SRI

SRI

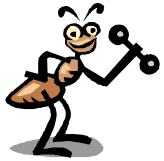
Dynamic Program Analysis

Klaus Havelund

The Automated Software Engineering Group

NASA Ames Research Center

California, USA



SRI

SRI

People Contributing

- Cyrille Artho (ETH Zurich, Switzerland)
- Saddek Bensalem (VERIMAG, France)
- Doron Drusinsky (Time Rover, USA)
- Allen Goldberg (NASA, USA)
- Corina Pasareanu (NASA, USA)
- Grigore Rosu (Univ. of Illinois at Urbana-Champaign) USA)
- Willem Visser (NASA, USA)

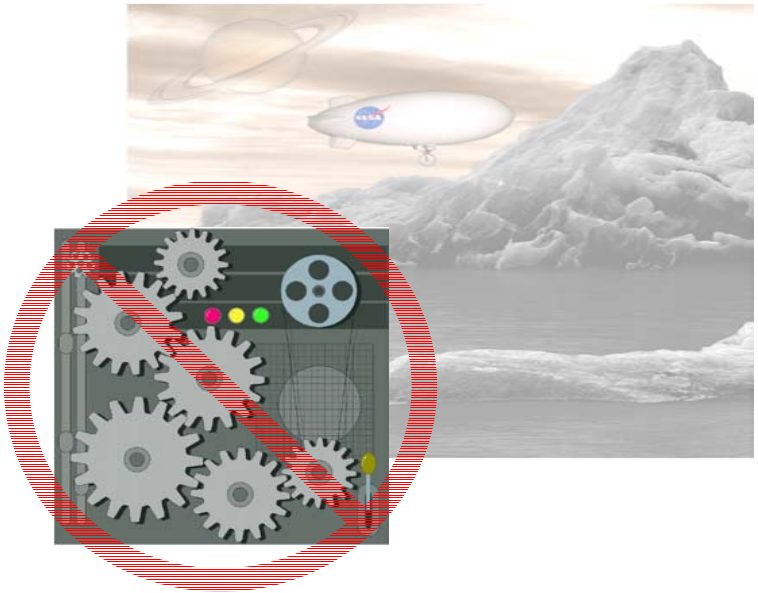
The Increased need for Software within NASA

- Systems must support **remote exploration**
- Systems must be **more autonomous**
- Systems must do **more complex tasks**

When people think of space, they think of rocket plumes and the Space Shuttle, but the future of space is information technology…

Daniel S. Goldin,

Previous NASA Administrator



No

SRI

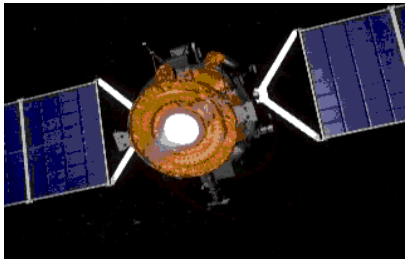
SRI

Hey Guys, this looks pretty good! I don't see any bugs!

The balance/combat is between:

- being complete (catching all bugs) and
- being able to scale (handle large programs).

ASE Group Program V&V History 1997-2003



Found 5 errors in the Remote Agent using the SPIN model checker. Program hand-translated to a model used by the SPIN model checker.

JPF 1

Model Checker that automatically translates Java code to a model used by the SPIN model checker. Translates 80% of Java.

Complete

JPF 2

Custom made model checker for Java programs.
Complete coverage of Java.

Complete

1997 1998 1999 2000

This talk

JPaX

Tool for monitoring temporal behavior and finding concurrency-errors (such as deadlocks and data-races) during execution of Java programs. Applies also to C and C++ programs

Scales

SRI

SRI

Dynamic Analysis

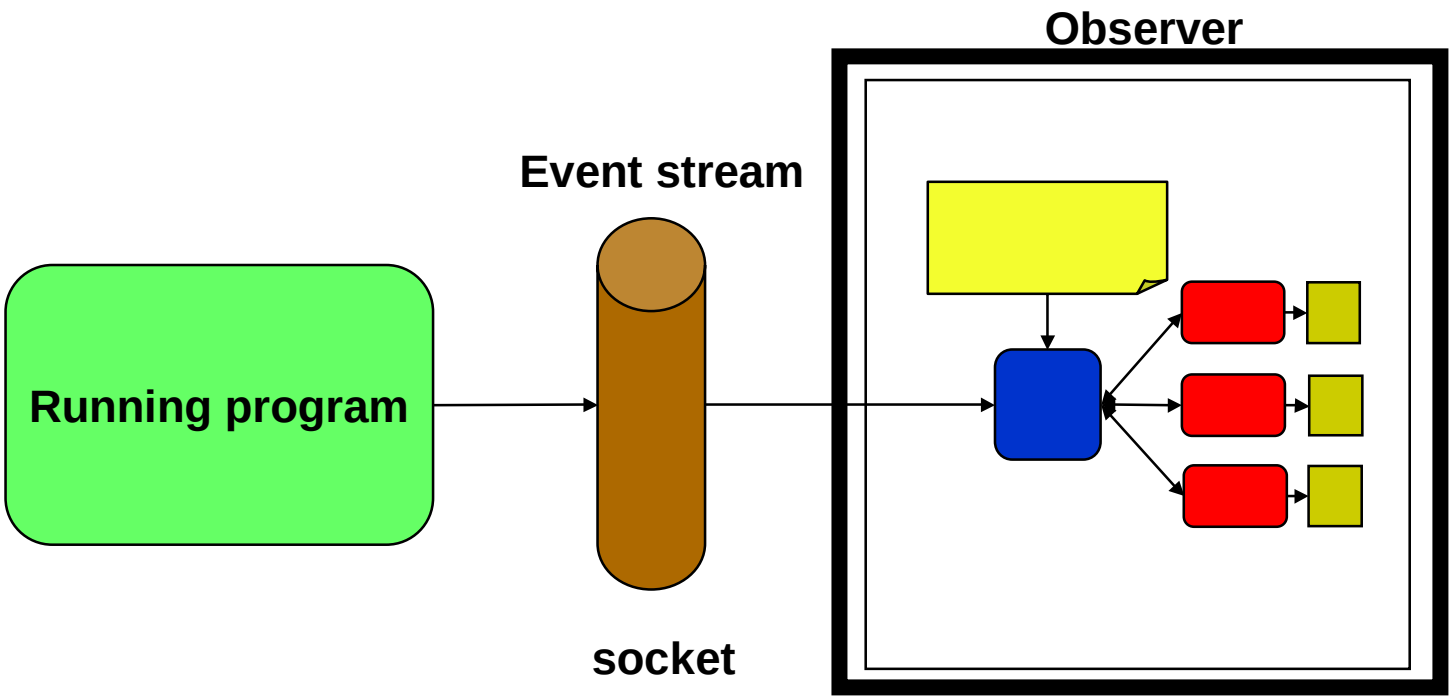
SRI

SRI

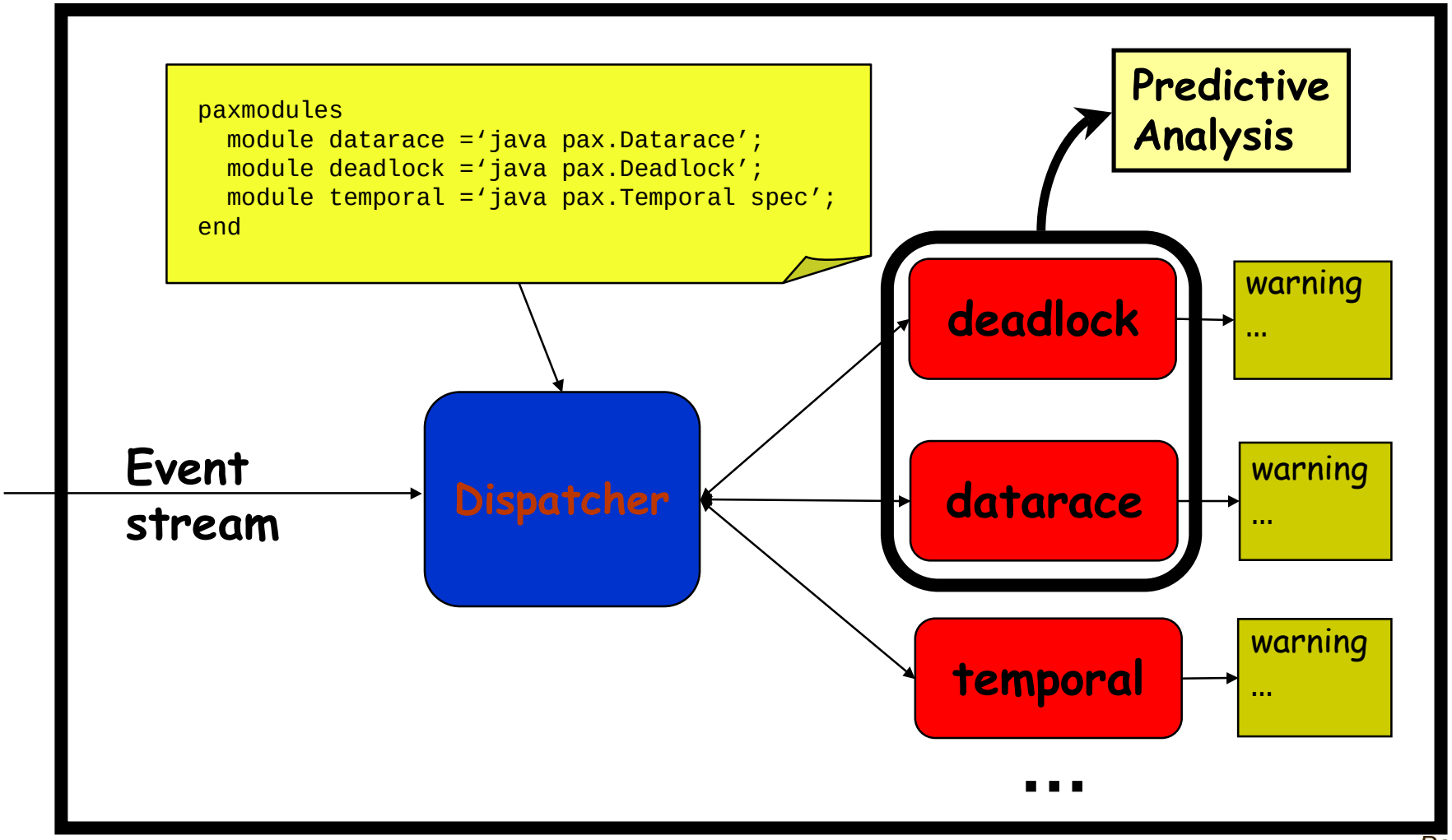
What is Dynamic Analysis

- Instrument program
- Run instrumented program
- Extract execution trace
- Analyze execution trace

PathExplorer



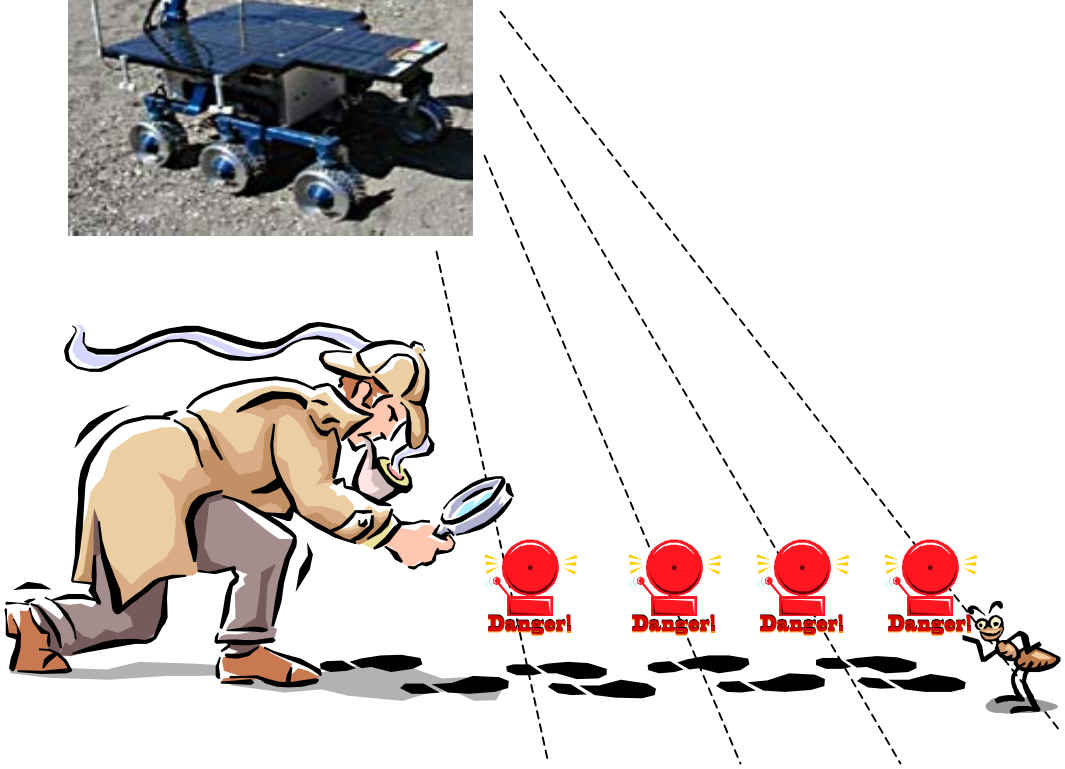
PathExplorer – the Observer



SRI

SRI

Looking for the Foot Prints Instead of for the Bug Itself



SRI

SRI

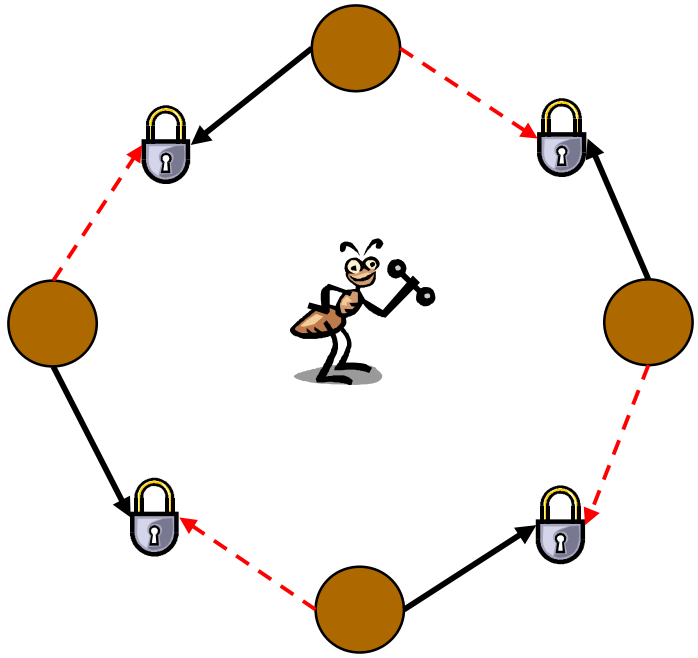
Deadlock Analysis

Two Kinds of Deadlocks

- Resource deadlocks
- Communication deadlocks

Resource Deadlocks

A resource deadlock can occur when two or more threads block each other in a cycle while trying to access synchronization locks (held by other threads) needed to continue their activities.



Java Program with Resource Deadlock

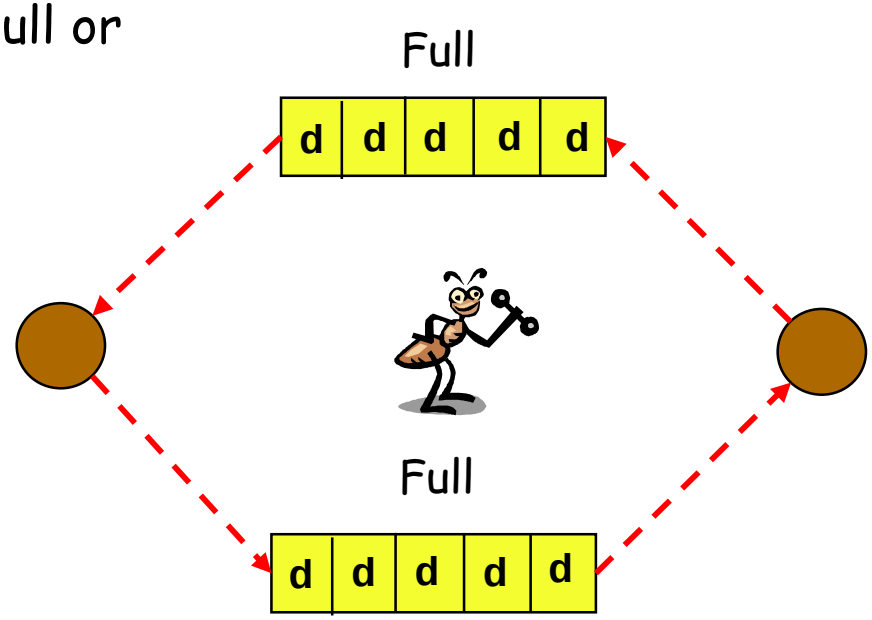
```
T1:

synchronized(R1){
   ..
   synchronized(R2){..}
   ..
}
```

```
T2:

synchronized(R2){
   ..
   synchronized(R1){..}
   ..
}
```

Deadlock: if T1 takes R1 and then T2 takes R2

Communication Deadlocks

A communication deadlock can occur when two or more threads block each other in a cycle while trying to communicate with each other and buffers are either full or empty.



We shall not consider communication deadlocks!

Code Analysis

```
class Main{
  Fork[] forks = new Fork[N];
  ..
  for(int i=0;i<N;i++){
    new Phisosopher(forks[i],forks[(i+1)%N];
  };
}
```

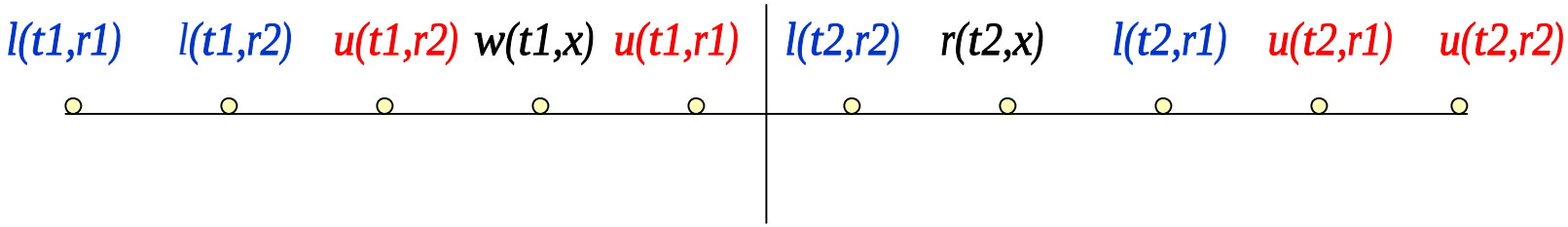
```
Philosopher:
  while(count<10){
    synchronized(left){
      synchronized(right){count++}
    }
  }
```

Static analysis *cannot find this problem due to the dynamic creation of forks and the '%' operator (experiment with JLint).*

Model checking *works for N=20, but if program is deadlock free (introducing gate lock) N=3 is max using 3 minutes (JPF).*

Alternative: Dynamic Analysis (Trace Analysis)

- Execute instrumented version of program and extract "random" execution trace:



T1:

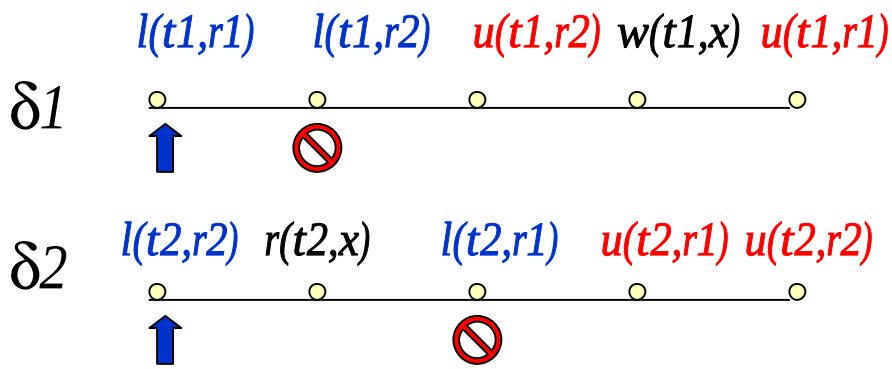
```
long x;
synchronized(R1){
  synchronized(R2){};
  x = big1*big2;
}
```

T2:

```
synchronized(R2){
  System.out.println(x);
  synchronized(R1){};
}
```

Model Checking Traces

- Project trace δ on each thread, obtaining δ1 and δ2
- Consider projections δ1 and δ2 as sequential programs
- Model check parallel composition of δ1 and δ2



- Deadlocking philosophers: for N=47 deadlock found in 5 minutes
- For ***deadlock free*** philosophers:
  - for **N=3** verified correct in 38 seconds
  - For **N=4** out of memory (1.5 GB memory)

SRI

SRI

Some Observations

- We can regard each projected trace as a transition system – and the parallel composition of these, called the composed transition system.
- We say that a trace is deadlock free if the composed transition system contains no deadlocks.

An execution trace can be regarded as an abstraction of the program. Hence similarities with other work on model checking abstractions.

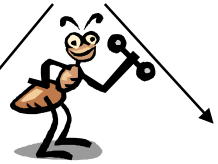
Testability: a Desirable Property

- Given a property φ, find a property ψ that is highly testable:
  - From verifying ψ on a single trace we extrapolate and conclude about φ on entire program.
- **Definition** (testable property in the ideal case) :
  - A property ψ is a testable property for φ if:
    - If $\square\delta\in P$. δ **satisfies** ψ then P **satisfies** φ
    - If $\square\delta\in P$. δ does not satisfiy ψ then P does not satisfiy φ
- Example:
  - φ = deadlock freedom
  - ψ = cycle freedom in a lock graph
- In practice we can only obtain approximations to testable properties: extrapolation with <u>high probability</u>.

Example

Guarded cycle

Deadlock cycle!



```
T1:

sync(G){
  sync(L1){
    sync(L2){}
  }
};
T3 = new T3();
j3.start();
J3.join();
sync(L2){
  sync(L1){}
}
```

```
T2:

sync(G){
  sync(L2){
    sync(L1){}
  }
}
```

```
T3:

sync(L1){
  sync(L2){}
}
```

Thread segmented cycle

Singular cycle

4 deadlock potentials
Only one is real

Execution Trace

```
T1:

sync(G){
  sync(L1){
    sync(L2){}
  }
};
T3 = new T3();
j3.start();
J3.join();
sync(L2){
  sync(L1){}
}
```

```
T2:

sync(G){
  sync(L2){
    sync(L1){}
  }
}
```

```
T3:

sync(L1){
  sync(L2)
}
```

Event format:

```
l(<thread>,<lock>)   - lock
u(<thread>,<lock>)   - unlock
s(<thread>,<thread>) - start
j(<thread>,<thread>) - join
```

Trace

```
l(T1,G)
l(T1,L1)
l(T1,L2)
u(T1,L2)
u(T1,L1)
s(T1,T3)
l(T2,G)
l(T2,L2)
l(T2,L1)
u(T2,L1)
u(T2,L2)
u(T2,G)
l(T3,L1)
l(T3,L2)
u(T3,L2)
u(T3,L1)
j(T1,T3)
l(T1,L2)
l(T1,L1)
u(T1,L1)
u(T1,L2)
```

Classical Algorithm

Algorithm: build lock graph and detect cycles in graph.
An edge goes from X to Y if a thread holds X while locking Y.

T1:

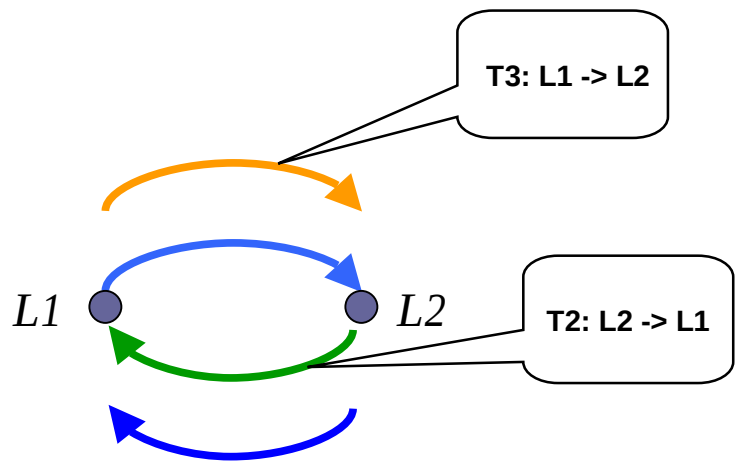
```
sync(G){
  sync(L1){
    sync(L2){}
  }
};
T3 = new T3();
j3.start();
J3.join();
sync(L2){
  sync(L1){}
}
```

T2:

```
sync(G){
  sync(L2){
    sync(L1){}
  }
}
```

T3:

```
sync(L1){
  sync(L2){}
}
```



4 cycles =
4 deadlock potentials reported (Visual Threads).

1 real deadlock! (3 false positives)

Guarded Algorithm

Algorithm: extend lock graph with labeled edges:
which thread, and set of guard locks.

T1:

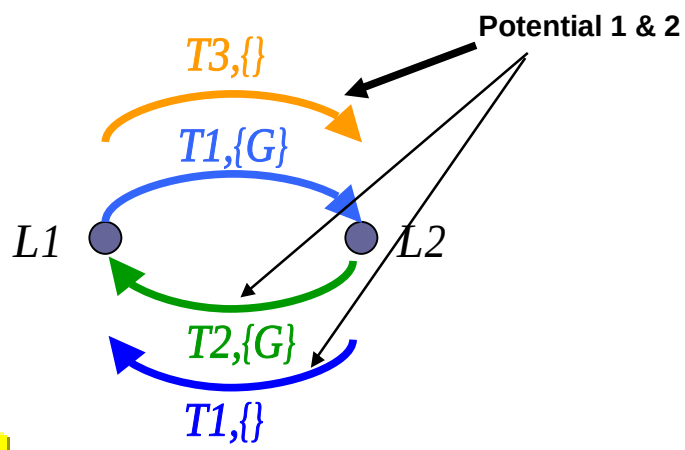
```
sync(G){
  sync(L1){
    sync(L2){}
  }
};
T3 = new T3();
j3.start();
J3.join();
sync(L2){
  sync(L1){}
}
```

T2:

```
sync(G){
  sync(L2){
    sync(L1){}
  }
}
```

T3:

```
sync(L1){
  sync(L2){}
}
```



Valid Cycles:

1. *Threads: must differ*
2. *Guard sets: must not overlap*

2 cycles =
2 deadlock potentials reported.

Segmented Algorithm

Algorithm: extend labels with segmentation information.

M:

```
• new T1().start();
• new T2().start();
```

T1:

```
sync(G){
  sync(L1){
    sync(L2){}
  }
};
T3 = new T3();
• j3.start();
• J3.join();
sync(L2){
  sync(L1){}
}
```

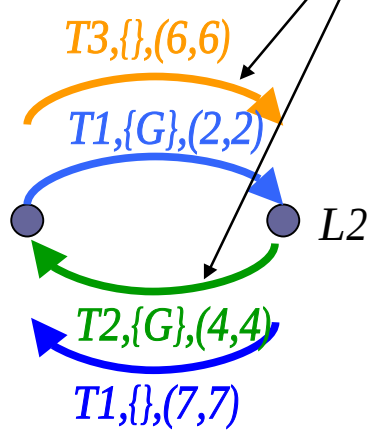
T2:

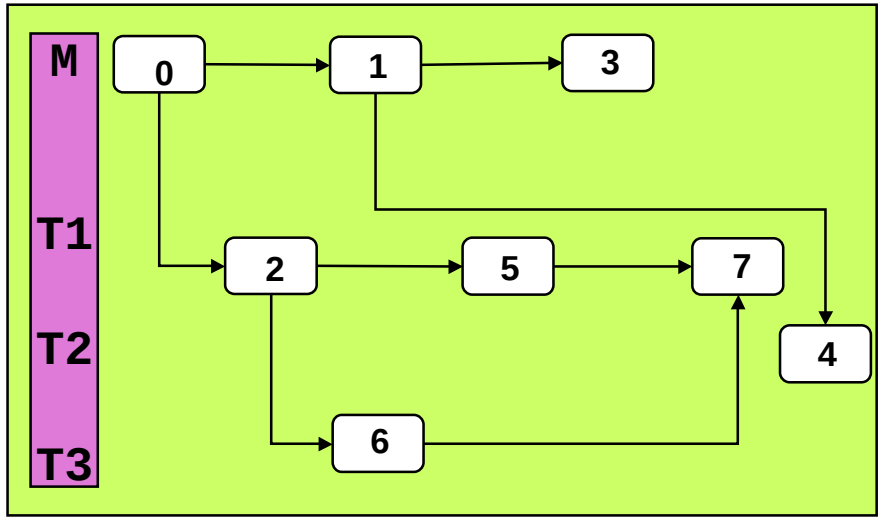
```
sync(G){
  sync(L2){
    sync(L1){}
  }
}
```

T3:

```
sync(L1){
  sync(L2){}
}
```

One potential left, the real deadlock!





Valid Cycles:

1. *Threads: must differ*
2. *Guard sets: must not overlap*
3. *Segments: must be parallel*

SRI

Correctness

Theorem:

For every deadlocking trace in the composed transition system, obtained by putting projected traces in parallel, there is a valid cycle that reflects it.

For every valid cycle there is a deadlocking trace in the composed transition system.

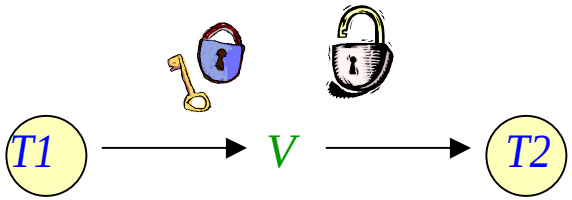
SRI

SRI

Data Race Analysis

Data Races

A datarace can occur between a set of 2 threads T={T1,T2} if they access a shared variable V and there is a state of the execution where T1 can write to the variable and T2 can write to or read from the variable without using a shared lock.



Java Program with Data Race

```
T1:

synchronized(R1){
  sum = sum + 100;
}
```

```
T2:

synchronized(R2){
  sum = sum + 50;
}
```

Data race: if T1 reads sum and then T2 reads sum where after the additions are made. Result is +100 or +50, depending on who writes first, not +150

Data Race Algorithm

- *Developed by Savage et al. (Compaq) and used in Visual Threads for Compaq platforms to work on C. Requires Compaq hardware to run. Here made platform independent and modified to work for Java.*

```
T1:

synchronized(R1){
  sum = sum + 100;
}
```

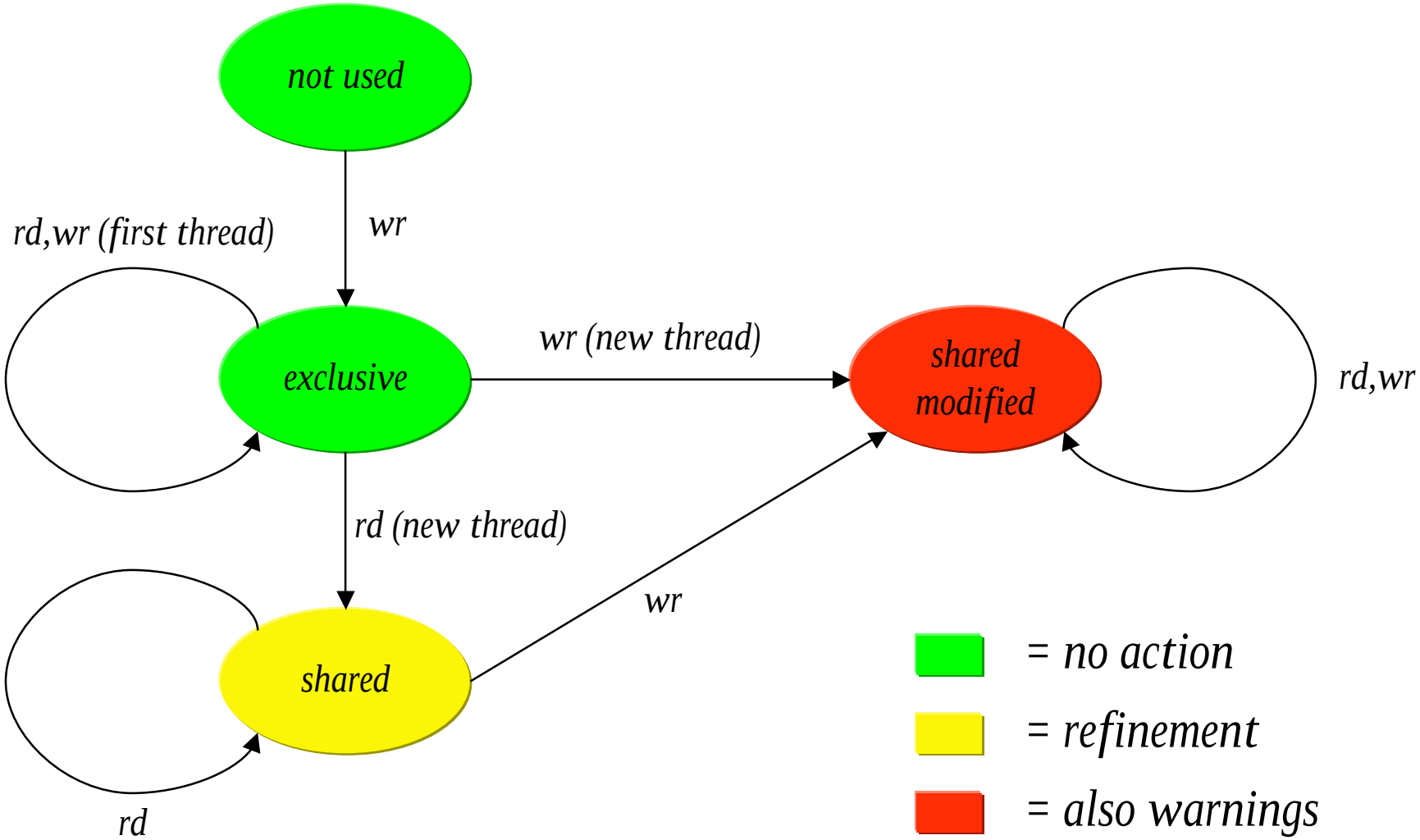
```
T2:

synchronized(R2){
  sum = sum + 50;
}
```

Initially: Lockset = {}

T1 executes: Lockset = {R1}

T2 executes: Lockset = Lockset ∩ {R2} = {}

Reducing False Positives
Associate Automaton with Variable



SRI

SRI

High-Level Data Race Analysis

High-Level Data Races
An Example

```
class CoordinatePair{
  int x = 0;
  int y = 0;
  int getX(){return x;}
  int getY(){return y;}
  void setX(int v){x = v;}
  void setY(int v){y = v;}
}
```

The two threads have <u>inconsistent views</u> on the granularity of synchronization..
Thread 1 accesses x and y in one block.
Thread 2 accesses x and y in two blocks,

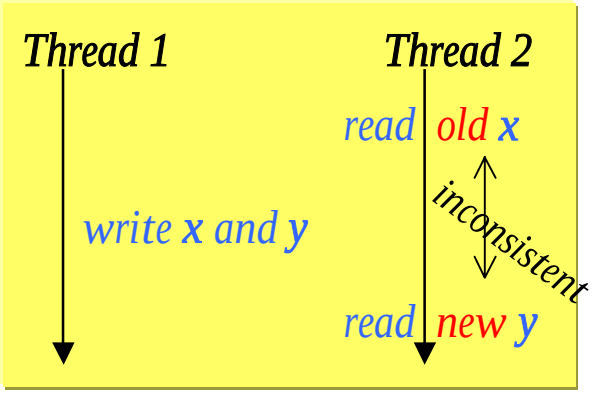
```
synchronized(lock){
  c.setX(1);
  c.setY(2);
}
```

Thread 1

```
synchronized(lock){
  my_x = c.getX();
}
```

```
synchronized(lock){
  my_y = c.getY();
}
```

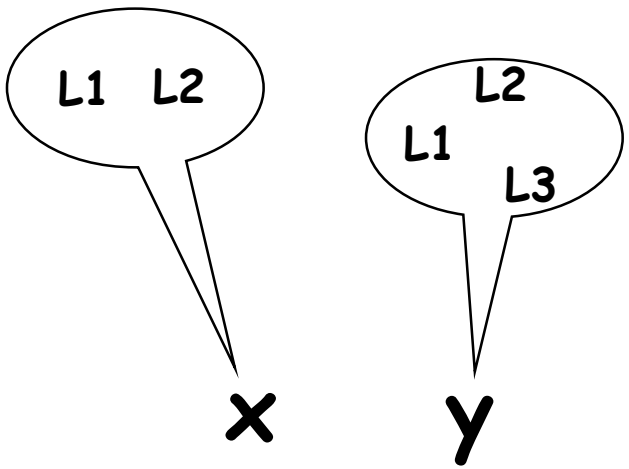
Thread 2



Low-Level versus High-Level Data races

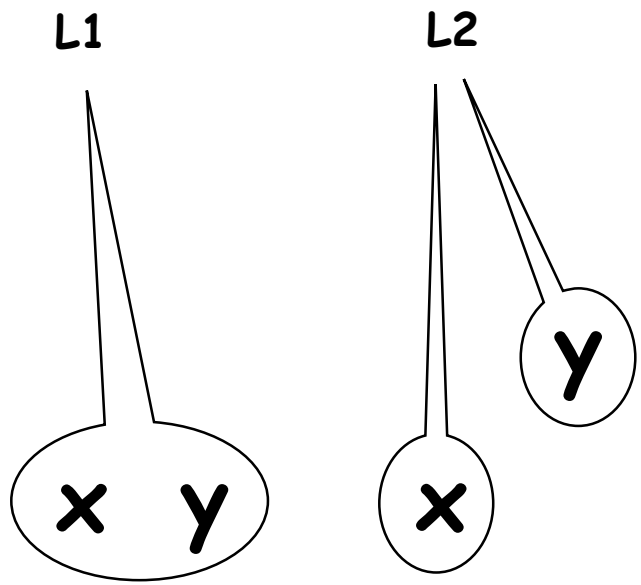
Low-Level

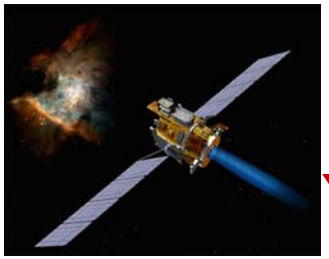
For each variable: a lock set



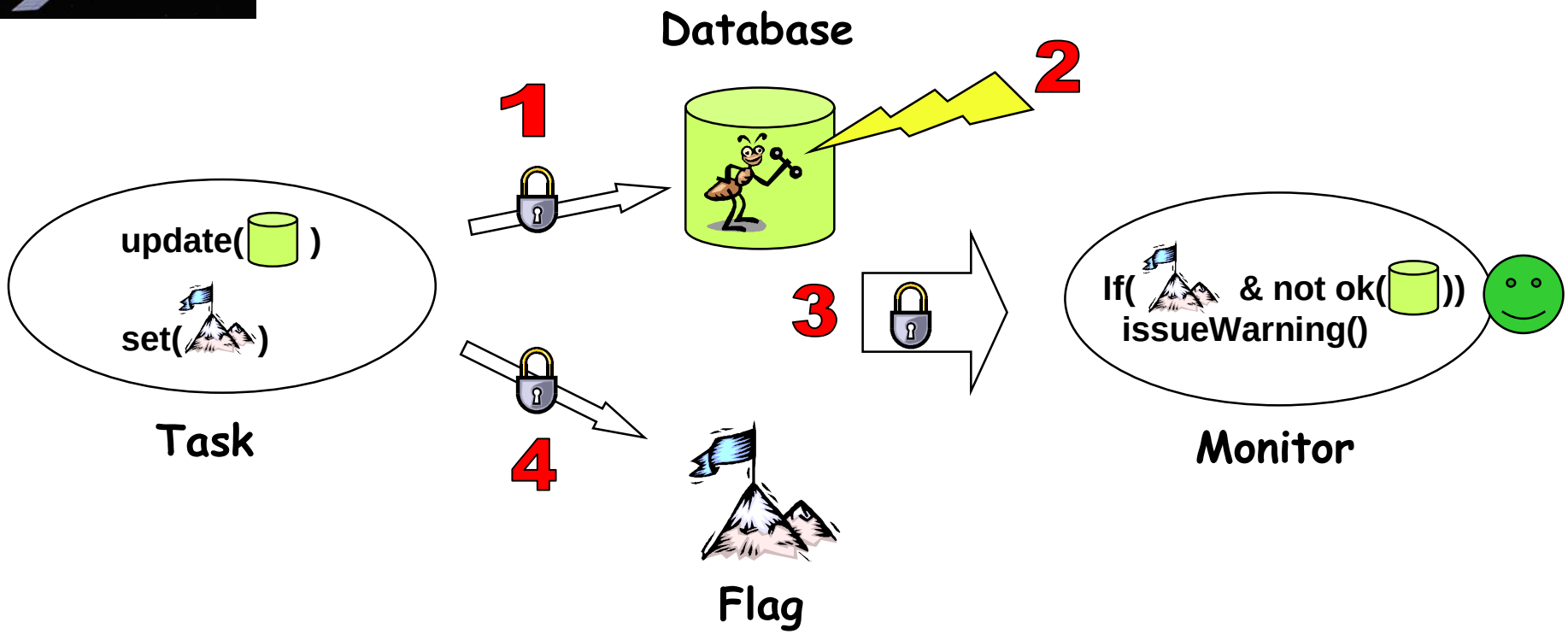
High-Level

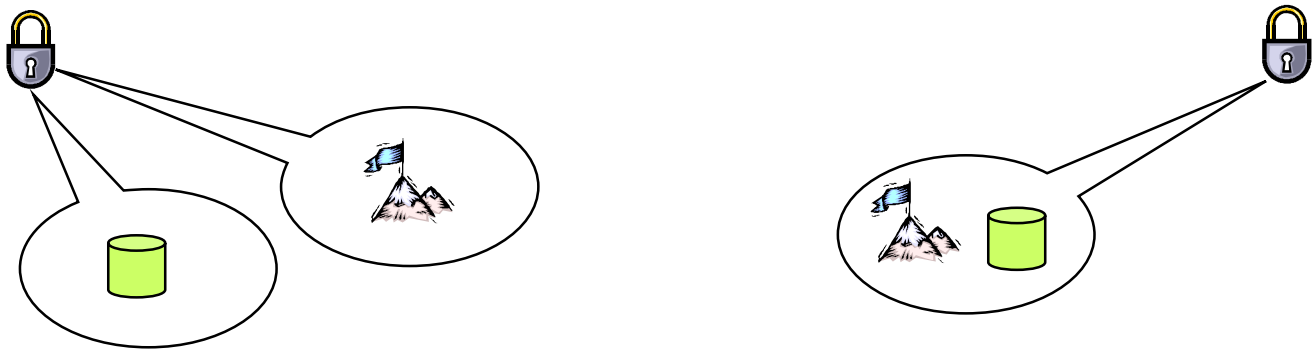
For each lock: a variable set (several)





Higher Level Data Race in Remote Agent

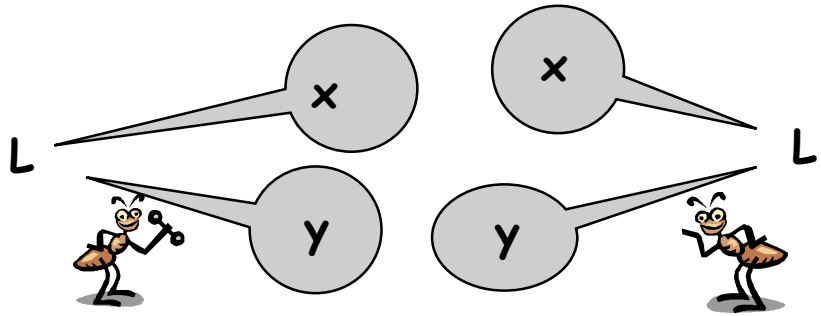




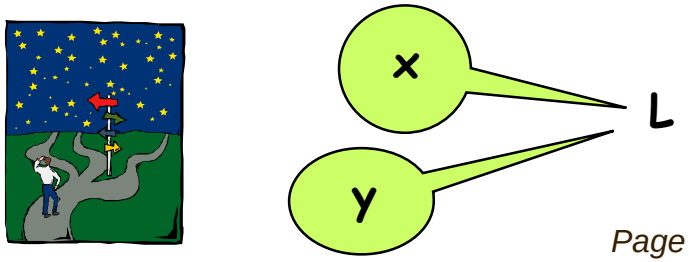
Soundness & Completeness

- **False positive when:**
  - One thread uses coarser locking that required due to efficiency.

- **False negatives when:**
  - All threads use the same locking



  - Random execution trace does not expose problem



More Examples

Thread 1	Thread 2	View Consistency
x y	x y	
x y x y	x y	
x y	x	
x y	x y x	
x y	x y	

View Consistency

All views overlapping with another thread's maximal View must be subsets of each other.

<u>View Consistency:</u>

$\forall$ threads t1 and t2 •
let m be a maximal view of t1,
let v1 and v2 be two views of t2,
let $v1' = m \cap v1$,
let $v2' = m \cap v2$,
then $v1' \subseteq v2' \vee v2' \subseteq v1'$.

Examples Explained

Thread 1	Thread 2	View Consistency
x y	x y	$\{x\} \not\subseteq \{y\}$
x y x y	x y	$\{x\} \not\subseteq \{y\}$
x y	x	
x y	x y x	$\{x\} \subseteq \{x,y\}$
x y	x y	

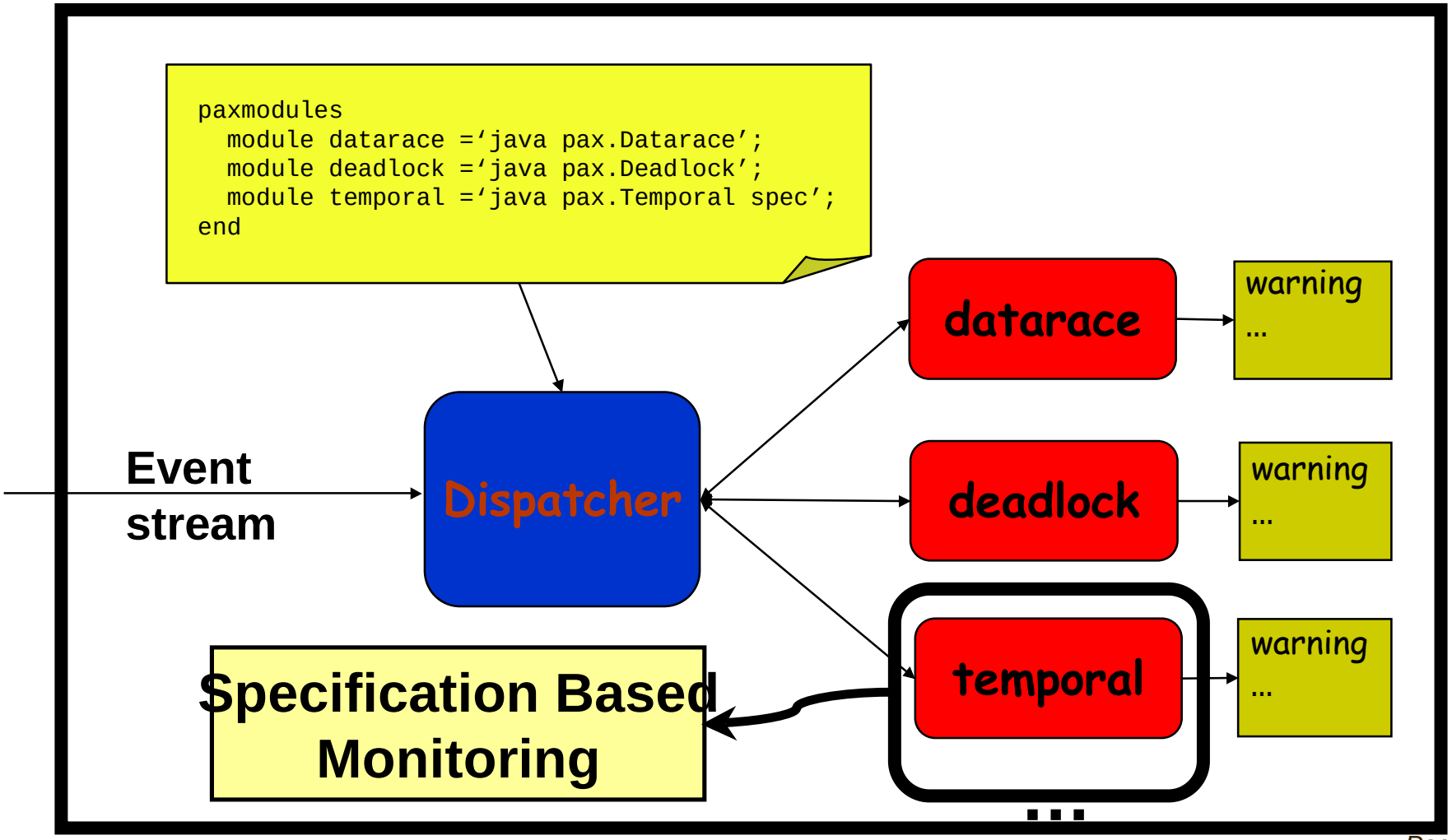
$\not\subseteq$

SRI

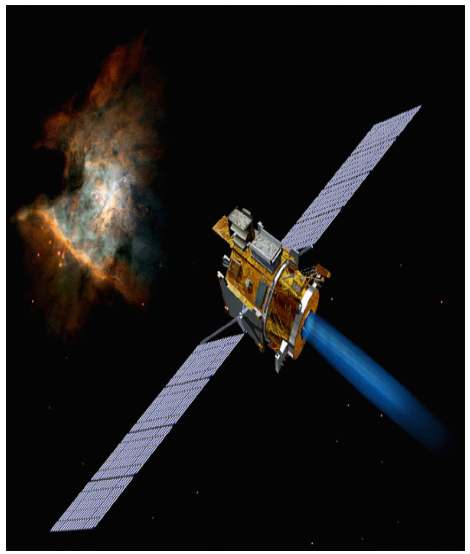
SRI

Temporal Logic Monitoring

PathExplorer – the Observer



Runtime Verification with Java PathExplorer



always(A -> **not** C **until** B)

Formalized temporal requirement

Translates into observer

observer

events

A B C A B A C

Temporal logic facilitates expression of requirements that relate a vehicle's states at different time points.

Maude as a Specification Language

- **Specification and verification system**

  In the OBJ family

- **Algebraic specification**

  Signatures + equations

- **Mixfix notation**

  Easy to define syntax of new logics.

- **Fast rewriting**

  Efficient semantics for monitoring.

Future Time Temporal Logic in Maude

Syntax – *Propositional Calculus plus*

o F (next) □ F (always) ◇ F (eventually) F U F′ (until)

Executable Semantics – *Rewriting*

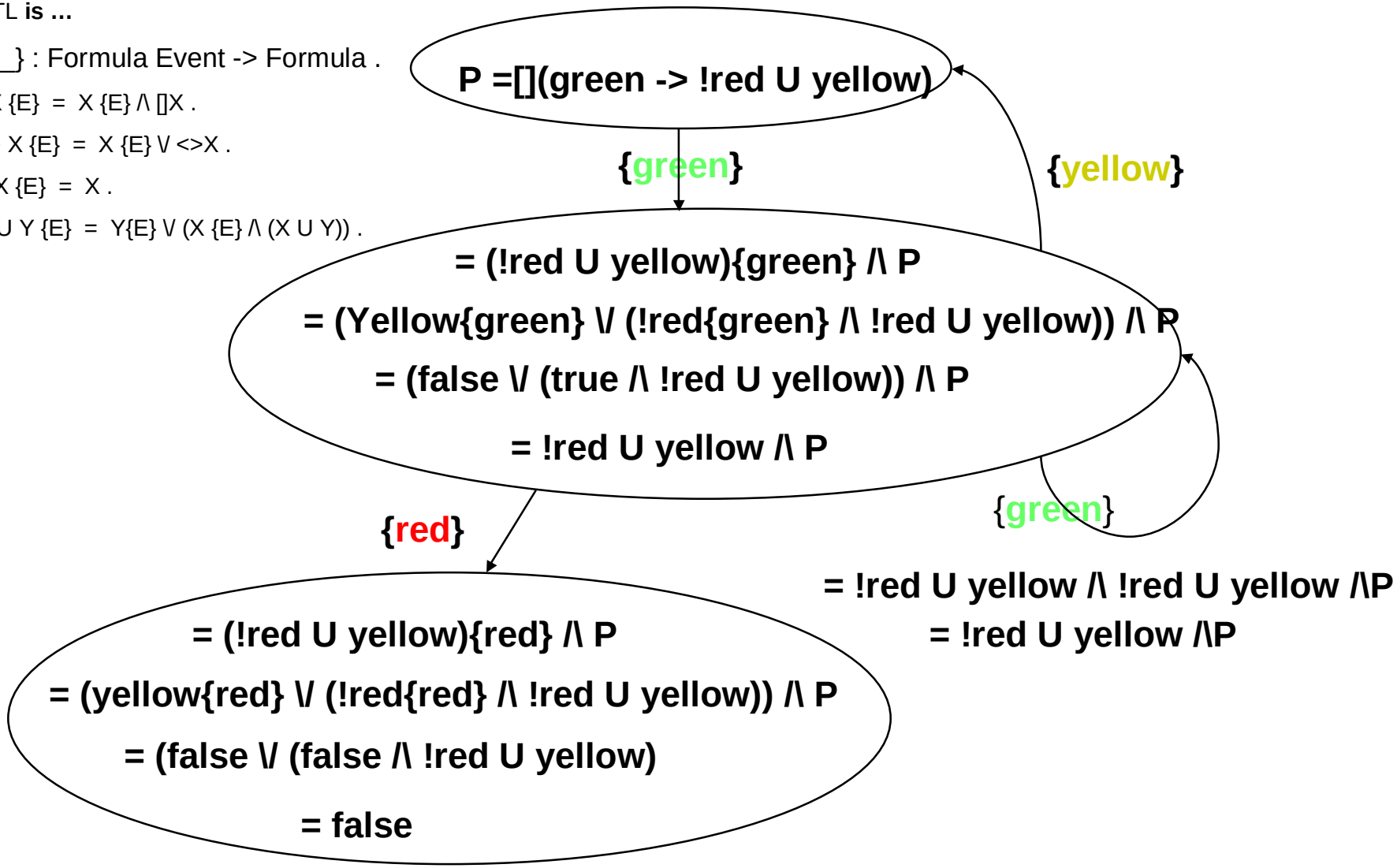
_{_} : Formula × Event -> Formula (*"consume" event e*)

F{e} *is formula that should hold after processing e*

p{e}	⇨	is the atomic predicate p true on e ?
(F *op* F′){e}	⇨	F{e} *op* F′{e}
(o F){e}	⇨	F
(□ F){e}	⇨	F{e} ∧ (□ F)
(◇ F){e}	⇨	F{e} ∨ (◇ F)
(F U F′){e}	⇨	F′{e} ∨ (F{e} ∧ (F U F′))

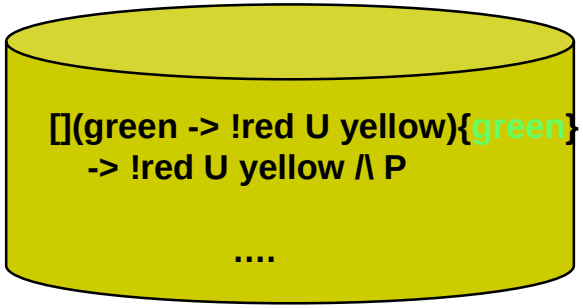
Example

```
fmod LTL is ...
  op _{_} : Formula Event -> Formula .
   eq []X {E}  =  X {E} /\ []X .
   eq <> X {E}  =  X {E} \/ <>X .
   eq o X {E}  =  X .
   eq X U Y {E}  =  Y{E} \/ (X {E} /\ (X U Y)) .
endfm .
```

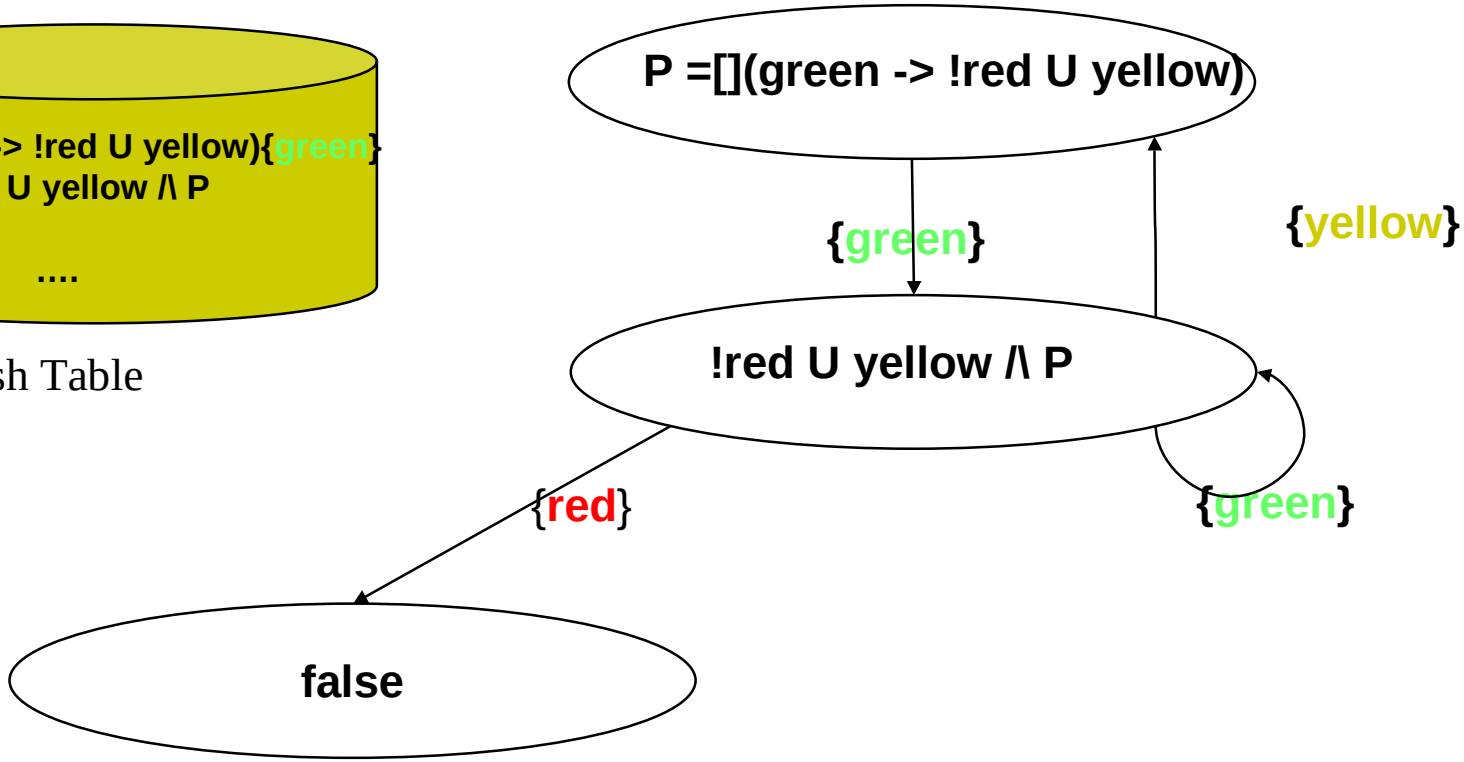


Using "memo" to Generate Automata Specialized to Trace

```
fmod LTL is
   ...
   op _{_} : Formula Event -> Formula [memo] .
   ...
endfm .
```



Hash Table



SRI

SRI

Code Instrumentation

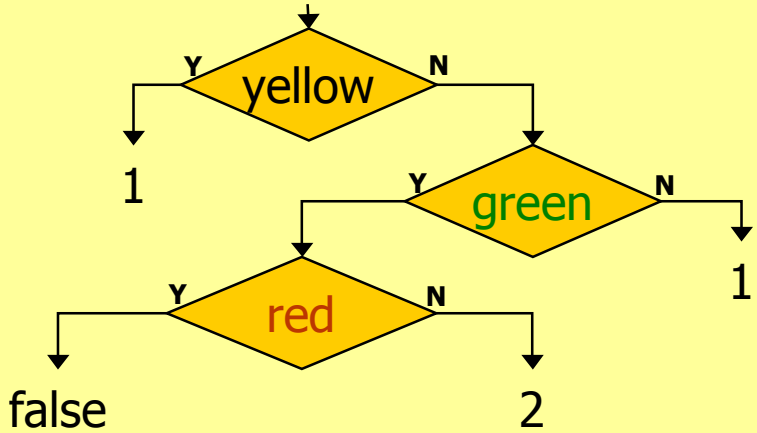
- Jtrek used for Java byte-code instrumentation:
  - Gives access to class files as abstract syntax trees
  - Allows to browse bytecode instructions and to insert new event transmitting instructions.
  - Instrumentation is automated guided by instrumentation scripts telling what to instrument.
- Instrumentation package built on top of Jtrek:
  - Driven by instrumentation script, which is a set of instrumentation rules (condition,action), where
    - Conditions are predicates on the static structure of the program
    - Actions are indications of what kind of information we want to observe

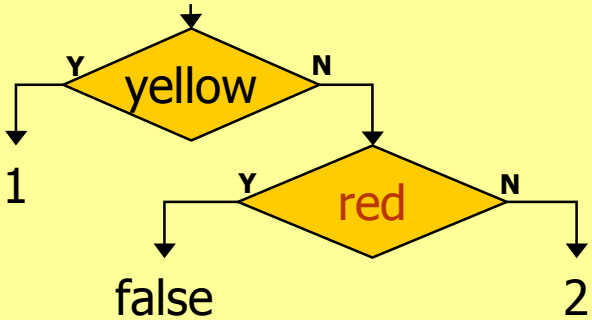
Generating FSM Observers

- There are applications where
  - Little monitoring overhead is allowed (real time)
  - Few resources available for monitoring
- Challenge: efficient and simple monitors!
- *Finite State Machine* observers can be built from formulae *before* monitoring
  - Lower runtime overhead
    - No inferences needed
    - Only some atomic predicates need to be evaluated
  - Higher start time overhead

Monitoring Future Time Temporal Logic using Binary Transition Trees (BTT)

Formula	$\Box$(green $\rightarrow$ $\neg$red U yellow)	
State	1	2
BTT	yellow ? 1 : green ? (red ? false : 2) : 1	yellow ? 1 : (red ? false : 2)





Building a Minimal BTT_FSM

- Idea
  - Do the rewrites for *all* possible values of predicates
  - Get a finite state machine
    - Nodes are LTL formulae
    - Optimize using a validity checker ($F \leftrightarrow F'$: one state)
    - Edges are propositions
    - Assign numbers to states
    - Replace edges by *Binary Transition Trees*
- Exponential, but works fine in practice

Use of BTT FSM Generation

Inline version of JPaX expands comments like

```
/* JPaX: after green comes yellow
    Atom red = tlc.state.getColor() == 1;
    Atom green = tlc.state.getColor() == 2;
    Atom yellow = tlc.state.getColor() == 3;

    Formula : []( green -> ( ! red U yellow ));
*/
```

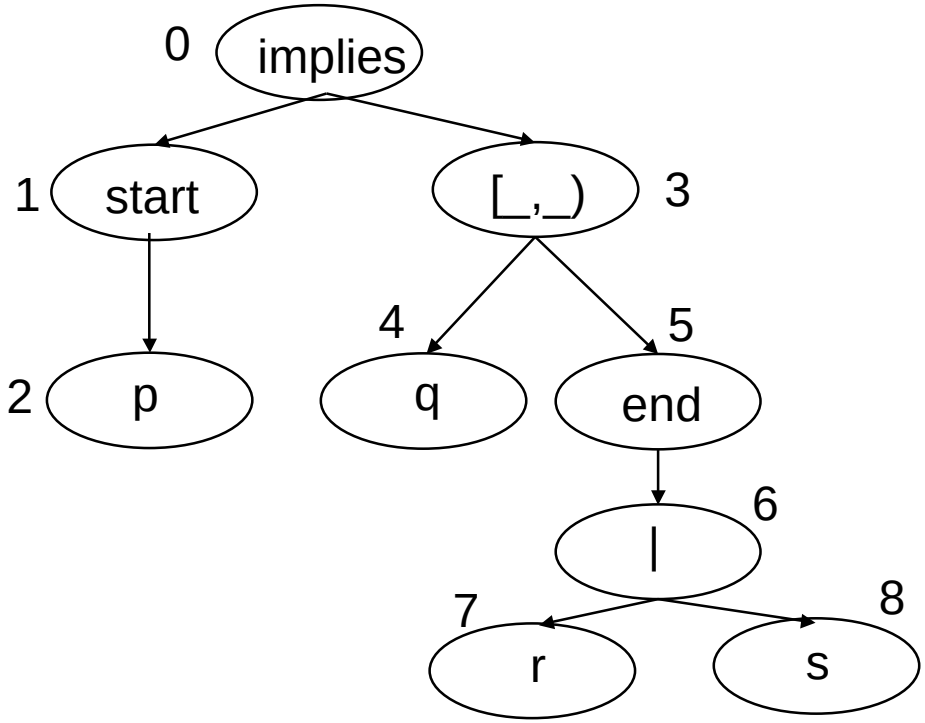
into code that throws an exception when formula is violated

```
try{
  switch(bttFsmState) {
     case -1: break;
     case  1: bttFsmState = tlc.state.getColor() == 3 ? 1 :
                            tlc.state.getColor() == 2 ?
                            tlc.state.getColor() == 1 ? 0 : 2 : 1; break;
     case  2: bttFsmState = tlc.state.getColor() == 3 ? 1 :
                            tlc.state.getColor() == 1 ? 0 : 2; break;
  }
  if(bttFsmState == 0) throw new Exception("Prop. Failure");
}
catch(Exception e){ ... };
```

Past Time Temporal Logic

```
specification
  P = start(p) -> [q,end(r|s)
end
```

F ::= true | false | Id
| not F | F implies F | F and F | F or F | (F)
| [F,F) | start(F) | end(F)




```
boolean pre[9];
boolean now[9];

now[2] = holds(p);
now[1] = now[2] && !pre[2];
now[4] = holds(q);
now[7] = holds(r);
now[8] = holds(s);
now[6] = now[7] || now[8];
now[5] = !now[6] && pre[6];
now[3] = (pre[3] || now[4])
            && !now[5];
now[0] = !now[1] || now[3];
if(now[0]==0)print("error");
pre = now;
```

Result of Translation is the "Formulae" Java Class

```
public class Formulae{
  abstract class Formula{
    protected String name;
    protected boolean[] pre;
    protected boolean[] now;
    protected State state;

    public Formula(String name,State state){
      this.name = name; this.state = state;
    }

    public string getName(){return name;}

    public abstract boolean evaluate();
  }

  private List formulae = new ArrayList();

  …
```

The "Formulae" Class

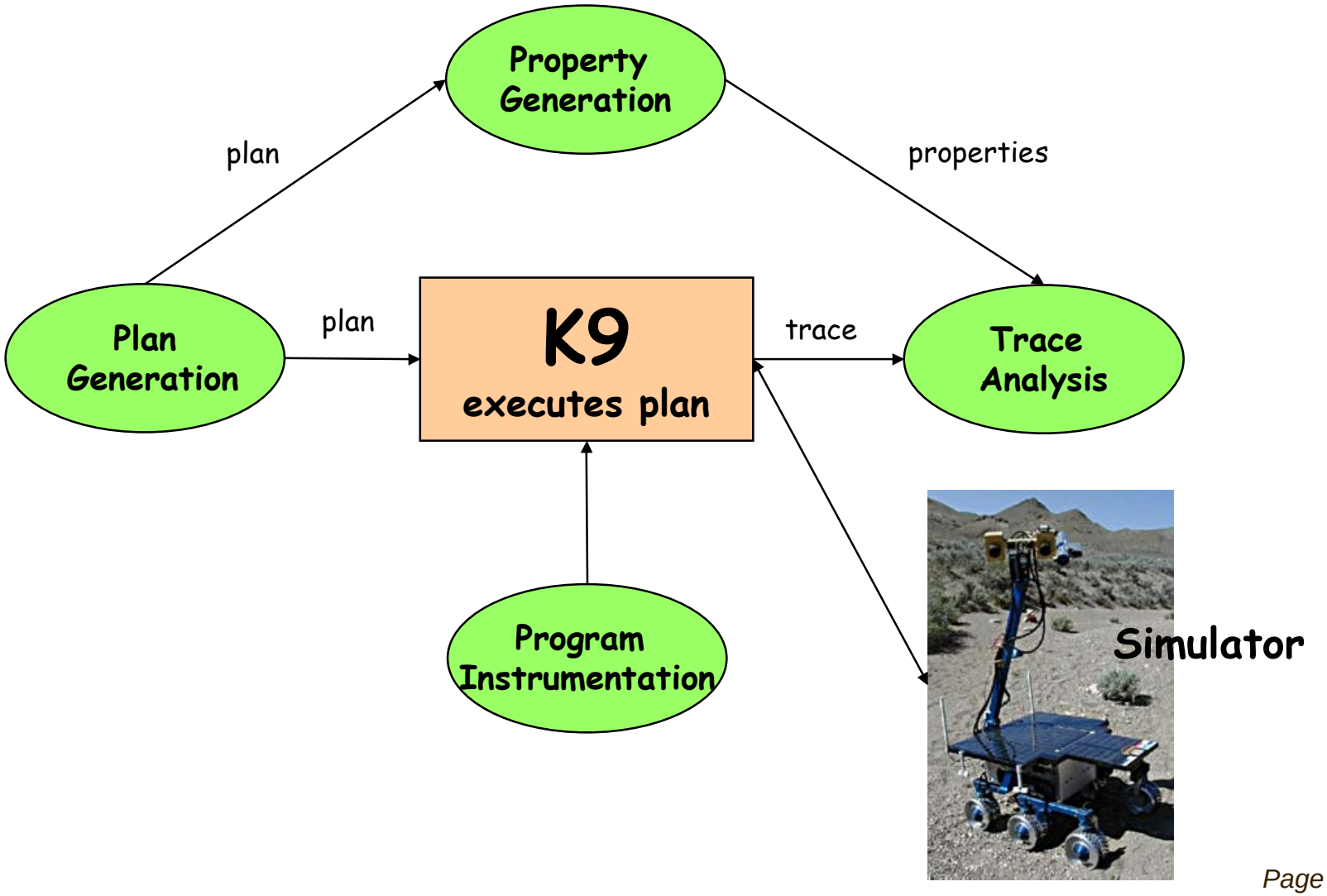
```
  public void evaluate(){
    Iterator it = formulae.iterator();
    while(it.hasNext()){
      Formula formula = (Formula)it.next();
      if(!formula.evaluate())
        System.out.println("Property " + formula.getName() + " violat
    };
  }

  class Formula_P extends Formula{
    …
  }

  public Formulae(State state){
    formulae.add(new Formula_P(state));
  }
}
```

The "Formulae" Java Class

```
class Formula_P extends Formula{
  public boolean evaluate(){
    now[2] = state.holds("p");
    now[1] = now[2] && !pre[2];
    now[4] = state.holds("q");
    now[7] = state.holds("r");
    now[8] = state.holds("s");
    now[6] = now[7] || now[8];
    now[5] = !now[6] && pre[6];
    now[3] = (pre[3] || now[4]) && !now[5];
    now[0] = !now[1] || now[3];
    System.arraycopy(now,0,pre,0,9);
    return now[0];
  }

  public Formula_P(State state){
    super("P",state);
    pre = new boolean[9];
    now = new boolean[9];
  }
}
```

Test Case Generation and Dynamic Analysis

Automated Test Environment for the Planetary Rover K9



Plan Grammar

Plan → *Node*

Node → *Block* | *Task*

Block → (block *NodeAttr*
:node-list (*NodeList*))

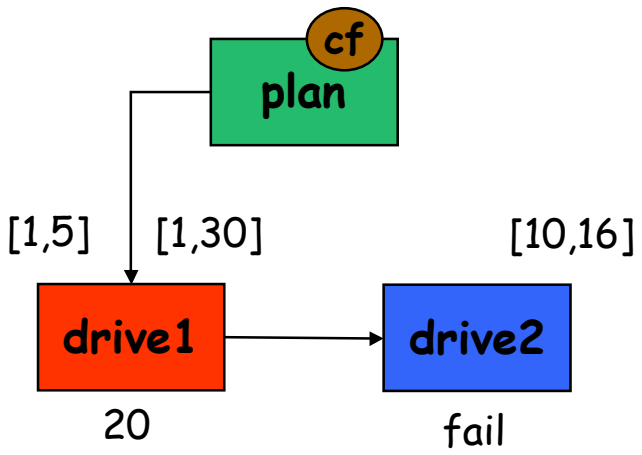
NodeList → *Node NodeList* | ε

Task → (task *NodeAttr*
:action *Symbol*
[:fail]
[:duration *DurationTime*])

NodeAttr → :id *Symbol*
[:start-condition *Condition*]
[:end-condition *Condition*]
[:continue-on-failure]

Condition → (time *StartTime EndTime*)

Example of Plan




```
(block :id plan
        :continue-on-failure
        :node-list (
         (task :id drive1
                :start-condition (time +1 +5)
                :end-condition (time +1 +30)
                :action BaseMove1
                :duration 20
         )
         (task :id drive2
                :end-condition (time +10 +16)
                :action BaseMove2
                :fail
         )
        ) )
```

Plan Generation

```
class UniversalPlanner { …
  static int nNodes = 0;
  static int tRange = 10;

  static void Plan(int nn, int tr) {
    nNodes = nn; tRange = tr;
    Node plan = UniversalNode();
    print(plan);
    assert(false);
  }

  static Node UniversalNode() {
    if(nNodes == 0) return null;
    if( Verify.chooseBool() ) return null;
    if( Verify.chooseBool() )
      return UniversalTask();
    return UniversalBlock();
  }

  static NodeAttr UniversalNodeAttr() { … }
```

```
static Node UniversalTask() {
    NodeAttr na = UniversalNodeAttr();
    Symbol action = new Symbol();
    boolean fail = Verify.randomBool();
    int duration = Verify.random(tRange);
    Task t = new Task(na, action, fail, duration);
    nNodes--;
    return t;
}
static Node UniversalBlock() {
    NodeAttr na = UniversalNodeAttr();
    nNodes--;
    ListOfNodes l = new ListOfNodes();
    for(Node n=UniversalNode(); n!=null; n=UniversalNode())
      l.addNode(n);
    Block b = new Block(na,l);
    return b;
}
```

Plan Properties

◊ start(plan)

□ (start(plan) → $\Diamond_{1,5}$ start(drive1))

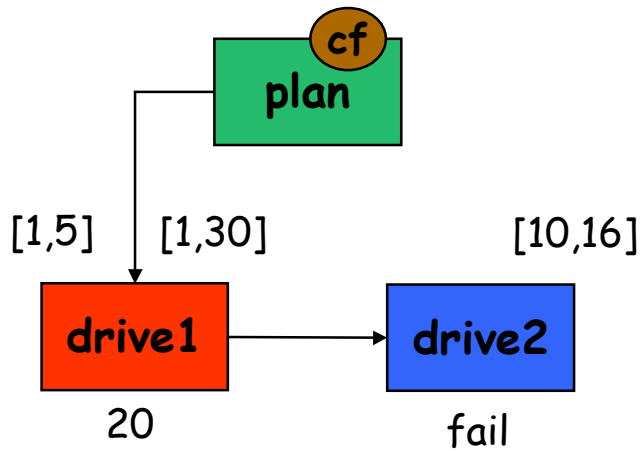
□ (start(drive1)→($\Diamond_{1,30}$success(drive1) ∨ ◊fail(drive1)))

□ (success(drive1) → ◊ start(drive2))

□ (end(drive2) → ◊ success(plan))

◊ success(drive1)

◊ fail(drive2)



"Demo" of K9-Explorer

Workshop

Third International Workshop on

Runtime Verification

RV'03

CAV'03
July 13, 2003
Boulder, Colorado
USA

http://www.cis.upenn.edu/rv2003/

Workshop

Fourth International Workshop on

Runtime Verification

RV'04

ETAPS'04
April 3-4, 2004
Barcelona
Spain

SRI

SRI

Aspect Oriented Programming for Testing and Debugging

Robert E. Filman Klaus Havelund

RIACS Kestrel Technology

rfilman@mail.arc.nasa.gov havelund@email.arc.nasa.gov

NASA Ames Research Center

Moffett Field, CA 94035 U.S.A.

Static and Dynamic Quantification

- In earlier papers, distinction 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
- Coming to the belief 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.

SRI

SRI

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

SRI

SRI

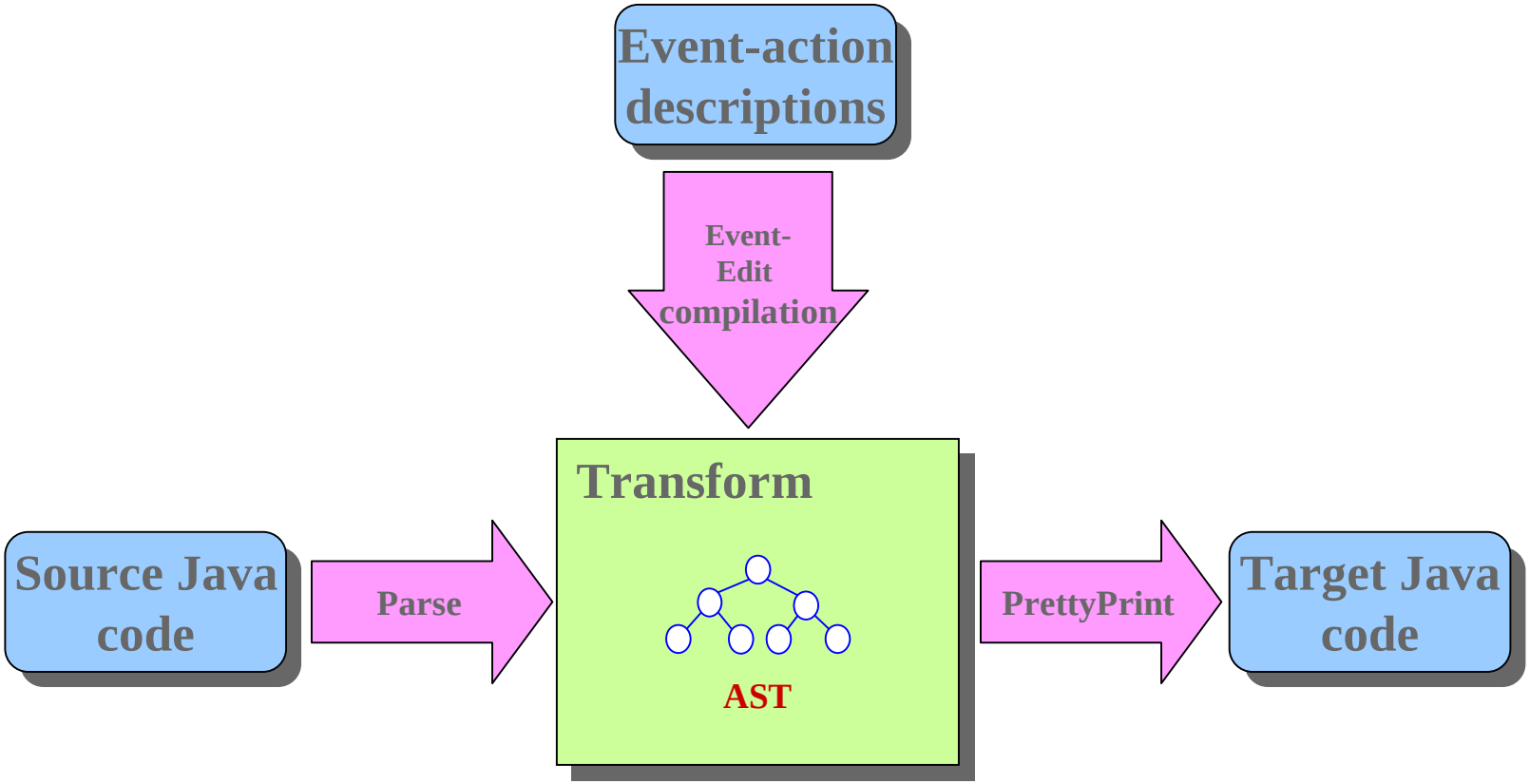
Research regime

- Define a language of events and actions on those events.
- Determine how each event is reflected (or can be made visible) in source code.
  - Its shadow
- 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



SRI

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.

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);
  }
}
```

SRI

SRI

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
 }
}
```

SRI

SRI

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.

An Evaluation

SRI

SRI

FSW Verification Tool Comparison

10 Oct 2002

Peter R. Glück

Jet Propulsion Laboratory

SRI

SRI

User Impressions

Tool	JPAX	JPF	Polyspace	DBRover
Impressions	- A wonderful tool! One of the most helpful tools seen in a long time. - All of the problems found were the type that would have taken months to find because they were intermittent. - Found stuff we didn't know about tha timproved the robustness of the application (with less than 4 hours total time invested).	- Easy to create a dumb, non-deterministic environment, but it may be harder to filter out unrealistic scenarios in a real project. - You have to manually put in atomic regions, which may not be trival in a real, multi-threaded application. - It takes some amoun of manual specification, which should be done at the requirements stage.	- The main hurdle is setting it up. The rest i pretty nice. - Liked the GUI and the display capabilities - Color coding and filtering are good. - Very easy to use once configured.	- If you can formally specify a FSW property in LTL and are able to inject it into the application code, then it excels as a very powerful run-time assert statement. - The simulator was essential. Without it it would have been much harder to validate the correctness properties This was a time-saver. - It is nice that it monitors multiple correctness properties simultaneously.