Source-Code Instrumentation and Quantification of Events

Robert E. Filman

RIACS

rfilman@mail.arc.nasa.gov

Klaus Havelund

Kestrel Technology

havelund@email.arc.nasa.gov

NASA Ames Research Center

Moffett Field, CA 94035 U.S.A.

Separation of Concerns

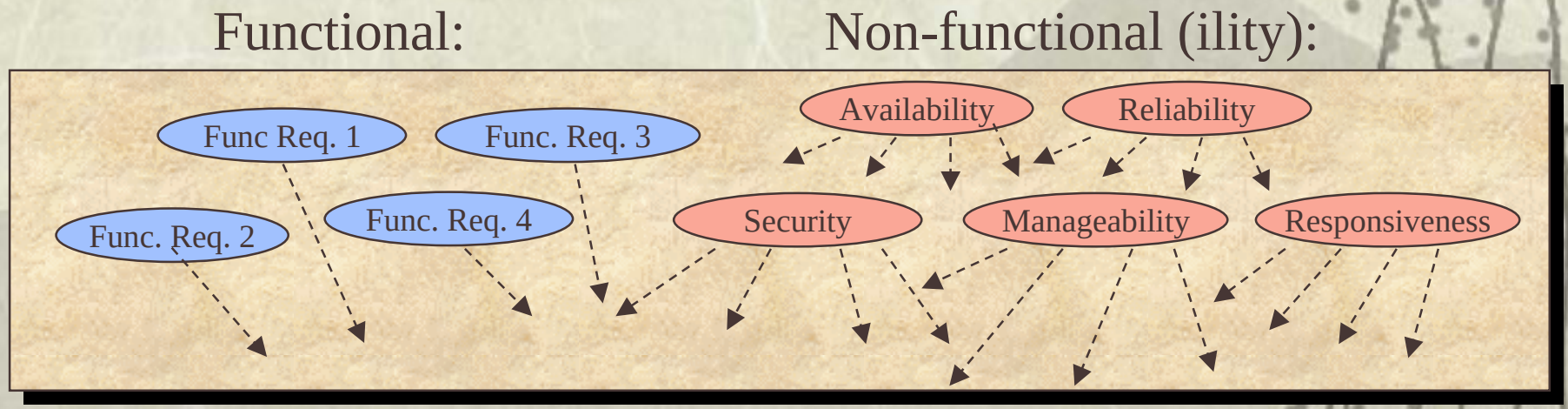
- A fundamental engineering principle is that of ***separation of concerns***
  - Realizing different system concepts as separate, weakly linked elements
  - Distribution of expertise
- Separation of concerns promises better
  - Maintainability
  - Evolvability
  - Reusability
  - Adaptivity
- Concerns cross-cut
  - Apply to different modules in a variety of places
- Concerns must be composed to build running systems

In conventional programming, the code for different concerns often becomes mixed-together (*tangled*)

Examples of Software Concerns

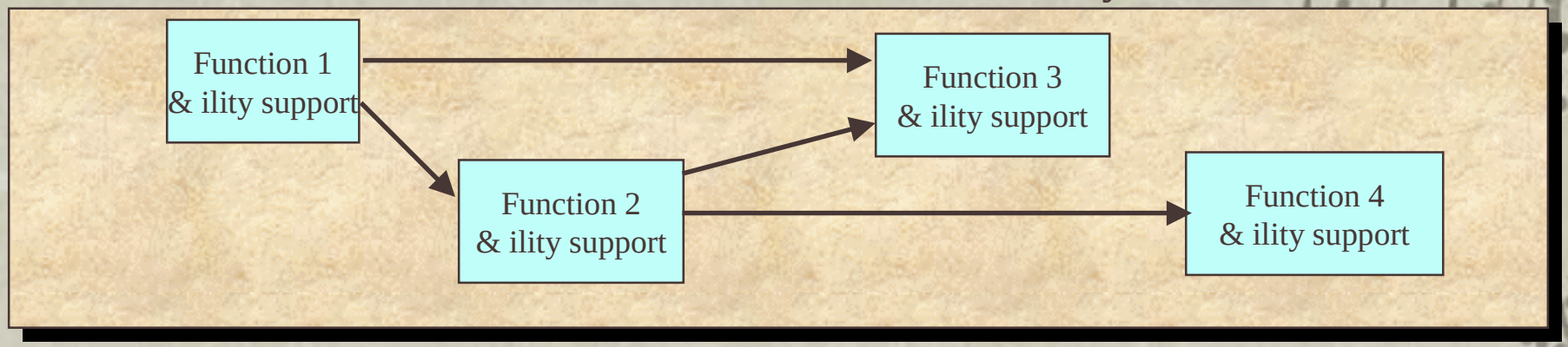
- **Security**
  - **Always call the security check before allowing database access**
- **Accounting**
  - **Always debit the user's account on each access to a service of objects in the class…**
- **Synchronization**
  - **Don't let multiple users call any of methods *f*, *g*, or *h* on a single object simultaneously**
  - **The effects of these actions should be transactional**
- **Quality of service**
  - **Queue up the waiting calls handling them by priority**
- **Reliability**
  - **Provide replicants of this object**
- **Performance enhancements**
  - **Cache the results of calls to elements in this class**
  - **Display routines should show the results of changes, except display routines called in the scope of other display routines should buffer their changes for display all at once**

Functional and Non-Functional Requirements



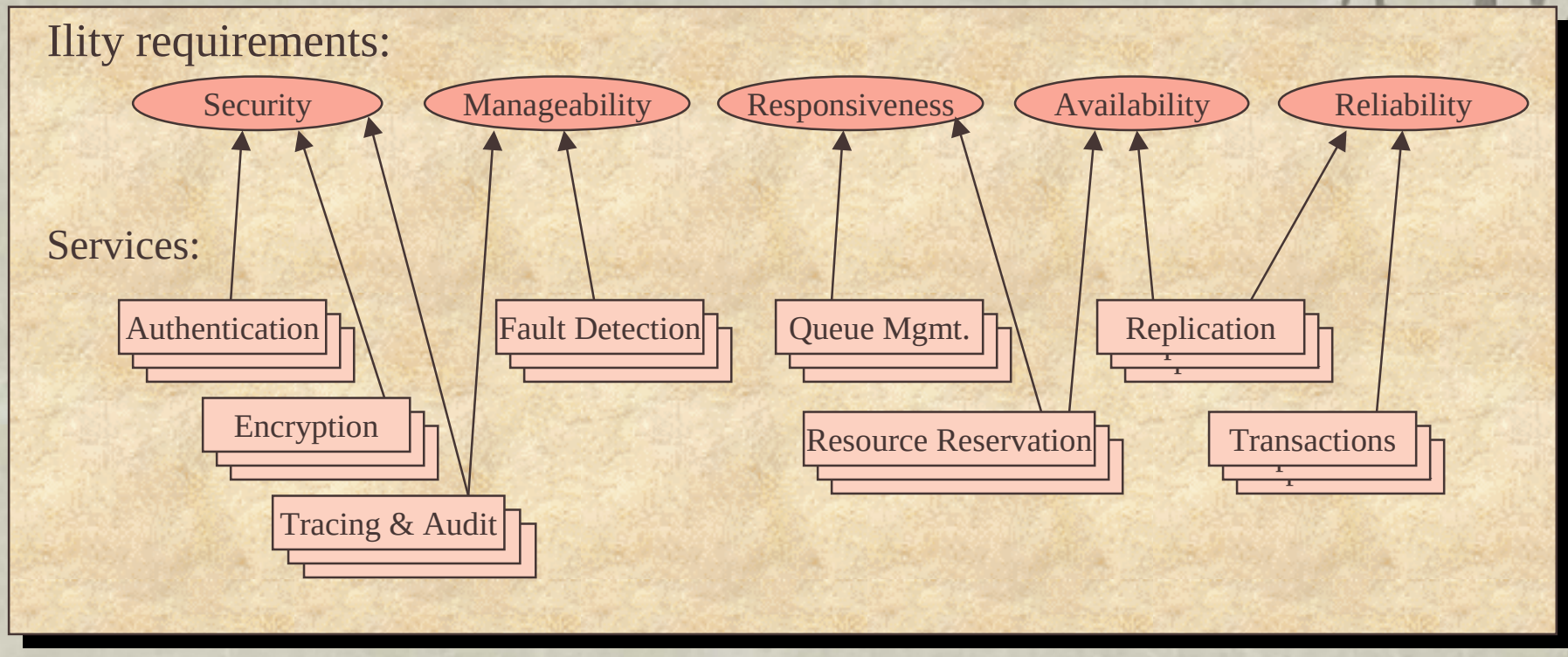
Functional requirements map to specific components

Ility requirements map almost everywhere



Services and Ilities

- Ility requirements are implemented by combinations of service algorithms.
- Supporting ilities involves a complex selection from sets of alternative service algorithms.
- The services must be invoked pervasively throughout the system.

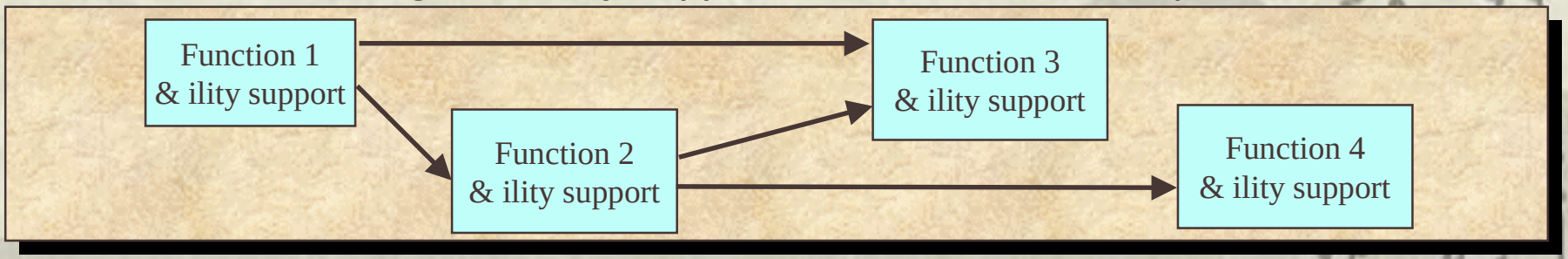


Object Infrastructure Framework Research Hypothesis

- **Ilities can be achieved by inserting services into the communication path between functional components**
  - **On both sides of the communication**
- **Frameworks that automate service insertion can systematically achieve non-functional requirements**
  - **Object Infrastructure Framework (OIF)**

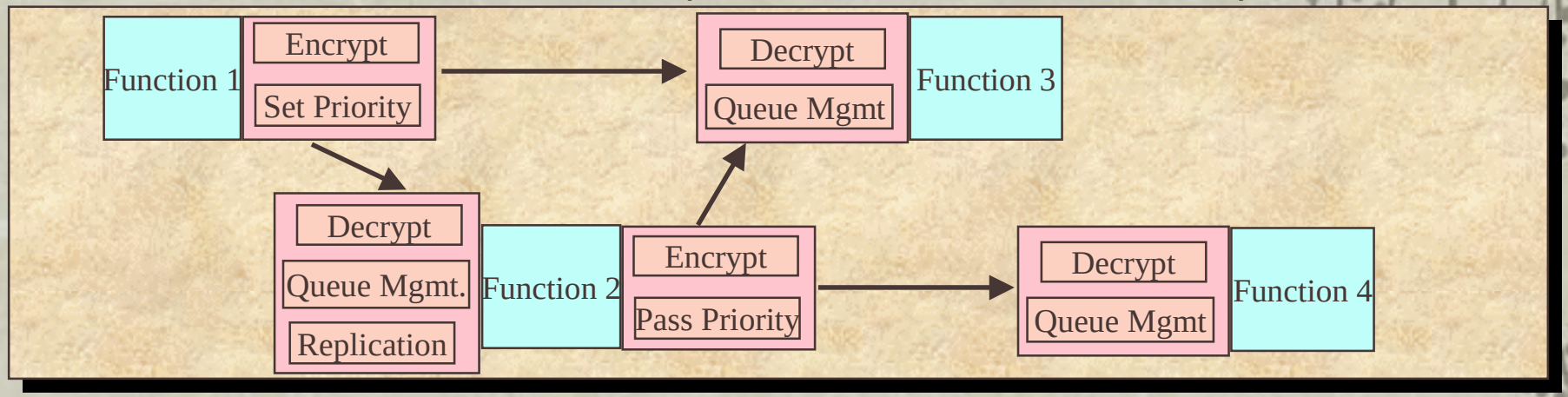
Architecture with Services in Component Communications

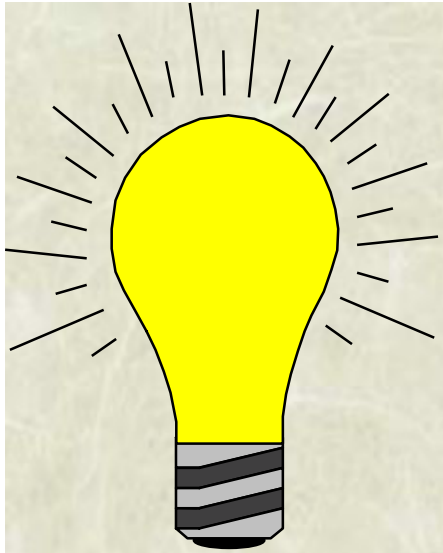
Traditional designs mix ility support within functional components:



Key insight: Service functionality can be separated from functional logic by inserting the services into the communications between functional components.

OIF is demonstrating frameworks that insert services Replication into the communication paths between functional components.

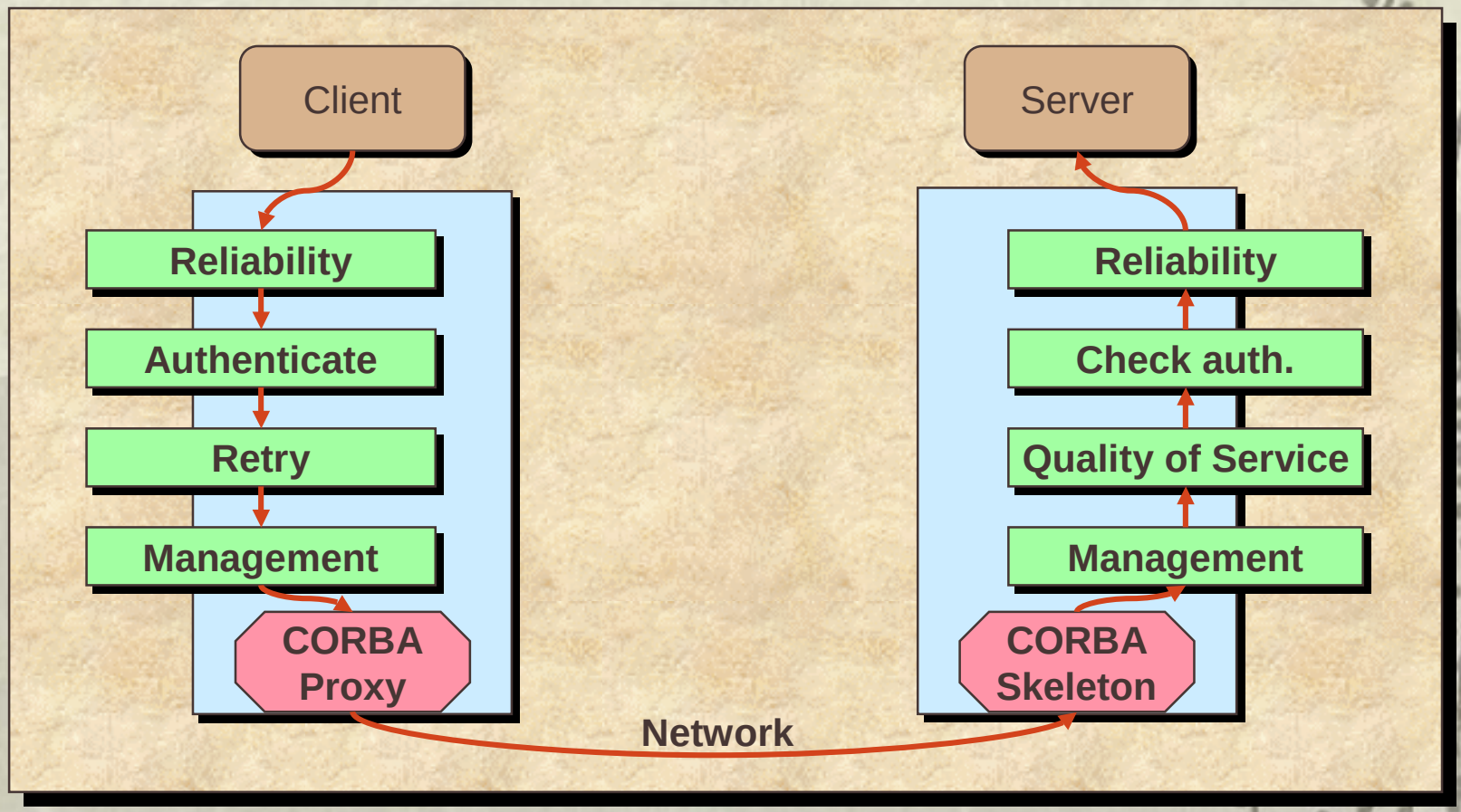




Key OIF Ideas

- **Injecting behavior** on the communicate paths between components
  - Injectors are discrete, uniform objects
  - Injectors are by object/method
  - Injectors are dynamically configurable
- **Annotated communications** allow injected services to pass parameters to service peers (e.g., message priority, user-id, tracing status)
- **Thread contexts** preserve annotations through calls
- **Pragma:** High-level specification language for describing desired injections (Pragma)

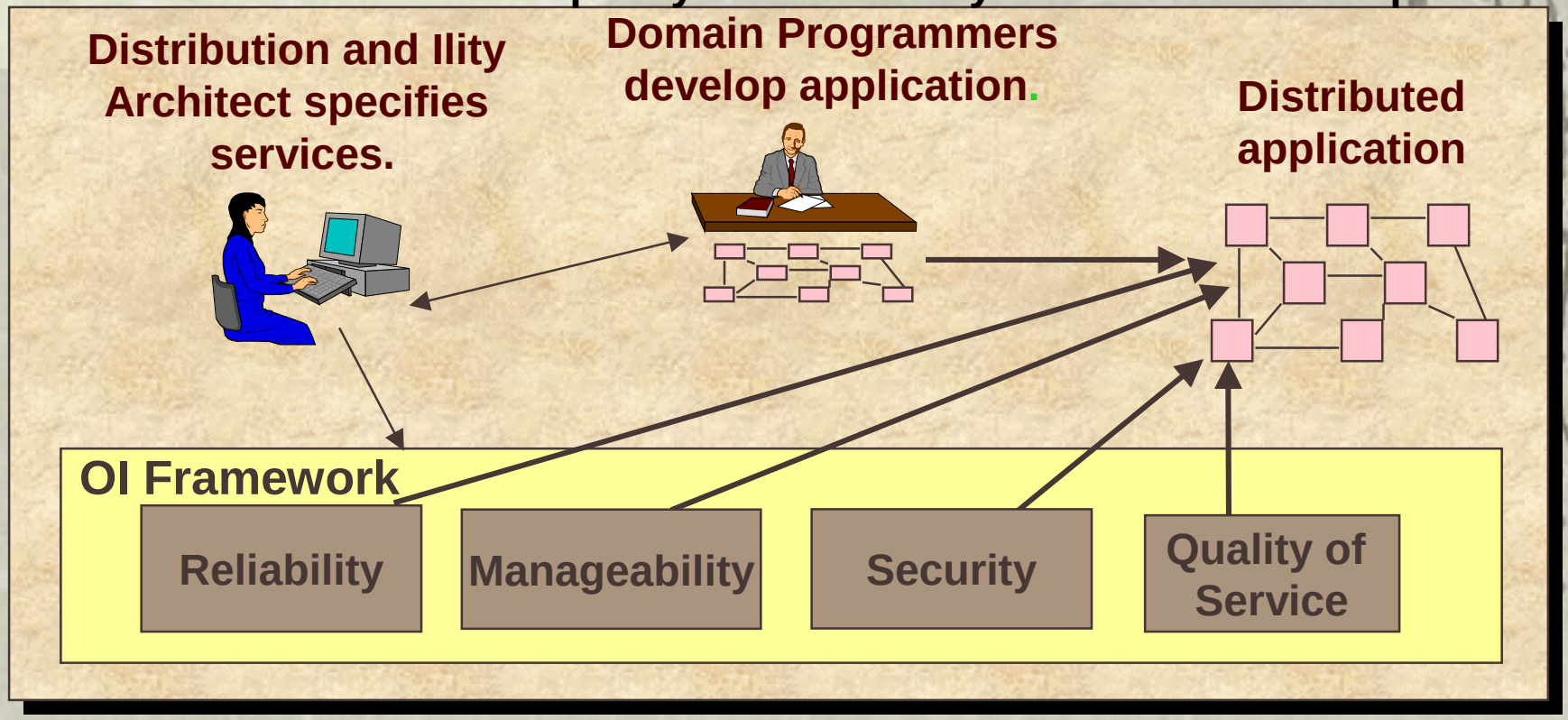
Configurable Proxies



- OIF's injectors can operate in pairs (e.g., encrypt/decrypt; request authentication/authenticate) or singly (e.g., retry on failure; log results)
- Configuration is by proxy/method instance
- Configuration is dynamic

OIF Process

- Map organizational *policies* to implementation
  - E.g.,
    - define a security policy
    - ensure that policy is followed by all distributed components



Aspect-Oriented Programming (AOP)

- **OIF is an Aspect-Oriented Programming system**
- **Software engineering technology for separately expressing systematic properties while nevertheless producing running systems that embody these properties**
- **Need to express**
  - **Base program**
  - **Separate concerns**
  - **How the separate concerns map to the base program**
    - **Or, if you prefer, just a jumble of program elements that must be combined.**

Quantification and Implicit Invocation

- **Programmatic essence of AOP is to be able to state universally quantified statements about (conventional, linear) programs**
- **And have the effect of these statements realized in programs that don't contain explicit notation applying these quantified statements**

In programs P, whenever condition C arises, perform action A.

- **Dimensions of concern for the designer and implementer of an AOP system:**
  - **Quantification: What kinds of conditions C can be specified.**
  - **Interface: What is the interface of the actions A. That is, how do they interact with base programs and each other.**
  - **Weaving: How will the system arrange to intermix the execution of the base actions of P with the actions A.**

Choices in Developing AOP Languages

- What quantified statements are allowed
  - Join points
    - Method calls
    - Field access
    - Abstract syntax
    - Control flow
  - Scope of quantification
    - Subclasses
    - By name
    - Lexical structure
- Syntax for expressing application
- Interaction among aspects and base code
  - Visibility
  - Ordering
  - Conflict resolution
- Implementation mechanism
  - Compiler
  - Byte-code manipulation
  - Dynamic wrapping
  - Frameworks
  - Meta-programming
  - Program transformation

Static and Dynamic Quantification

- In earlier work, we distinguished between
  - Static quantification: discernable from the syntactic structure of the specimen program
    - E.g, calls
  - Dynamic quantification: matching events that happen in the course of program execution.
    - E.g., cflow
- Currently exploring the hypothesis that (almost) all interesting “events” are dynamic, and that static quantification merely refers to those events that can be simply inferred from the static structure of the program.
  - Shadows in the code
  - Exception: program structural changes

Extreme Experiment

- **The extreme of expressiveness in quantification is to be able to quantify over all the history of events in a program execution**
- **Events are with respect to the abstract interpreter of a language**
- **Unfortunately, language definitions don't define their abstract interpreters.**

Events and Event Loci

Event	Syntactic locus
Accessing the value of a variable or field	References to that variable
Modifying the value of a variable or field	Assignments to that variable
Invoking a subprogram	Subprogram calls
Cycling through a loop	Loop statements
Branching on a conditional	The conditional statement
Initializing an instance	The constructors for that object
Throwing an exception	Throw statements
Catching an exception	Catch statements
Waiting on a lock	Wait and synchronize statements

More Events and Loci

Event	Syntactic locus
Resuming after a lock wait	Other's notify and end of synchronizations
Testing a predicate on several fields	Every modification of any of those fields
Changing a value on the path to another	Control and data flow analysis over statements (slices)
Swapping the running thread	Not reliably accessible, but atomization may be possible
Being below on the stack	Subprogram calls
Freeing storage	Not reliably accessible, but can try using built-in primitives
Throwing an error	Not reliably accessible; could happen anywhere

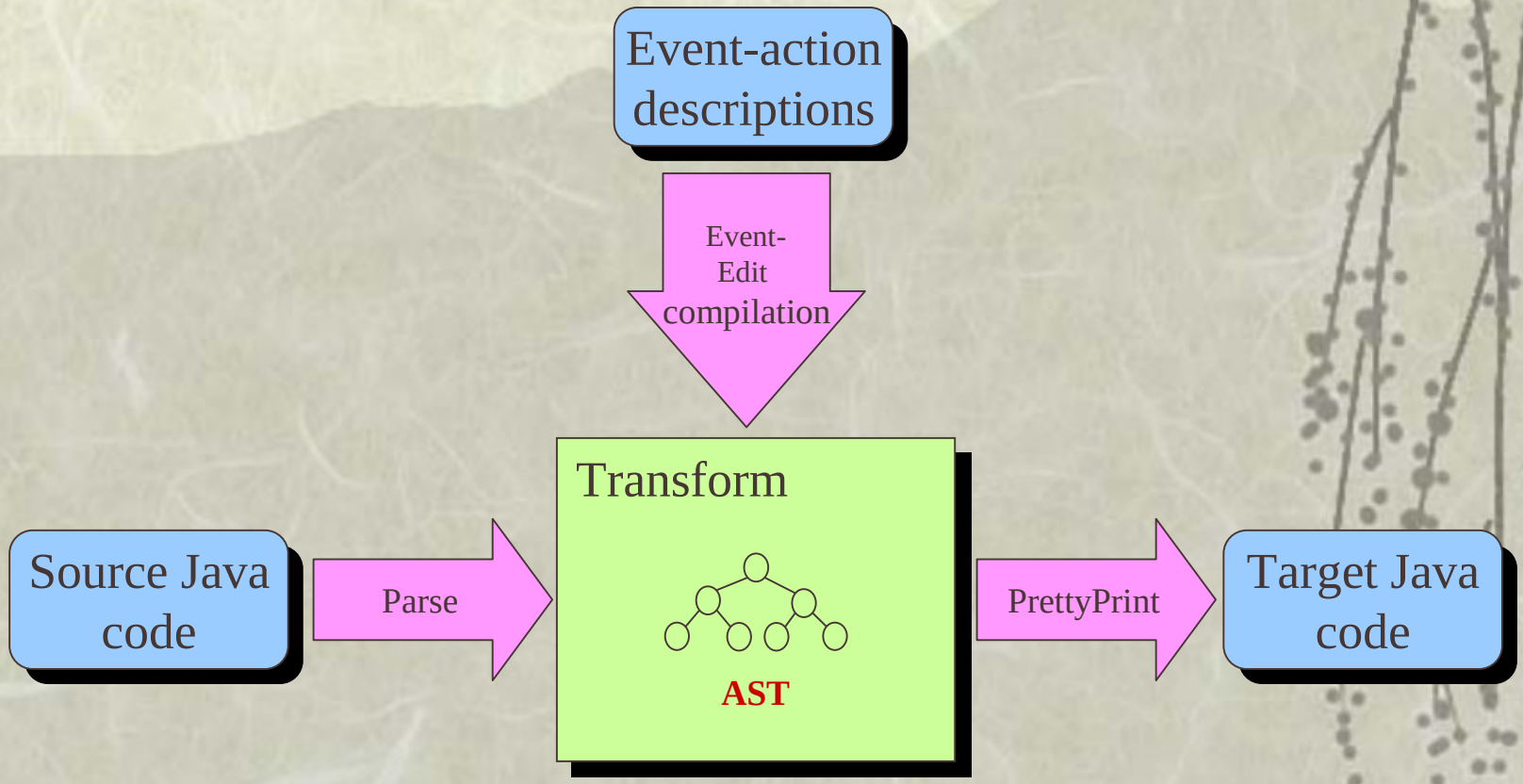
Research regime

- **Define a language of events and actions on those events.**
- **Determine how each event is reflected (or can be made visible) in source code.**
- **Create a system to transform programs with respect to these events and actions.**

Transformational Alternatives

- **For Java, can transform at**
  - **The source-code level**
  - **The byte-code level**

Architectural View



Applications

- **Applying AOP to debugging and validating concurrent programs.**
- **Applying AOP to monitor programs during operation, so that actions can be initiated in case bad things happen.**
- **Applying AOP as a general programming paradigm.**

Program Debugging

- **Detect multi-threading problems caused by access to shared resources by competing threads.**
- **Validate trace executions against user requirements.**
- **Validate multithreaded programs by exploring schedule interleavings.**

Detect Multi-threading Problems

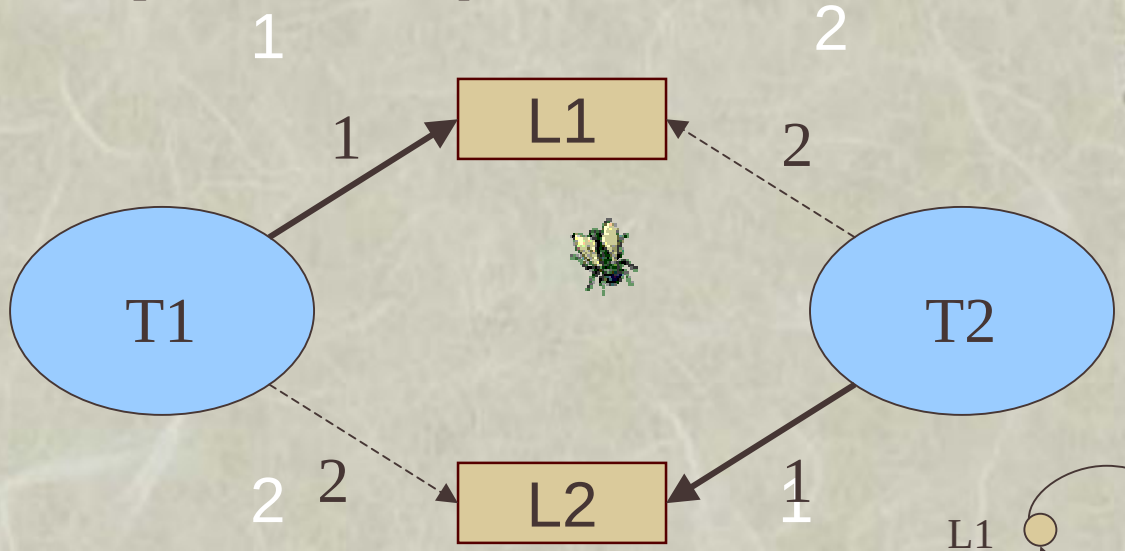
- Deadlocks**: Observe in what order locks are taken and released and infer potential deadlocks from cycles.**
- Data Races**: Observe what locks threads own when they access variables and infer potential data races from empty overlaps.**

Deadlocks

A deadlock can occur when threads access and lock shared resources, and lock these in different order.

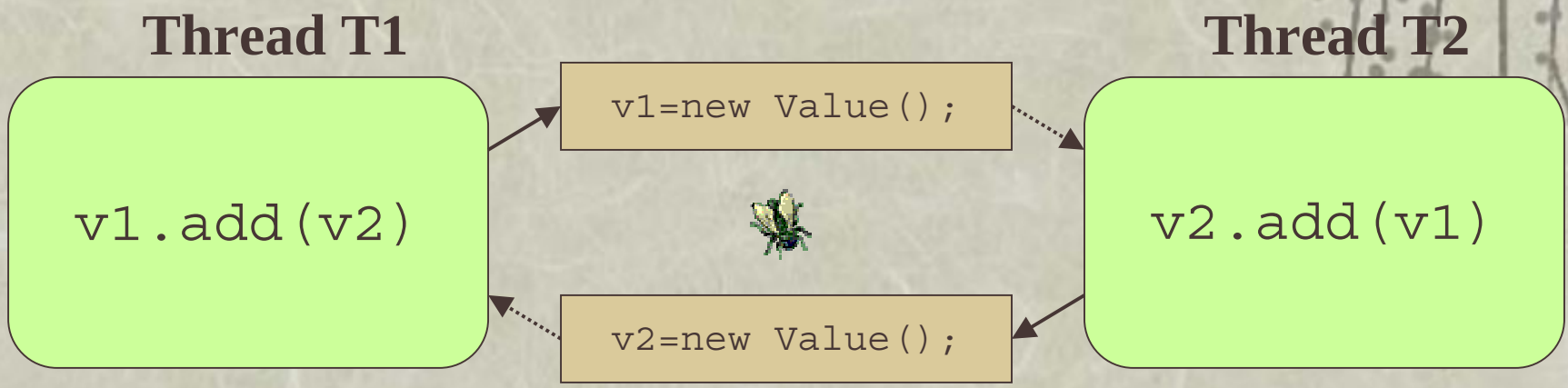
Example Solution: Impose order on locks: L1 < L2

Problem:
T1 locks L1 first
T2 locks L2 first



Java Program with Deadlock

```
class Value{
  int    x = 1;
  synchronized void add(Value v){x = x + v.get();}
  synchronized int  get(){return x;}
}
```



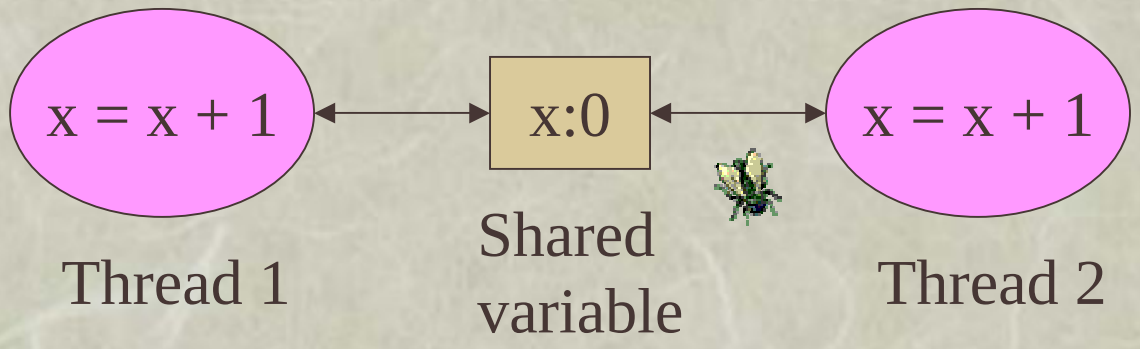
Applying AOP

```
aspect DeadlockDetection{
  when synchronize(obj){
    Thread curr = Thread.currentThread();
    Set locks = Threads.getLocks(curr);
    Graph.addEdges(locks,obj);
    Graph.findCycles();
    Threads.addLock(curr,obj);
  }
  when endof synchronize(obj){
    Threads.remove(curr,obj);
  }
}
```

Data Races

A data race occurs when two threads
- Access a shared variable,
- At least one access is a write, and
- No mechanism is used to prevent simultaneous access.

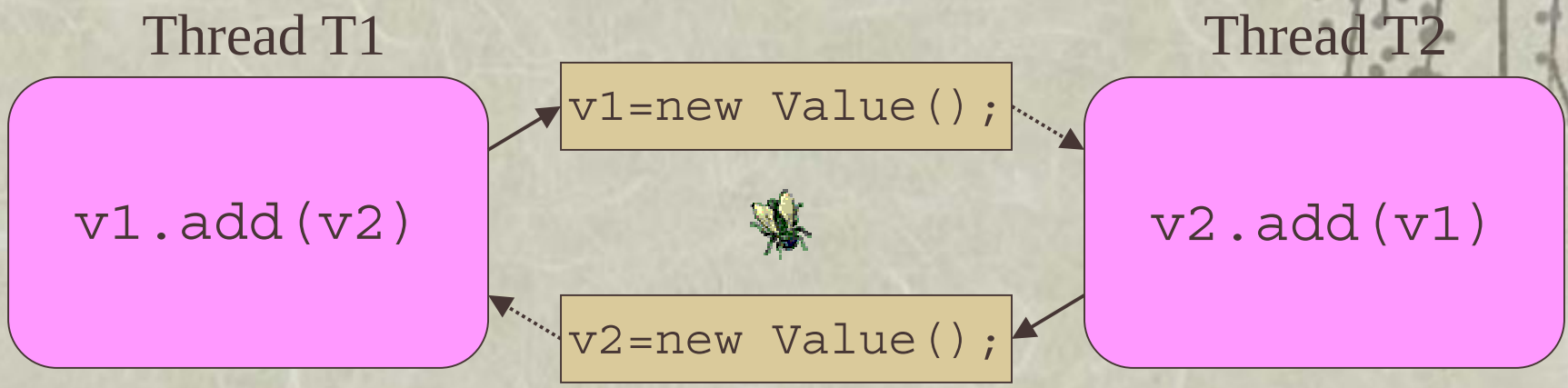
Example Solutions: monitors, semaphores, …



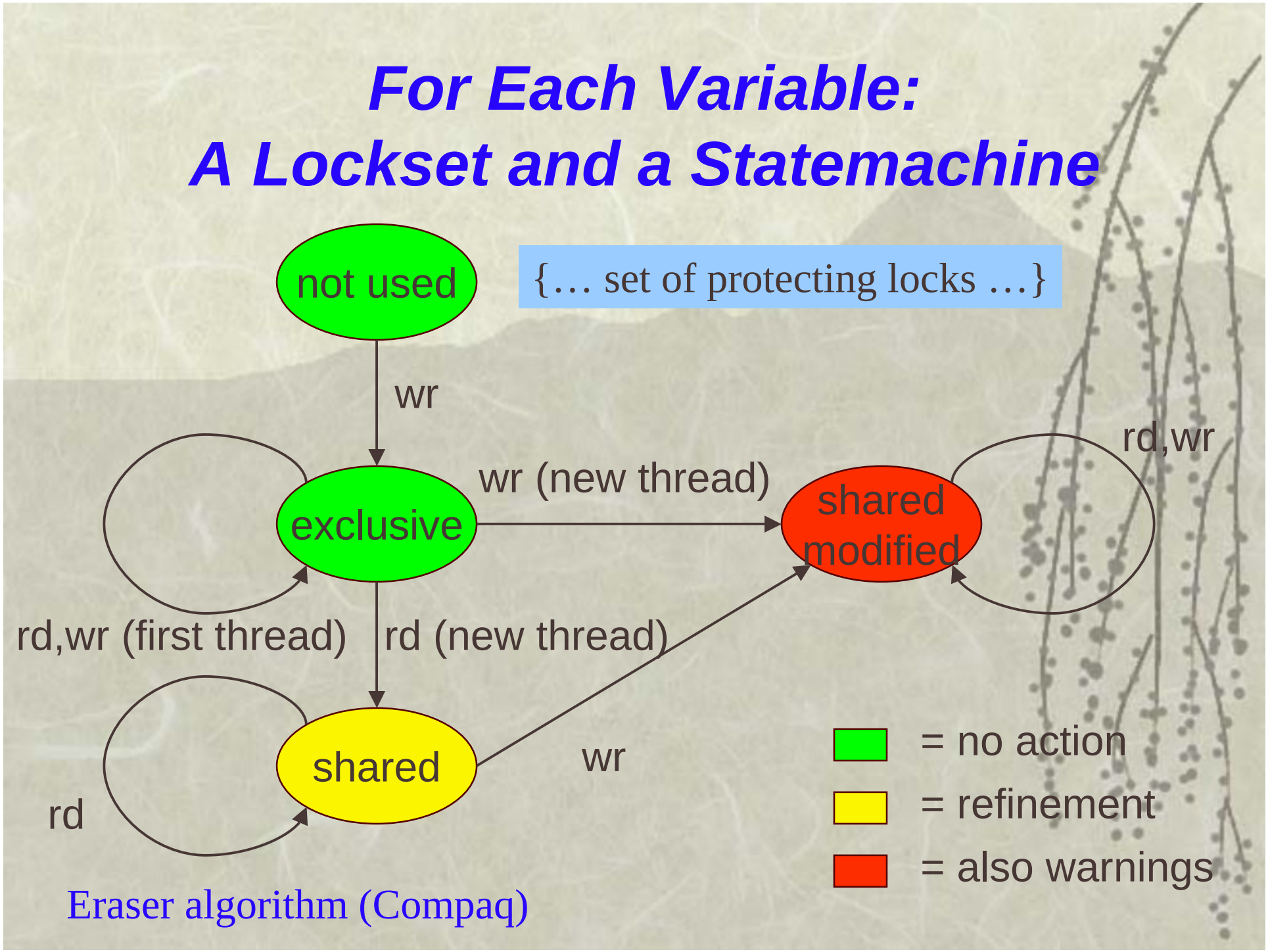
Result after both updates : 2 … or maybe 1

Java Program with Datarace

```
class Value{
  int x = 1;
  void add(Value v){x = x + v.get();}
  int get(){return x;}
}
```



For Each Variable:
A Lockset and a Statemachine



Eraser algorithm (Compaq)

Applying AOP

```
aspect DataraceDetection{
  when synchronize(obj){
    Thread curr = Thread.currentThread();
    Threads.addLock(curr,obj);
  }
  when endof synchronize(obj){
    Thread curr = Thread.currentThread();
    Threads.remove(curr,obj);
  }
  when accessto(var,isWrite){
    Thread curr = Thread.currentThread();
    Statemachine.update(curr,var,isWrite);
    Statemachine.checkEmptyness(var);
  }
}
```

Validating Execution Traces Against User Requirements

```
aspect CheckRequirements{
  when(CLOSED and not previously DO_CLOSE){
    Report("System closed by itself");
  }
  whennot(DO_CLOSE implies
            eventually(20)CLOSED){
    Report("System did not close");
    CloseSystem();  // repair
  }
}
```

Explore Scheduling

- **Simple example: assume that all variable accesses are protected with locks.**
- **Insert a call of a randomized yield statement in front of all synchronization statements and calls of synchronized methods.**
- **This will cause the scheduler to randomly make a context switch whenever a lock is taken. This may be used, for example, to reveal deadlocks.**

Related work

- De Volder et al. metaprogramming
- AOP through program transformation
  - Colcumbet, Fradet and Sudholt
  - Schonger et al. XML transformation
  - Skipper ku
- Nelson et al. concern-level foundational composition operators: correspondence, behavioral semantics and binding
- Walker and Murphy on events as join points

Concluding remarks

- **Presented**
  - **AOP**
  - **OIF, an AOP system**
  - **Event-based quantified approach to AOP**
    - **Developing an environment for experimenting with AOP languages (DSL for AOP)**