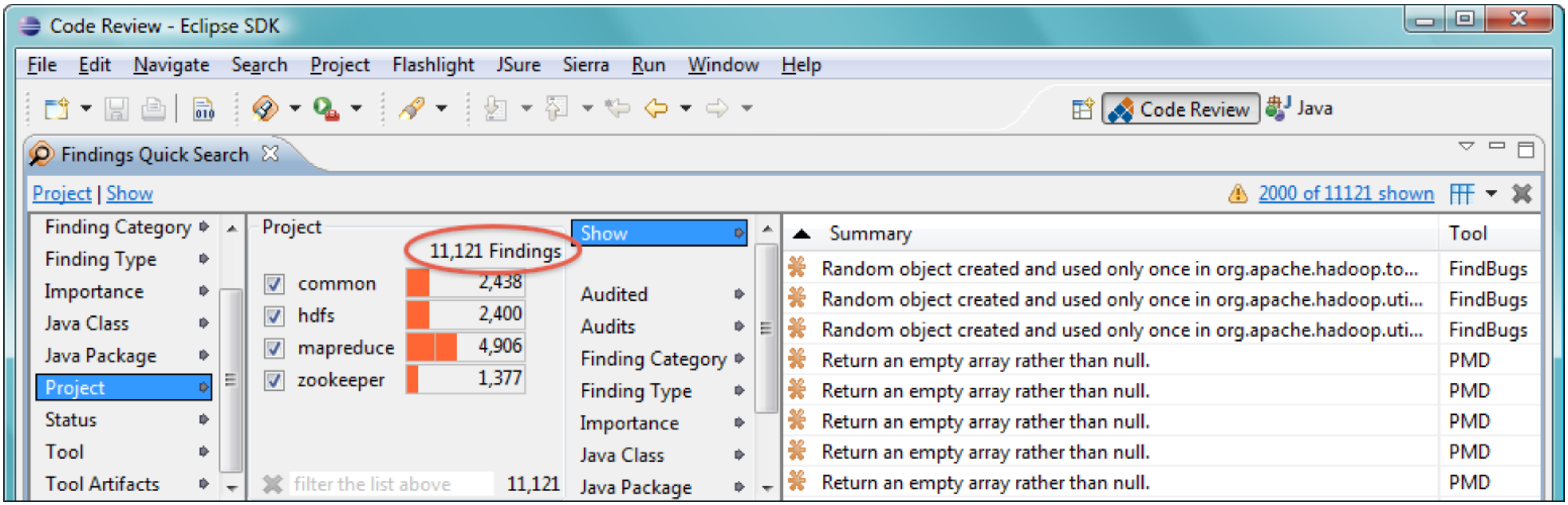
How many infinite recursive loops can FindBugs find in your code?

5 Sun’s JDK 1.5.0_01
10 Sun’s AppServer 8.1 2005Q1
14 IBM’s WebSphere 6.0.2
13 JBoss 4.0.2
3 Eclipse 3.1M7
2 Tomcat 5.5.9



Everyone makes stupid mistakes. What do you use to help you find and fix yours?

False positives and false negatives



		Actuality	
		Fault	*No fault*
Tool says	*Fault*	True positive	**False positive**
	No fault	**False negative**	True negative

Intent sharpens heuristic analysis

```
public @NonNull String convert(@NonNull Object o) {
   return o.toString();
}
```

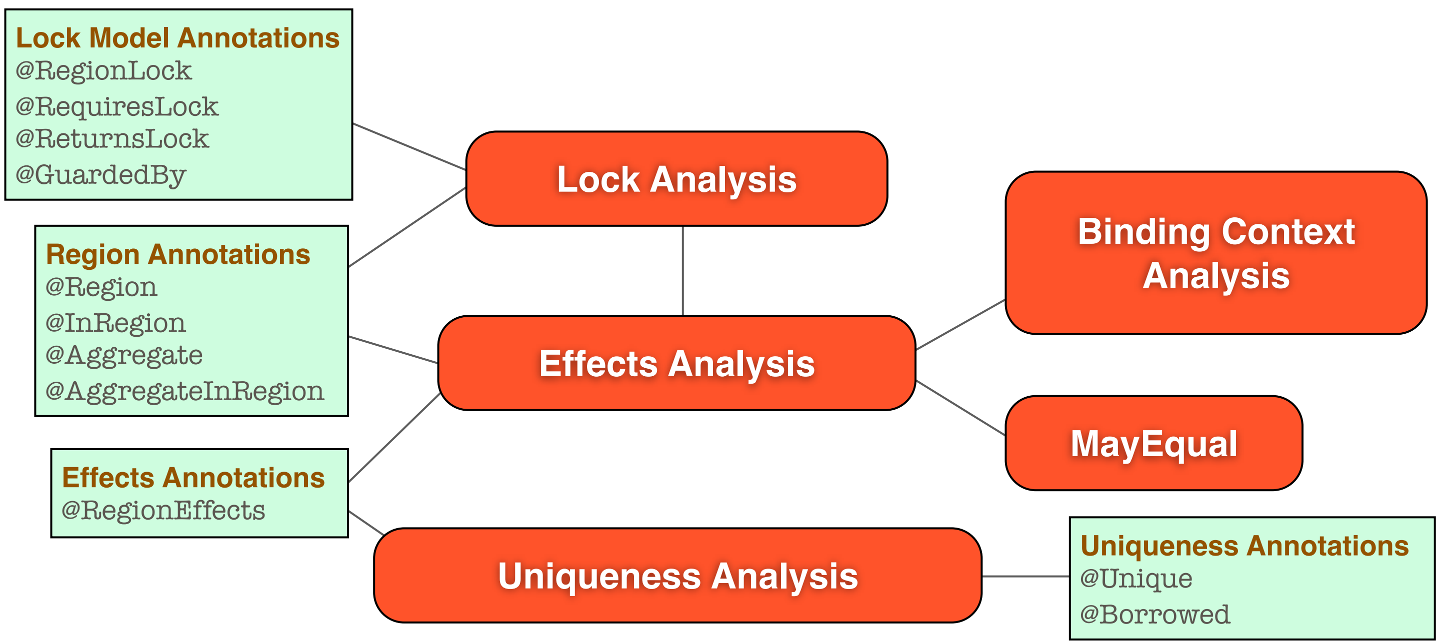
- Why? To reduce false positive results
  - *"The static analysis crowd jokes that too high a percentage of false positives leads to 100% false negatives because that's what you get when people stop using the tool."* [Chess, McGraw]
- Best result: "I didn't find anything wrong"
  - Does not answer the question: "Is this design intent fully consistent with my code?"
  - That is, there may be something wrong that it didn't find

Answerable by verification: classical theorem proving, sound static analysis, etc.

What is analysis-based verification?

- Tool-supported verification, based upon sound static analysis
- Prior work developed annotations and a set of verifying analyses
  - Boyland: Uniqueness, effects
  - Greenhouse: Lock use policy, effects
  - Sutherland: Thread use policy

Work done by the Fluid research group at Carnegie Mellon



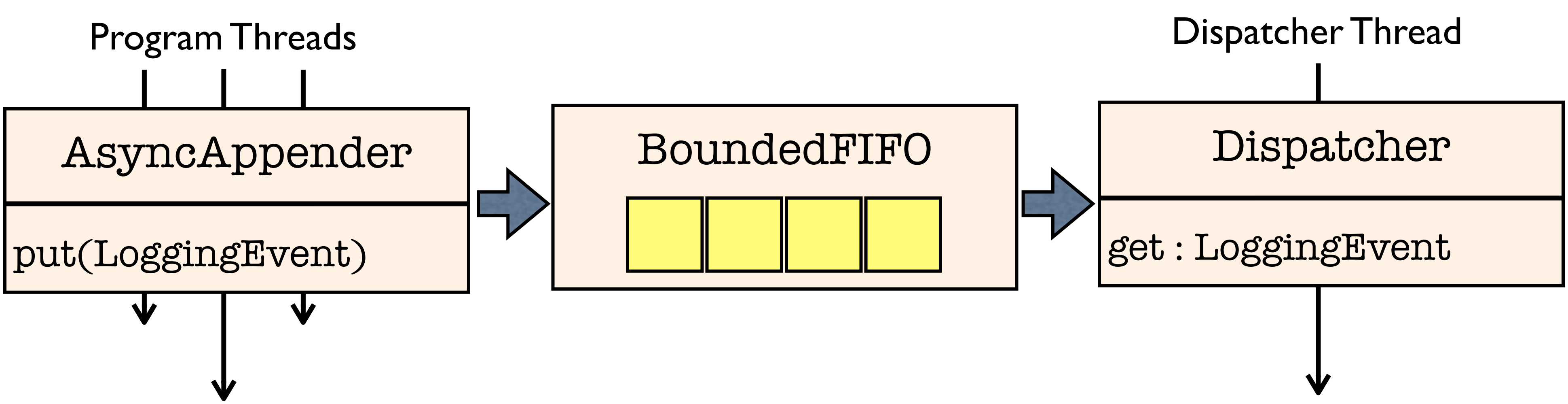
Sound combined analyses

- Creates verification results by combining analysis results
  - Multiple constituent program analyses ("plug-in")
  - Analyses report fragmentary results
- Verification results are always with respect to some some specification — usually narrowly focused with respect to attribute and code region
- What do we mean by *sound*?
  - For program analyses: Sound (also called conservative) means no false negatives. A judgement of inconsistency may mean "not sure" [Rice]
  - For our approach: Sound means results derivable our proof calculus are 'consistent' in a semantics of the analysis results
    - Demonstrated by proof in Halloran's dissertation (Chapter 2)

We introduce our approach via a "tour" of its features

A running example

- BoundedFIFO from Apache Log4j
  - The program enqueues an event into the buffer and returns
  - A dispatcher thread removes events from the buffer and processes them (as events become available)
  - Not exemplary Java — but typical of (some) code we encountered in the field
  - Annotations reflect the use of the class within Log4j



Annotated code

```
@RegionLock("FIFOLock is this protects Instance")
public class BoundedFIFO {
  @Unique
  @Aggregate
  LoggingEvent[] buf;

  int numElts = 0, first = 0, next = 0, size;

  @Unique("return") public BoundedFIFO(int size) { ... }

  @RequiresLock("FIFOLock") public LoggingEvent get() { ... }
  @RequiresLock("FIFOLock") public void put(LoggingEvent o) { ... }
  @RequiresLock("FIFOLock") public int getMaxSize() { ... }
  @RequiresLock("FIFOLock") public int length() { ... }
  @RequiresLock("FIFOLock") public boolean wasEmpty() { ... }
  @RequiresLock("FIFOLock") public boolean wasFull() { ... }
  @RequiresLock("FIFOLock") public boolean isFull() { ... }
}
```

Supporting verification

- Prior approach: "compiler-like" output →
  - Analyses report:
    - "Points of consistency" +
    - "Points of inconsistency" ×
- Limitations:
  - Relationships among promises are lost
    - Impact of "X" on consistency of other promises difficult to understand
  - Fails to answer the question, "Is my model consistent with the code?"

Lock Policy Analysis Results for `BoundedFIFO`

	Finding	About	Description
f_1	+	r_1	thread-confined access to `numElts` at line 8
f_2	+	r_1	thread-confined access to `first` at line 8
f_3	+	r_1	thread-confined access to `next` at line 8
f_4	+	r_1	thread-confined access to `size` at line 13
f_5	+	r_1	thread-confined access to `buf` at line 14
f_6	+	r_1	`FIFOLock` held for access to `numElts` at line 19
f_7	+	r_1	`FIFOLock` held for access to `buf` at line 20
f_8	+	r_1	`FIFOLock` held for access to `first` at line 20
f_9	+	r_1	`FIFOLock` held for access to `buf[first]` at line 20
f_{10}	+	r_1	`FIFOLock` held for access to `first` at line 21
f_{11}	+	r_1	`FIFOLock` held for access to `size` at line 21
f_{12}	+	r_1	`FIFOLock` held for access to `first` at line 21
f_{13}	+	r_1	`FIFOLock` held for access to `numElts` at line 22
f_{14}	+	r_1	`FIFOLock` held for access to `numElts` at line 28
f_{15}	+	r_1	`FIFOLock` held for access to `size` at line 28
f_{16}	+	r_1	`FIFOLock` held for access to `buf` at line 29
f_{17}	+	r_1	`FIFOLock` held for access to `next` at line 29
f_{18}	+	r_1	`FIFOLock` held for access to `buf[next]` at line 29
f_{19}	+	r_1	`FIFOLock` held for access to `next` at line 30
f_{20}	+	r_1	`FIFOLock` held for access to `size` at line 30
f_{21}	+	r_1	`FIFOLock` held for access to `next` at line 30
f_{22}	+	r_1	`FIFOLock` held for access to `numElts` at line 31
f_{23}	+	r_1	`FIFOLock` held for access to `size` at line 36
f_{24}	+	r_1	`FIFOLock` held for access to `numElts` at line 39
f_{25}	+	r_1	`FIFOLock` held for access to `numElts` at line 42
f_{26}	+	r_1	`FIFOLock` held for access to `numElts` at line 45
f_{27}	+	r_1	`FIFOLock` held for access to `size` at line 45
f_{28}	+	r_1	`FIFOLock` held for access to `numElts` at line 48
f_{29}	+	r_1	`FIFOLock` held for access to `size` at line 48

Uniqueness Analysis Results for `BoundedFIFO`

	Finding	About	Description
f_{30}	+	r_6	reference held by `buf` is unique (*i.e.*, unaliased)
f_{31}	+	r_{10}	constructor does not alias `this`
f_{32}	+	r_{10}	`super()` promises not to alias `this`

Lock Policy Analysis Results for `Dispatcher`

	Finding	About	Description
f_{33}	×	r_{38}	`FIFOLock` not held when invoking `length()` at line 61
f_{34}	×	r_{17}	`FIFOLock` not held when invoking `get()` at line 66
f_{35}	×	r_{44}	`FIFOLock` not held when invoking `wasFull()` at line 67

Issue of scale: 2,146 analysis results on our first field trial, ~12K analysis results on Electric)

Lost relationships among promises

```
1 @RegionLock("FIFOLock is this protects Instance")
2 public class BoundedFIFO {

10   @Unique("return")
11   public BoundedFIFO(int size) {
12     if (size < 1) throw new IllegalArgumentException();
13     this.size = size;
14     buf = new LoggingEvent[size];
15   }

38   @RequiresLock("FIFOLock")
39   public int length() { return numElts; }
```

Lock Policy Analysis Results for `BoundedFIFO`

	Finding	About	Description
f_{24}	+	r_1	`FIFOLock` held for access to `numElts` at line 39

- The `length()` method lock is not synchronized on `this`, so f_{24} "trusts" the `@RequiresLock("FIFOLock")` promise at line 38

Unknown impact of an "X" result

```
50 public class Dispatcher {

58   private LoggingEvent get() {
59     synchronized (this) { // Broken – acquires the wrong lock
60       LoggingEvent e;
61       while (fifo.length() == 0) {
     ...
```

```
1  @RegionLock("FIFOLock is this protects Instance")
2  public class BoundedFIFO {

38   @RequiresLock("FIFOLock")
39   public int length() { return numElts; }
```

f_{24}

Lock Policy Analysis Results for `Dispatcher`

	Finding	About	Description
f_{33}	×	r_{38}	`FIFOLock` not held when invoking `length()` at line 61

- The `get()` method in the `Dispatcher` class acquires the wrong lock at line 59, so f_{38} reports that `FIFOLock` is not held when invoking `fifo.length()` at line 61
- What is the impact of this inconsistent result? r_1 is not verifiable!

Not answering the right question

```
@RegionLock("FIFOLock is this protects Instance")
public class BoundedFIFO {
```

- Are the annotations consistent with the code?
  - All the analysis results about r_1 are all consistent
  - BUT the use of the "wrong lock" in the `get()` method in the `Dispatcher` class causes r_1 to be indirectly inconsistent

Lock Policy Analysis Results for `BoundedFIFO`

	Finding	About	Description
f_1	+	r_1	thread-confined access to `numElts` at line 8
f_2	+	r_1	thread-confined access to `first` at line 8
f_3	+	r_1	thread-confined access to `next` at line 8
f_4	+	r_1	thread-confined access to `size` at line 13
f_5	+	r_1	thread-confined access to `buf` at line 14
f_6	+	r_1	`FIFOLock` held for access to `numElts` at line 19
f_7	+	r_1	`FIFOLock` held for access to `buf` at line 20
f_8	+	r_1	`FIFOLock` held for access to `first` at line 20
f_9	+	r_1	`FIFOLock` held for access to `buf[first]` at line 20
f_{10}	+	r_1	`FIFOLock` held for access to `first` at line 21
f_{11}	+	r_1	`FIFOLock` held for access to `size` at line 21
f_{12}	+	r_1	`FIFOLock` held for access to `first` at line 21
f_{13}	+	r_1	`FIFOLock` held for access to `numElts` at line 22
f_{14}	+	r_1	`FIFOLock` held for access to `numElts` at line 28
f_{15}	+	r_1	`FIFOLock` held for access to `size` at line 28
f_{16}	+	r_1	`FIFOLock` held for access to `buf` at line 29
f_{17}	+	r_1	`FIFOLock` held for access to `next` at line 29
f_{18}	+	r_1	`FIFOLock` held for access to `buf[next]` at line 29
f_{19}	+	r_1	`FIFOLock` held for access to `next` at line 30
f_{20}	+	r_1	`FIFOLock` held for access to `size` at line 30
f_{21}	+	r_1	`FIFOLock` held for access to `next` at line 30
f_{22}	+	r_1	`FIFOLock` held for access to `numElts` at line 31
f_{23}	+	r_1	`FIFOLock` held for access to `size` at line 36
f_{24}	+	r_1	`FIFOLock` held for access to `numElts` at line 39
f_{25}	+	r_1	`FIFOLock` held for access to `numElts` at line 42
f_{26}	+	r_1	`FIFOLock` held for access to `numElts` at line 45
f_{27}	+	r_1	`FIFOLock` held for access to `size` at line 45
f_{28}	+	r_1	`FIFOLock` held for access to `numElts` at line 48
f_{29}	+	r_1	`FIFOLock` held for access to `size` at line 48

Uniqueness Analysis Results for `BoundedFIFO`

	Finding	About	Description
f_{30}	+	r_6	reference held by `buf` is unique (*i.e.*, unaliased)
f_{31}	+	r_{10}	constructor does not alias `this`
f_{32}	+	r_{10}	`super()` promises not to alias `this`

Lock Policy Analysis Results for `Dispatcher`

	Finding	About	Description
f_{33}	×	r_{38}	`FIFOLock` not held when invoking `length()` at line 61
f_{34}	×	r_{17}	`FIFOLock` not held when invoking `get()` at line 66
f_{35}	×	r_{44}	`FIFOLock` not held when invoking `wasFull()` at line 67

To be "safe" the programmer has to fix all inconsistent analysis results

Overcoming these limitations

- The *drop-sea* proof management system
  - What is proof management?
    - The manipulation of formal proofs and proof fragments (lemmas) as data structures
  - Separation of overall *proof mgmt* from *constituent analyses*
    - *Proof mgmt*: combining fragmentary results, abductive inference (proposed promises), contingency management (red dot), truth maintenance (incremental recomputation)
      - Independent of language semantics!
    - *Analyses*: embody aspects of programming language semantics, creating a plug-in model (cf. Nelson-Oppen)
  - Challenges
    - Scale-up to very large proofs
    - Usability and visualization/debuggability
    - Enabling composition w.r.t. multiple underlying analyses, multiple components being "composed," and new bits of design intent being added (expanding the scope of consideration w.r.t. models)

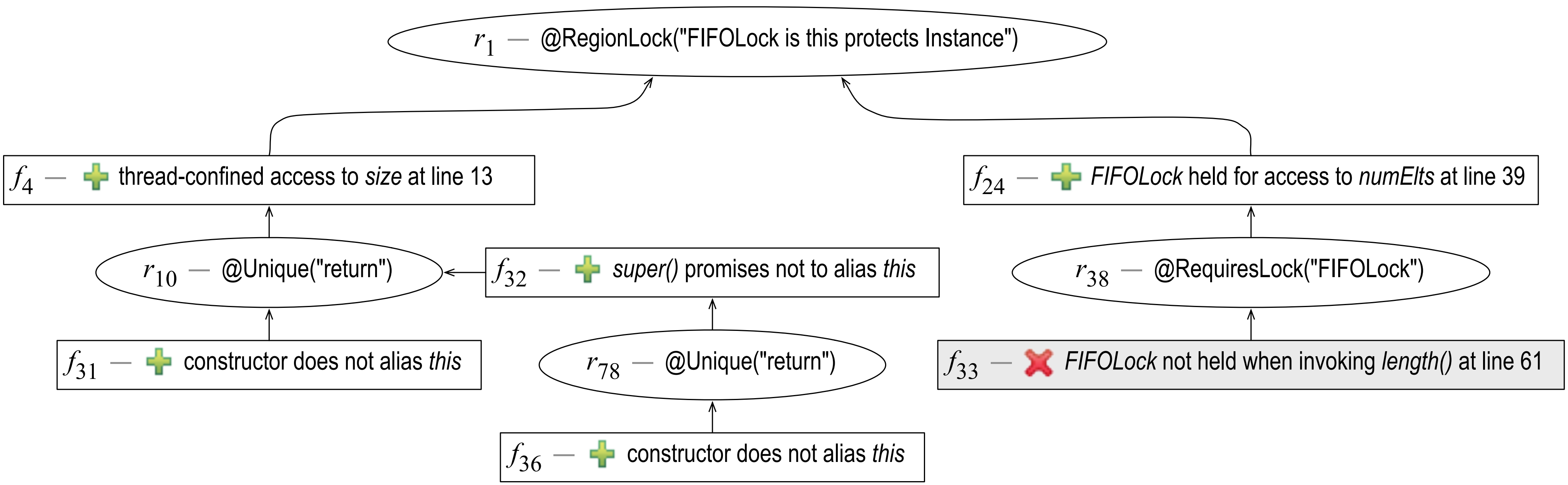
Overcoming lost relationships

```
1 @RegionLock("FIFOLock is this protects Instance")
2 public class BoundedFIFO {

10   @Unique("return")
11   public BoundedFIFO(int size) {
12     if (size < 1) throw new IllegalArgumentException();
13     this.size = size;
14     buf = new LoggingEvent[size];
15   }

38   @RequiresLock("FIFOLock")
39   public int length() { return numElts; }
```

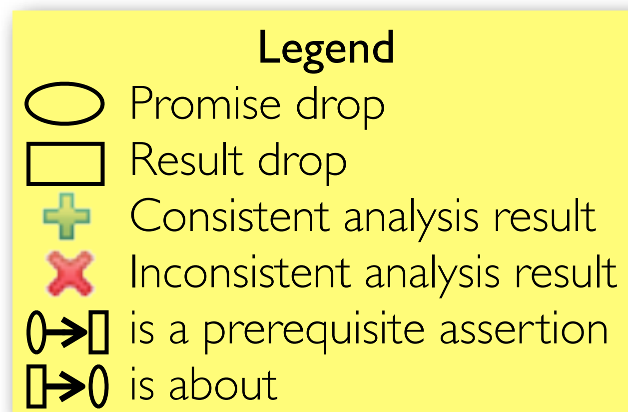
Analysis Results for `BoundedFIFO`

	Finding	About	Prerequisite	Description
f_{24}	+	r_1	r_{38}	`FIFOLock` held for access to `numElts` at line 39

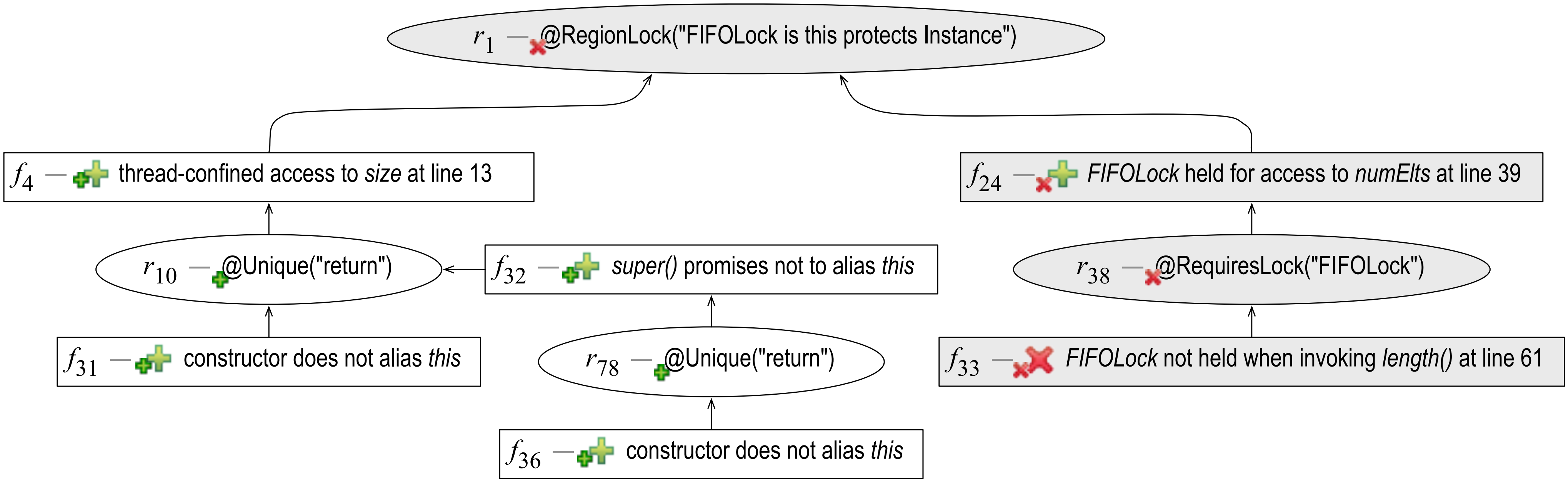
Drop-sea graph: Tracking relationships



- Tabular analysis results are modeled as nodes a graph (tree if no recursion)
- *Drops* are the nodes in the graph



Overcoming unknown impact of an ✖

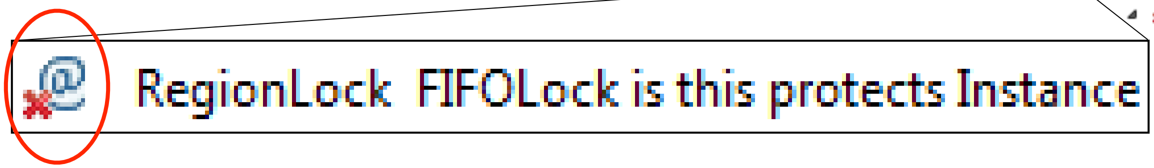


- Traversals of the graph yield aggregate verification results, which are stored on the drops
- The graph structure reveals relationships to the human users

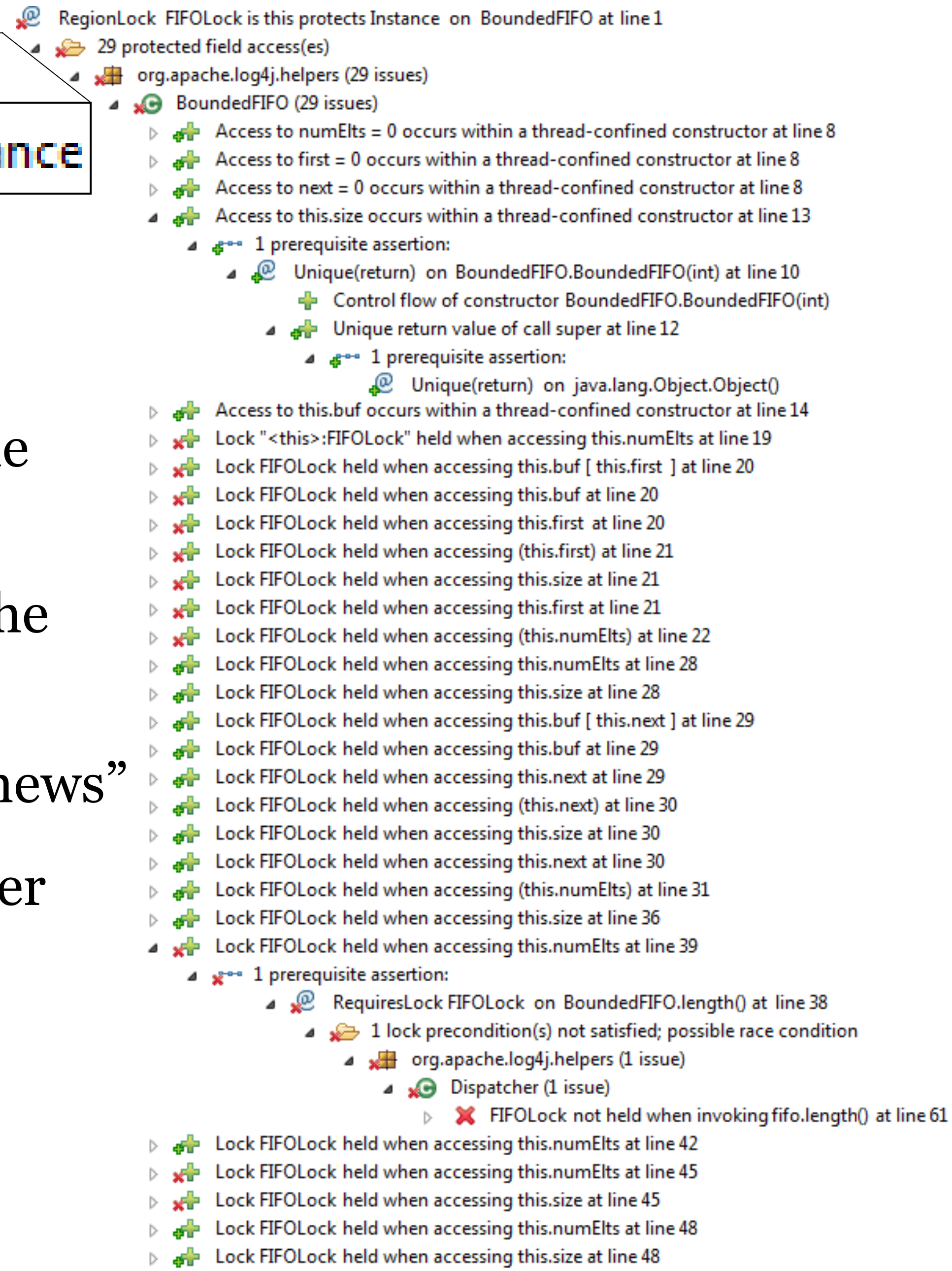
Legend

- Proof of model-code consistency (verified)
- Can't prove model-code consistency
- Promise drop
- Result drop
- Consistent analysis result
- Inconsistent analysis result
- is a prerequisite assertion
- is about

Answering the big question



- The lock use policy, FIFOLock, is *inconsistent* with the code
- The question to be addressed by the developer is *why?*
- The JSure tool presents a view of the drop-sea graph to the user →
    - There is "good news" and "bad news"
- To work toward consistency the user follows the trail of "X"s



Tool interaction toward consistency (I)

- RegionLock FIFOLock is this protects Instance on BoundedFIFO at line 1
  - 29 protected field access(es)
    - org.apache.log4j.helpers (29 issues)
      - BoundedFIFO (29 issues)
        - ...
        - Lock FIFOLock held when accessing this.numElts at line 39
          - 1 prerequisite assertion:
            - RequiresLock FIFOLock on BoundedFIFO.length() at line 38
              - 1 lock precondition(s) not satisfied; possible race condition
                - org.apache.log4j.helpers (1 issue)
                  - Dispatcher (1 issue)
                    - FIFOLock not held when invoking fifo.length() at line 61

Double-clicking on the inconsistent result (bottom) brings up the unprotected call in the source code of Dispatcher

Tool interaction toward consistency (2)

FIFOLock not held when invoking fifo.length() at line 61

```
58  LoggingEvent get() {
59    synchronized (this) {
60      LoggingEvent e;
61      while (fifo.length() == 0) {
62        try {
63          fifo.wait();
64        } catch (InterruptedException ignore) { }
65      }
66      e = fifo.get();
67      if (fifo.wasFull()) {
68        fifo.notify();
69      }
70      return e;
71    }
72  }
```

The programmer determines that the code is wrong and fixes line 59

```
58  LoggingEvent get() {
59    synchronized (fifo) {
60      LoggingEvent e;
61      while (fifo.length() == 0) {
```

JSure re-runs its analysis

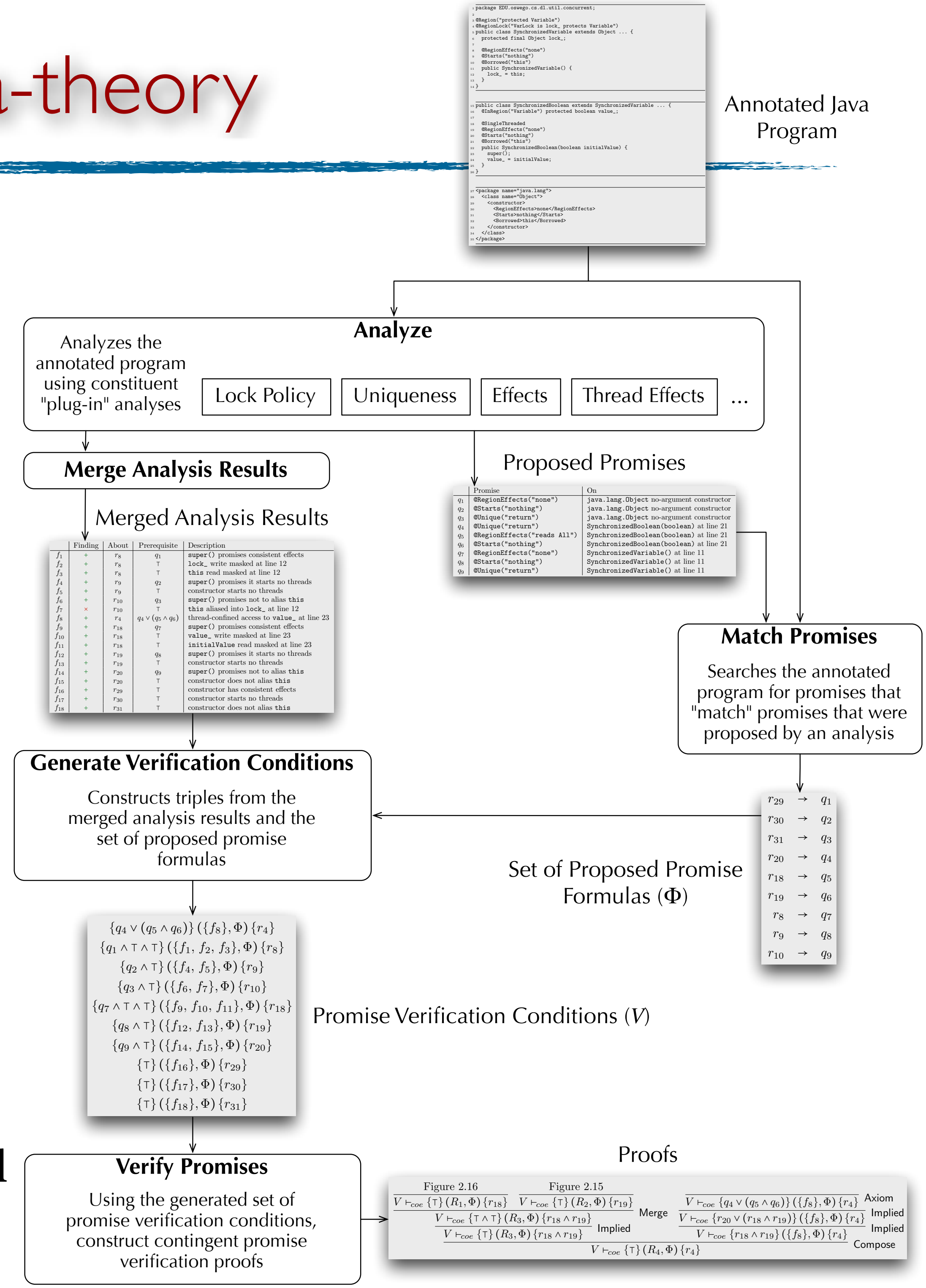
RegionLock FIFOLock is this protects Instance on BoundedFIFO at line 1

An aside on the meta-theory

- New meta-theory to support the drop-sea proof management model (Halloran)
- Overall assertion of soundness involves multiple formalisms

$$\textit{If } V \vdash_{coe} \{\phi\}\,(R',\Phi)\,\{\psi\} \textit{ is valid}$$
$$\textit{then } \mathcal{M}_{(T)} \models_{pl} \phi \rightarrow \psi \textit{ holds}$$

- (Not a Hoare Logic, but rather a logic that links chains of evidence)
- Feasible prerequisites for the constituent analyses to be combined
- Basis of abductive reasoning to “fill in” missing pieces of a model (next topic...)



Supporting model expression

- A limitation of analysis-based verification is the number of annotations required
  - 11 annotations were required to verify the lock use policy of BoundedFIFO, a tiny program
- Why so many annotations?
  - The annotations allow the verifying analyses to be modular (*i.e.*, avoiding a whole program analysis)
- We introduce two approaches to assist the programmer with model expression
  - Proposed promises
  - The scoped promise, @Promise
- These approaches can reduce the extent of model expression by orders of magnitude
  - In some cases down to 6.3 annotations per KSLOC (Sutherland)

Proposed promises

- How does the verification process "connect" analysis fragments?
  - Constituent analyses *propose promises* rather than look for them
  - A specialized program analysis, called promise matching, "matches" each proposed promise with a real promise in the code

Analysis Results for `BoundedFIFO`

	Finding	About	Prerequisite	Description
f_4	+	r_1	$q_1 \vee (q_2 \wedge q_3)$	thread-confined access to `size` at line 13
...				
f_{24}	+	r_1	q_4	`FIFOLock` held for access to `numElts` at line 39
...				
f_{32}	+	r_{10}	q_5	`super()` promises not to alias `this`

Proposed Promises

	Promise	On
q_1	`@Unique("return")`	`BoundedFIFO(int)` constructor
q_2	`@RegionEffects("none")`	`BoundedFIFO(int)` constructor
q_3	`@Starts("nothing")`	`BoundedFIFO(int)` constructor
q_4	`@RequiresLock("FIFOLock")`	`BoundedFIFO.length()`
q_5	`@Unique("return")`	`java.lang.Object` no-argument constructor

Promise matching

```
 2 public class BoundedFIFO {

10   @Unique("return")
11   public BoundedFIFO(int size) {

38   @RequiresLock("FIFOLock")
39   public int length() {
```

```
75 <package name="java.lang">
76  <class name="Object">
77    <constructor>
78      <Unique>return</Unique>
```

Proposed Promises

	Promise	On
q_1	`@Unique("return")`	`BoundedFIFO(int)`
q_2	`@RegionEffects("none")`	`BoundedFIFO(int)`
q_3	`@Starts("nothing")`	`BoundedFIFO(int)`
q_4	`@RequiresLock("FIFOLock")`	`BoundedFIFO.length()`
q_5	`@Unique("return")`	`Object()`

Matched Promises (the set Φ)

$$r_{10} \rightarrow q_1$$
$$r_{38} \rightarrow q_4$$
$$r_{78} \rightarrow q_5$$

- A specialized program analysis
- Results in a set of implications →
  - A real promise "implies" a proposed promise
  - If the real assertion holds the proposed assertion must hold
- Our proof calculus allows this set to be used to mark proposed promises as intended

Promise matching: Why implications?

Annotated Program

```
public class SynchronizedBoolean extends ... {
  @InRegion("Variable") protected boolean value_;

r18   @RegionEffects("none")
      @Starts("nothing")
r20   @Unique("return")
  public SynchronizedBoolean(boolean initialValue) {
    super();
    value_ = initialValue;
  }
}
```

Matches

$r_{20} \rightarrow q_4$

$r_{18} \rightarrow q_5$

Proposed Promises

```
...
q4  @Unique("return")
q5  @RegionEffects("reads All")
...
```

- A "match" is a *semantic* match—not a *textual* match
- Example: The match $r_{18} \rightarrow q_5$ (above)
  - Promising not to read or write to global program state is a stronger assertion than promising to only read global state
  - If the former holds the latter must hold

What has this got to do with supporting model expression?

Tool-assisted completion of partial models

- Promise matching has a practical aspect with respect to supporting model expression
  - The remaining proposed promises, after promise matching, can be proposed by the tool to the developer -- e.g., using a specially flagged annotation ("is this your intent?")

- The computation that produces verification results computes a "weakest prerequisite assertion" using remaining proposed promises
  - Computed in a manner analogous to weakest precondition in the classic verification literature -- but with very different semantics
  - Example: BoundedFIFO (with code repaired) from one promise

Using proposed promises (I)

The programmer enters the @RegionLock promise into BoundedFIFO

```
@RegionLock("FIFOLock is this protects Instance")
public class BoundedFIFO {

  LoggingEvent[] buf;
```

JSure can't verify the promise, but it proposes "missing" promises

RegionLock FIFOLock is this protects Instance on BoundedFIFO at line 1

27 unprotected field accesses; possible race condition detected

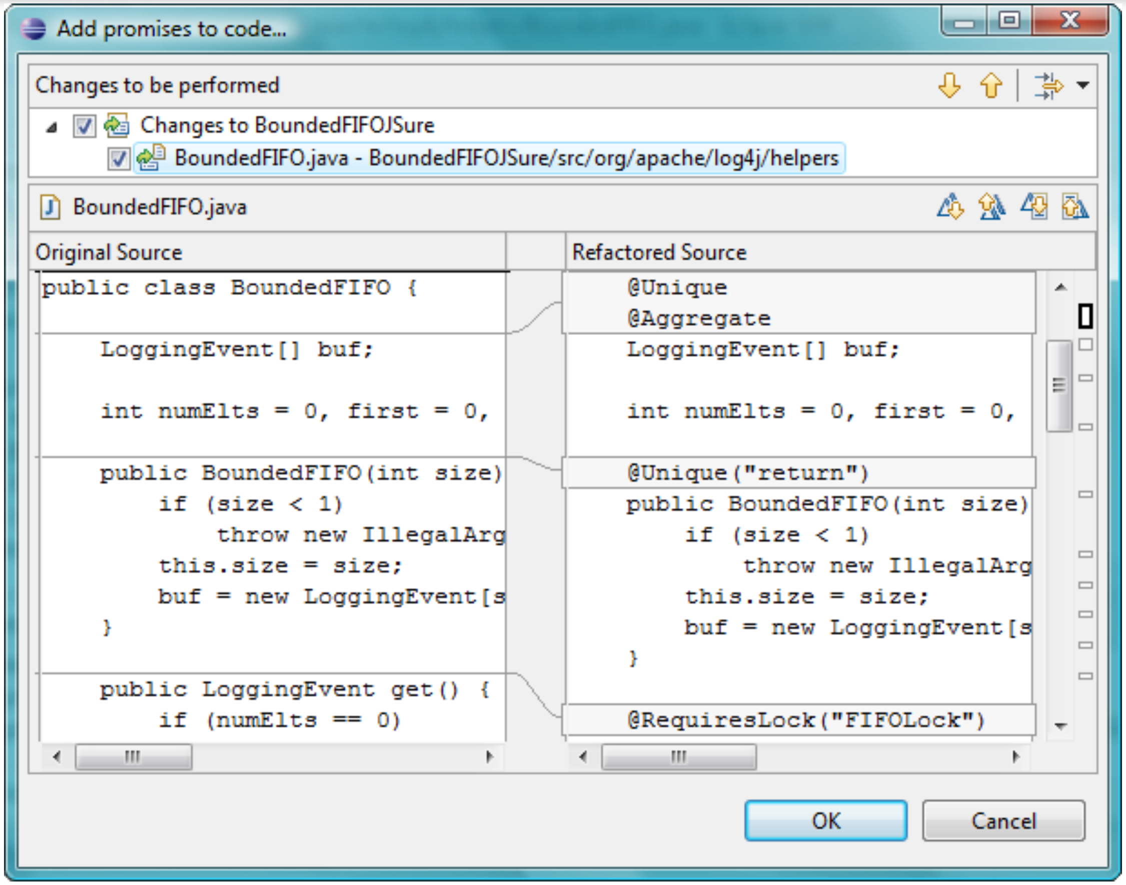
Proposed Promises (BoundedFIFOJSure)

Add promises to code...

Description	Resource	Line
Aggregate	/BoundedFIFOJSure/src/org/apache/log4j/helpers/BoundedFIFO.java	8
Unique	rg/apache/log4j/helpers/BoundedFIFO.java	8
Unique("return")	/BoundedFIFOJSure/src/org/apache/log4j/helpers/BoundedFIFO.java	12
RequiresLock("FIFOLock")	/BoundedFIFOJSure/src/org/apache/log4j/helpers/BoundedFIFO.java	19
RequiresLock("FIFOLock")	/BoundedFIFOJSure/src/org/apache/log4j/helpers/BoundedFIFO.java	29
RequiresLock("FIFOLock")	/BoundedFIFOJSure/src/org/apache/log4j/helpers/BoundedFIFO.java	38
RequiresLock("FIFOLock")	/BoundedFIFOJSure/src/org/apache/log4j/helpers/BoundedFIFO.java	42
RequiresLock("FIFOLock")	/BoundedFIFOJSure/src/org/apache/log4j/helpers/BoundedFIFO.java	46
RequiresLock("FIFOLock")	/BoundedFIFOJSure/src/org/apache/log4j/helpers/BoundedFIFO.java	50
RequiresLock("FIFOLock")	/BoundedFIFOJSure/src/org/apache/log4j/helpers/BoundedFIFO.java	54

Using proposed promises (2)

Using the context menu the programmer directs the tool to add the promises



With the 10 additional promises, JSure can verify the model

RegionLock FIFOLock is this protects Instance on BoundedFIFO at line 1

Tool-assisted completion of partial models

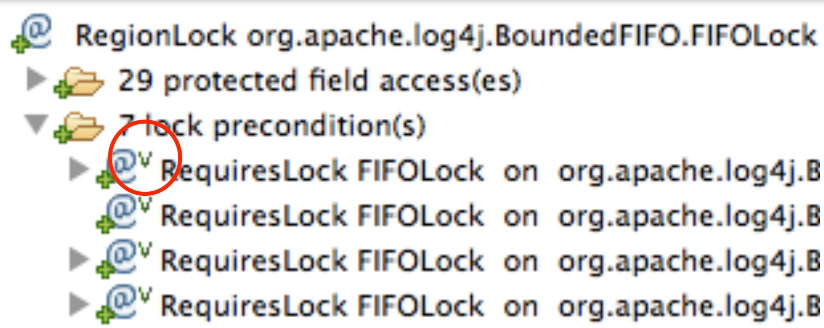
- The approach is *abductive*—working from a desired consequent to a possible antecedent
  - Our example worked because we supplied the lock use policy — the remaining annotations were proposed by the tool (typical)
- Everything is tool-verified, so we remain sound
  - Composition (key to scale-up) in this case can assist the tool user with model expression
- Most of our underlying analyses have low "perplexity," which facilitates practical abduction

@Promise: Avoiding repetitive annotation

There is another...

- We introduce @Promise to help avoid repetitive annotation
- One intent—one annotation
- Uses an aspect-like syntax
- Semantics: *all* (even in future)
- Constituent analyses see *virtual* promises

```
@RegionLock("FIFOLock is this protects Instance")
@Promises({
  @Promise("@Unique(return) for new(**)"),
  @Promise("@RequiresLock(FIFOLock) for *(**)")
})
public class BoundedFIFO {

  @Unique
  @Aggregate
  LoggingEvent[] buf;

  ...
}
```



Thread coloring [Sutherland]

- Allows developers to specify and verify thread usage policies
  - Non-lock concurrency (thread-confinement)
- Benefits
  - @Promise is effective for documenting thread usage policies
    - "By using scoped promises, we replace over **1,700** color constraint annotations with **six** scoped promises in each of the nine packages"

```
@Promises({
  @Promise("@Color(DBExaminer | DBChanger) for get*(**) | is*(**) | same*(**)"),
  @Promise("@Color(DBExaminer | DBChanger) for compare(**) | connectsTo(**)"),
  @Promise("@Color(DBExaminer | DBChanger) for contains*(**) | describe()"),
  @Promise("@Color(DBExaminer | DBChanger) for find*(**) | num*(**)"),
  @Promise("@Color(DBChanger) for set*(**) | make*(**) | modify*(**)"),
  @Promise("@Color(DBChanger) for clear*() | new(**) | add*(**)")
})
package com.sun.electric.database.network;
```

Takes advantage of stylized naming schemes

Sutherland and Scherlis, *Composable Thread Coloring*, in Proc. PPOPP, 2010, pp.233-244.

Supporting contingencies

- Our approach supports three kinds of unverified contingencies:
  - @Vouch – Vouches for presumptive false positives
  - @Assume – Assume truth of unverified assertion (e.g., about a library component)
  - Turning off a constituent analysis – promises that need to be verified by that analysis will show as *correct with contingency*

- The "red dot"
  - The impact of all contingencies are visibly indicated with a trail of "red dot"s in the user interface
  - A programmer must be willing to prick a finger and vouch for the unverified contingency with a small drop of virtual blood

@Vouch – Hadoop MapReduce

```
@Region("StatusState")
@RegionLock("StatusLock is this protects StatusState")
public class JobInProgress {
  @InRegion("StatusState")
  JobStatus status;
  ...
}
```

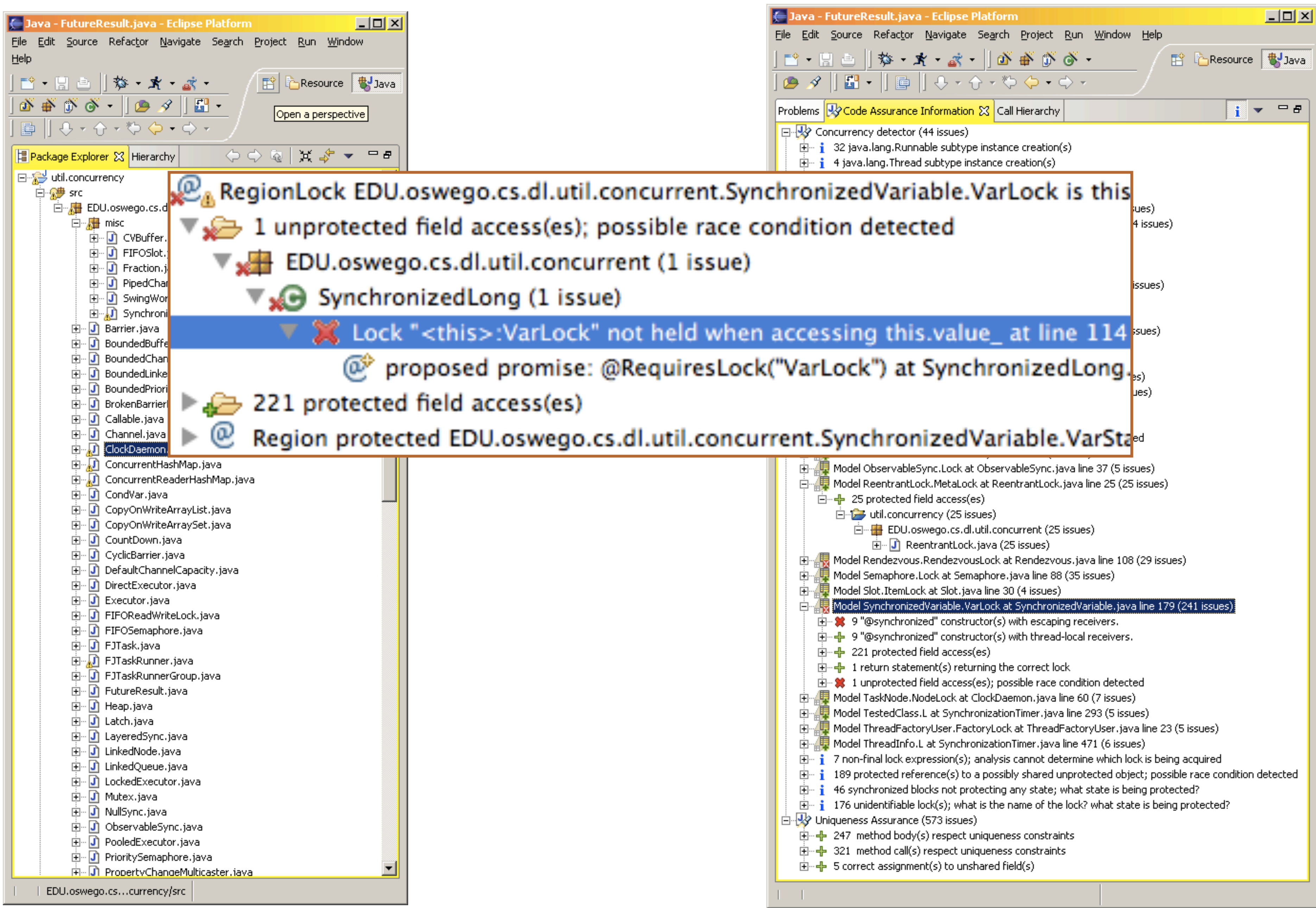
Vouching for test code as an exception to a lock use policy

```
public class CapacityTestUtils {
  @Vouch("This code is used only for testing")
  static class FakeJobInProgress extends JobInProgress { ... }
  ...
}
```

This vouch only applies to results within the declaration of FakeJobInProgress

- RegionLock StatusLock is this protects StatusState on JobInProgress at JobInProgress.java line 102
  - 21 unprotected field access(es); possible race condition detected
    - org.apache.hadoop.mapred (21 issues)
      - CapacityTestUtils.FakeJobInProgress (3 issues)
        - Lock "<this>:StatusLock" not held when accessing this.status at line 319
          - 1 prerequisite assertion:
            - Vouch "This code is used only for testing" at CapacityTestUtils.java line 299
        - Lock "<this>:StatusLock" not held when accessing this.status at line 323
        - Lock "<this>:StatusLock" not held when accessing this.status at line 324
      - CapacityTestUtils.FakeTaskTrackerManager (1 issue)

Example: A bug in Oswego util.concurrent



Evaluation activity: Field trials

- Conducted nine field trials of the JSure tool with disinterested practitioners
  - Field trials were conducted in the client's facilities
- On-site at client's location (code access limited)
- Experienced client engineers worked side-by-side work with JSure

Chris Douglas (of Yahoo!) and Nathan Boy (of SureLogic) working inside Yahoo Building E

A small sample of code examined

Duration (days)	Organization	Software Examined	Code Size (KSLOC)
3	*Company-A*	Commercial J2EE Server-*A*	350
3	NASA/JPL	Distributed Object Manager	42
		MER Rover Sequence Editor	20
		File Exchange Interface	12
		Space InfeRed Telemetry Facility	18
3	Sun	Electric – VLSI Design Tool	140
3	*Company-B*	Commercial J2EE Server-*B*	150
3	Lockheed Martin	Sensor/Tracking (CSATS)	50
		Weapons Control Engagement	30
1	Lockheed Martin	Equipment Web Portal	75
3	NASA/JPL	Testbed	65
		Service Provisioning (SPS)	40
		Mission Data Processing (MPCS)	100
		Next-Generation DSN Array	50
3	NASA/JPL	Maestro	17
		Command GUI	139
		Accountability Services Core	48
3	Yahoo!	Hadoop HDFS	107
		Hadoop MapReduce	281
		Hadoop ZooKeeper	62

Two broad categories: (1) server/infrastructure and (2) naval and aerospace mission support

Evaluation of approach

1. Scalability with respect to code size
   - Tool scales linearly, 64-bit JVM, uniqueness (turned-off/red-dot)
2. Effectiveness with respect to defects found and perceived value
   - Identified 79 race conditions in 1.6 million lines of Java code
   - Developed 376 models of programmer intent about lock use
     - 1,603 annotations added to 1.6 million lines of Java code
3. Compatibility with the incremental reward principle
   - "We found a number of significant issues with just a few hours of work. We really like the **iterative approach**. We really like the start-with-nothing approach (We hate tools that spew thousands of problems that are not actionable)."
4. Support for adoption late in the software lifecycle
   - Most systems examined were in operations and maintenance
     - Some very mature (JavaEE Server-B released for 3 years)
     - Code had passed acceptance evaluation for deployment

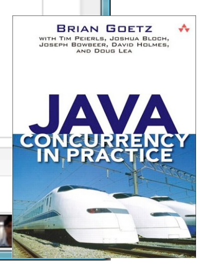
Perception of client participants

- “It would have been **difficult if not impossible** to find these issues without [JSure].”
- “The instances uncovered in this analysis were in **very mature operational code**.”
- “Team developed 63 lock models and [JSure] identified logic and programming errors in the Common Sensor and Tracking (CSAT) servers and Weapons Control Engagement segments that **extensive review and testing did not discover**.”
- “To me the most valuable thing is the basic fact that you’ve given us a methodology to **document the concurrency related design intent**. I’m actually considering implementing a policy that you can’t add a synchronize to the code without documenting [in JSure] what region it applies to.”
- “[JSure] was reported by all participants as helping them to **understand and document the thread interactions they had already designed and implemented**. This was an unanticipated, and indirect, benefit from the study.”
- To a manager, “one mistake and the **phone starts to ring**.”

JSure Modeling Language

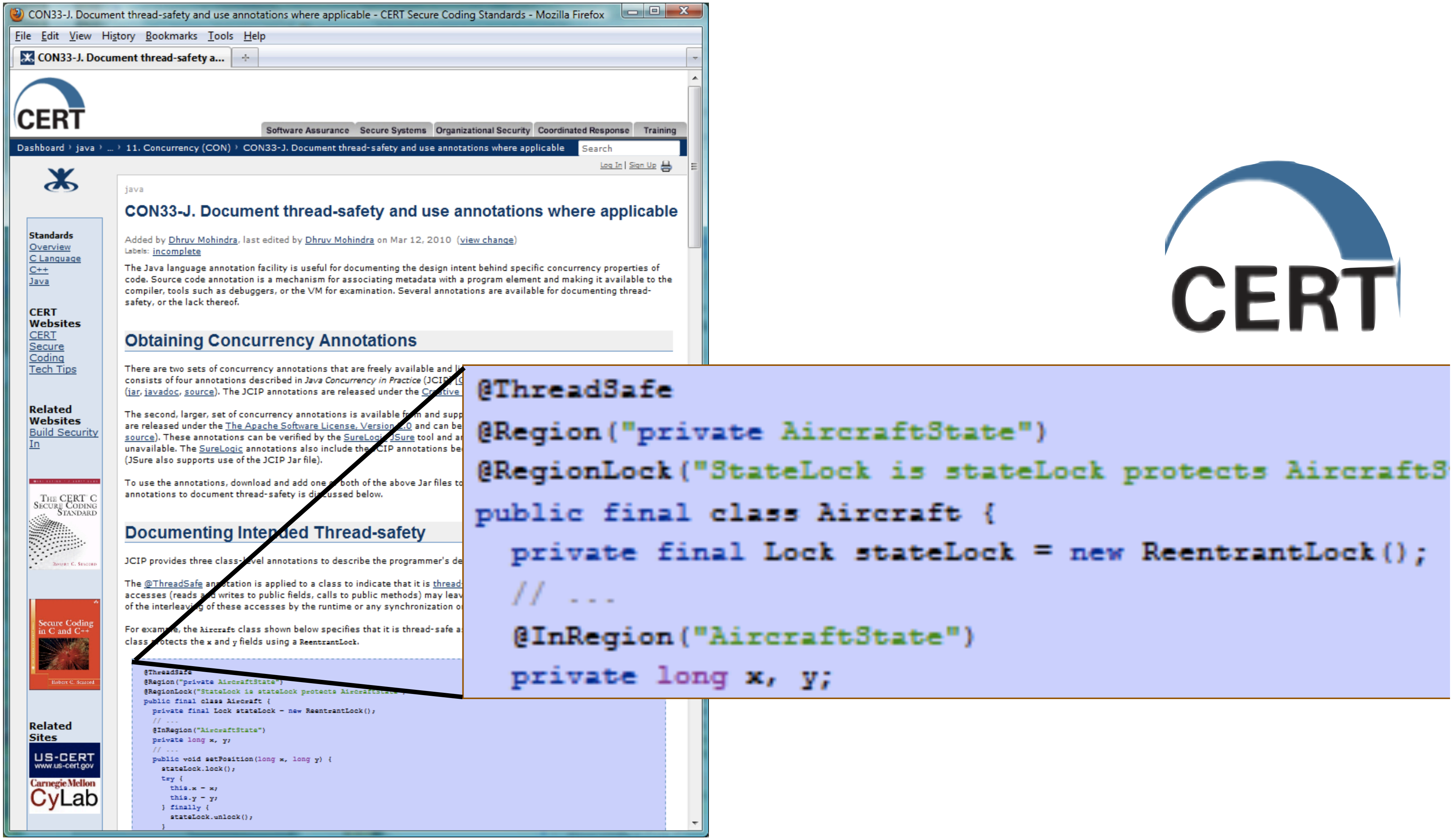
- Released under Apache open source license
  - http://surelogic.com/promises/index.html
  - http://promises.sourceforge.net/
- Primarily for use on Apache Hadoop

- Used by the Timing Framework
  - Animation in Swing
  - Haase - Filthy Rich Clients



- Goetz, et al. (JCIP) annotations are supported by the tool
  - e.g., @GuardedBy

Java Secure Coding standards



Summary

- ***Vision***: Create focused *analysis-based verification* for software quality attributes[1] as a scalable[2] and adoptable[3] approach to verifying[4] consistency of code with its design intent[5]

1. Quality attributes: E.g., safe concurrency with locks, data confinement to thread roles, static layer structure, many others
2. Scalable: Adapt constituent analyses to enable composition
   - Keys: chosen quality attributes, drop-sea (composition), scoped promises, contingencies
3. Adoptable: Before-lunch test (incremental reward principle)
4. Verification: No false negatives from analysis targeted to an attribute and a model
5. Design intent: Fragmentary models/specifications focused on quality attributes

Soundness at scale that ordinary programmers can use on non-trivial program properties