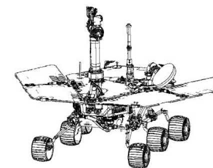
KESTREL TECHNOLOGY

Verimag

Scalable Dynamic Deadlock Analysis of Multi-Threaded Programs

I see a **deadlock** potential in that execution

log file
lock(T1,L1)
lock(T1,L2)
release(T1,L2)
release(T1,L1)
lock(T2,L2)
lock(T2,L1)



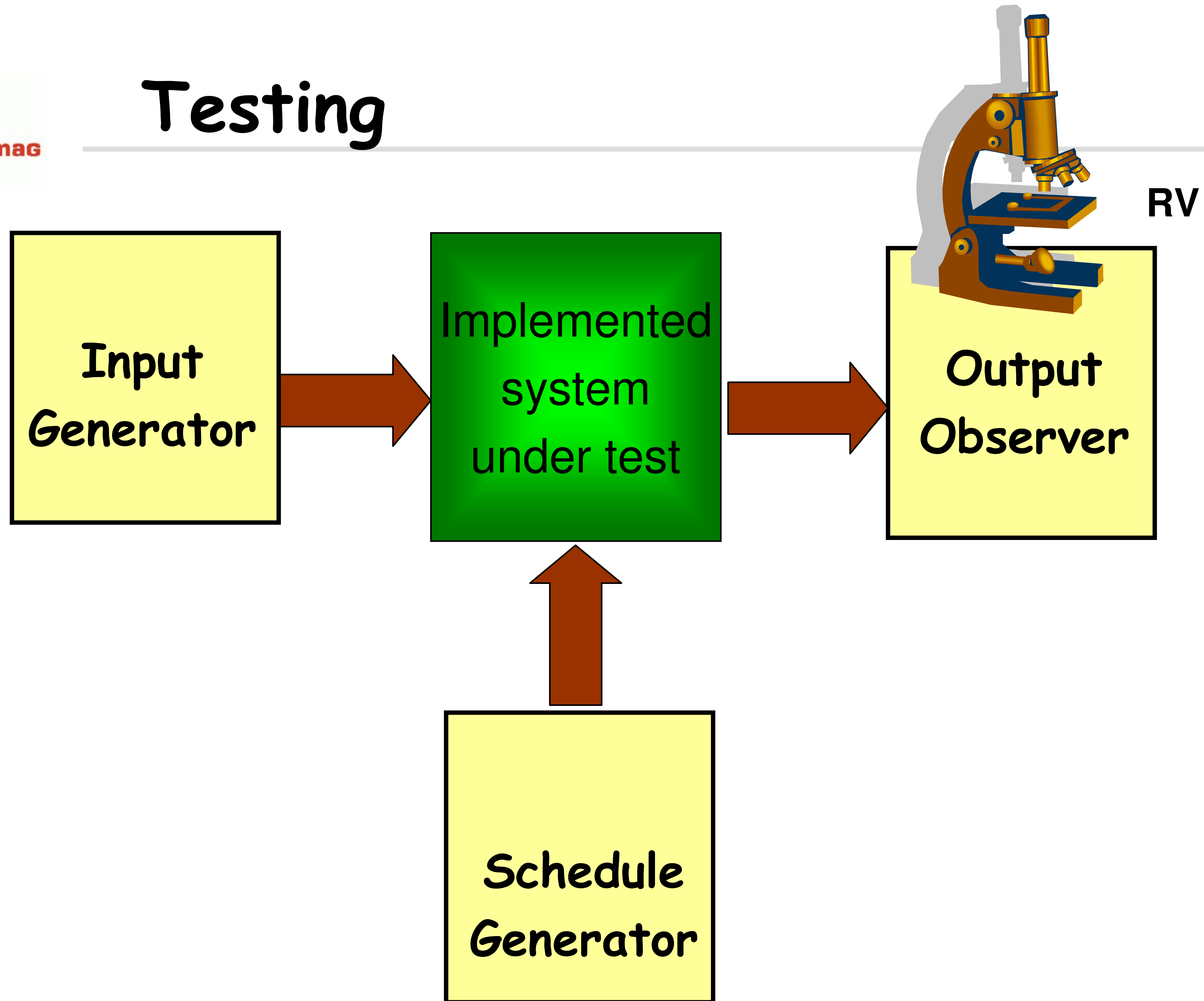
Saddek Bensalem

Verimag, Grenoble, France

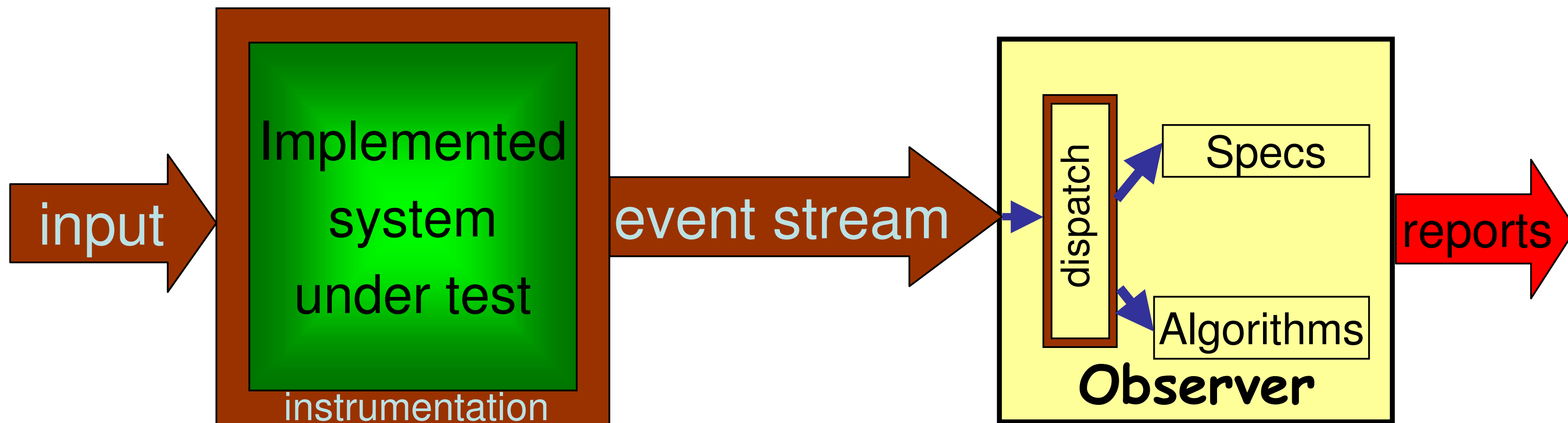
Klaus Havelund

Kestrel Technology Palo Alto, CA, USA

Testing



Runtime Verification



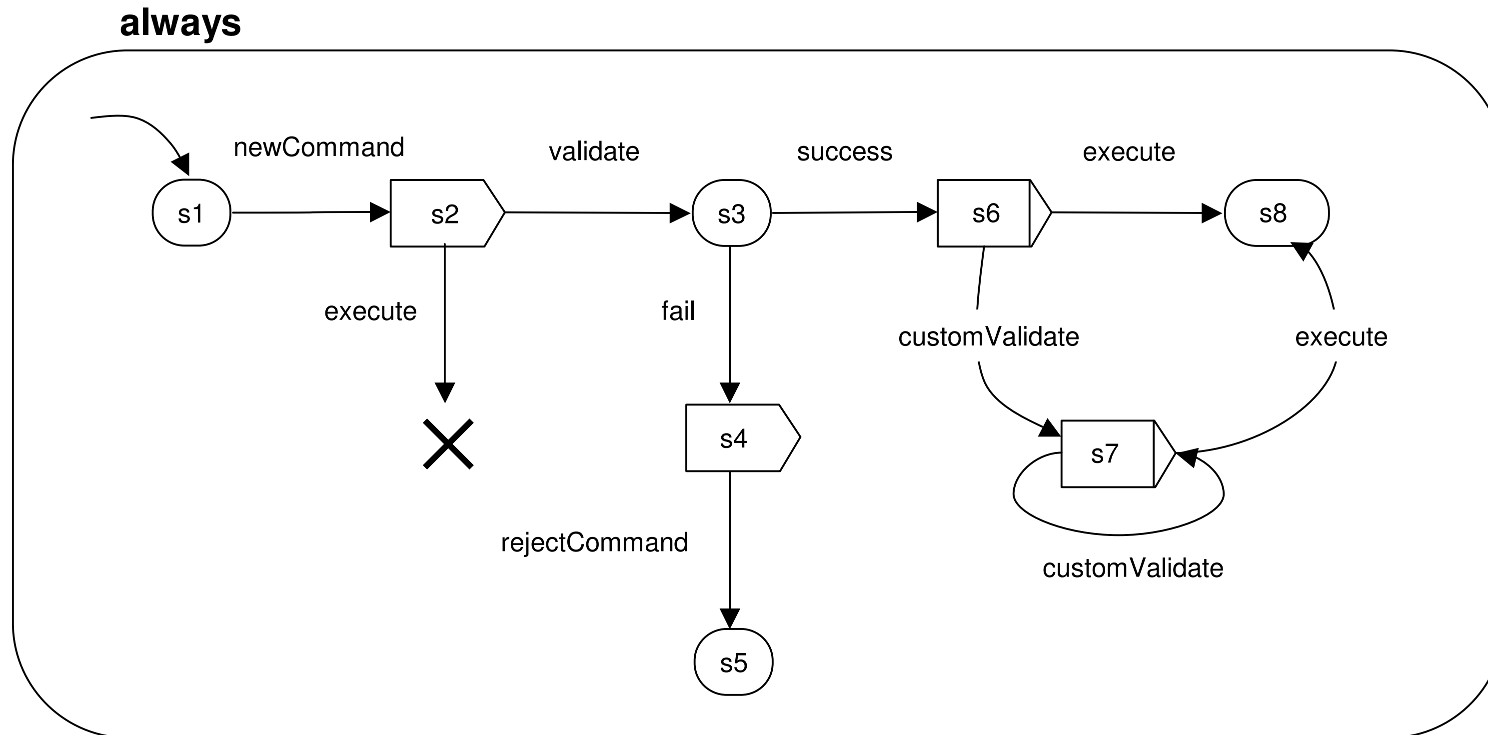
runtime monitoring
runtime checking
runtime validation
runtime analysis
dynamic analysis

Runtime Verification Tools

- Specification-based runtime verification
  - Eagle: A very powerful temporal logic
  - Rmor: A simple state machine-like language

- Algorithm-based runtime verification
  - Deadlocks - **this talk**
  - Low-level data races (Eraser)
  - High-level data races
  - Atomicity violations

Specification-Based Runtime Verification: RMOR



Two Kinds of Deadlocks

- Communication deadlocks

  Problems with **wait/notify**.

  Not analyzed in this work.

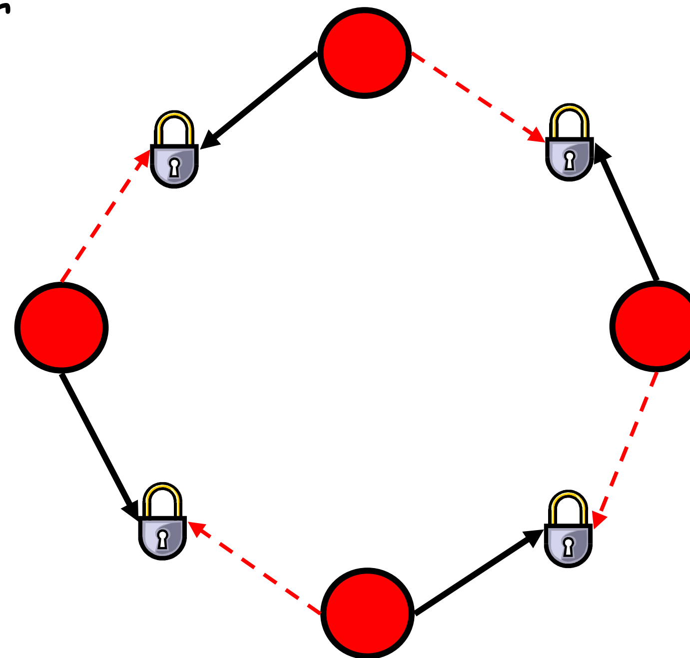
- Resource deadlocks

  Problems with **synchronized**.

  Analyzed in this work.

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.



A Java Example

A basic scenario

synchronized statements

Deadlock: T1 takes L1 and
T2 takes L2

T1:

```
synchronized(L1){
   ...
   synchronized(L2){
      ...
   }
   ...
}
```

T2:

```
synchronized(L2){
   ...
   synchronized(L1){
      ...
   }
   ...
}
```

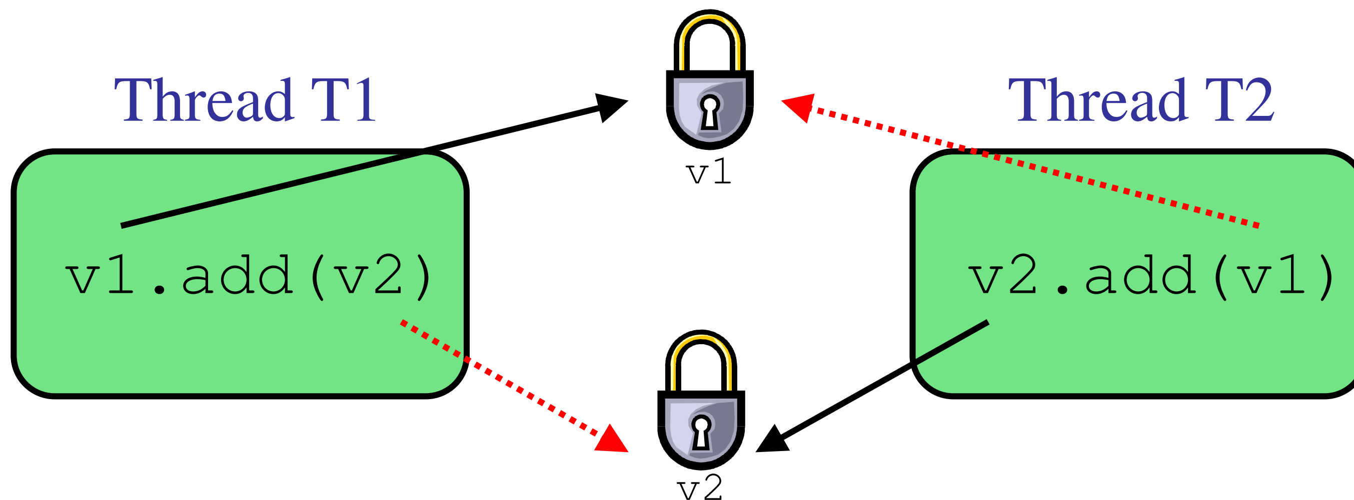
A Second Java Example

Dynamic locks

synchronized methods

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

```
v1 = new Value();
v2 = new Value();
```



A Third Java Example

A challenge for static analysis

dynamic locking

N = number of philosophers

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

arithmetic

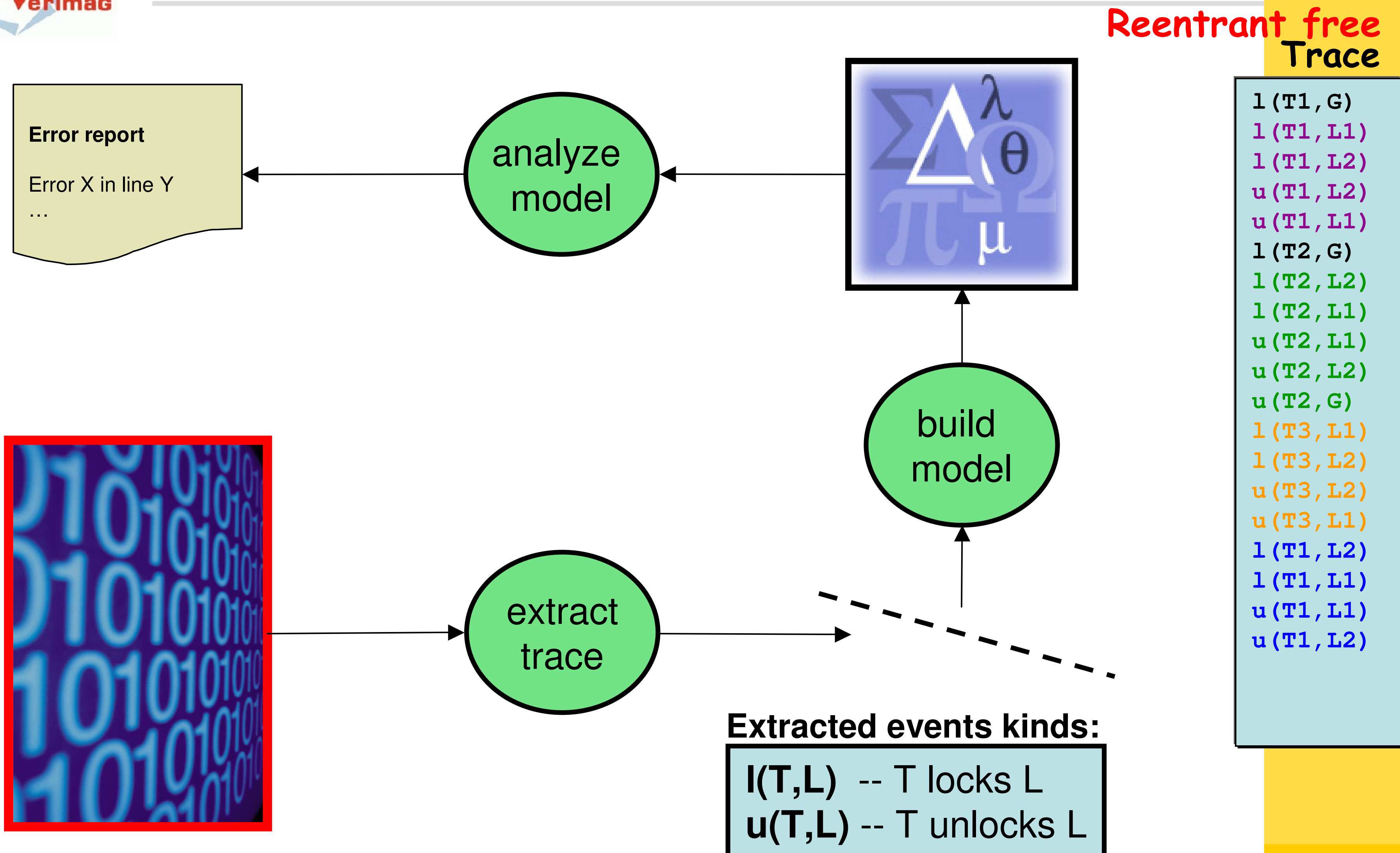
Philosopher:

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

  }
```

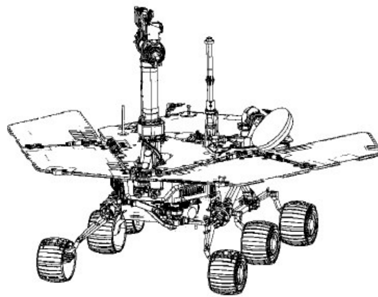
Analyzing the Dining Phil. with Static Analysis and Model checking

- **Jlint:** a static Java analyzer
  - No deadlocks found in deadlocking version
- **JPF:** a Java model checker
  - Deadlocking version:
    - N=15 (32 seconds)
    - N=20 (2.51 minutes)
    - N=21 (out of mem.)
  - Deadlock free version:
    - N=3 (3 minutes)
    - N=4 (26 minutes and out of mem.)
- **SPIN:** Similar experience as with JPF

Runtime Verification



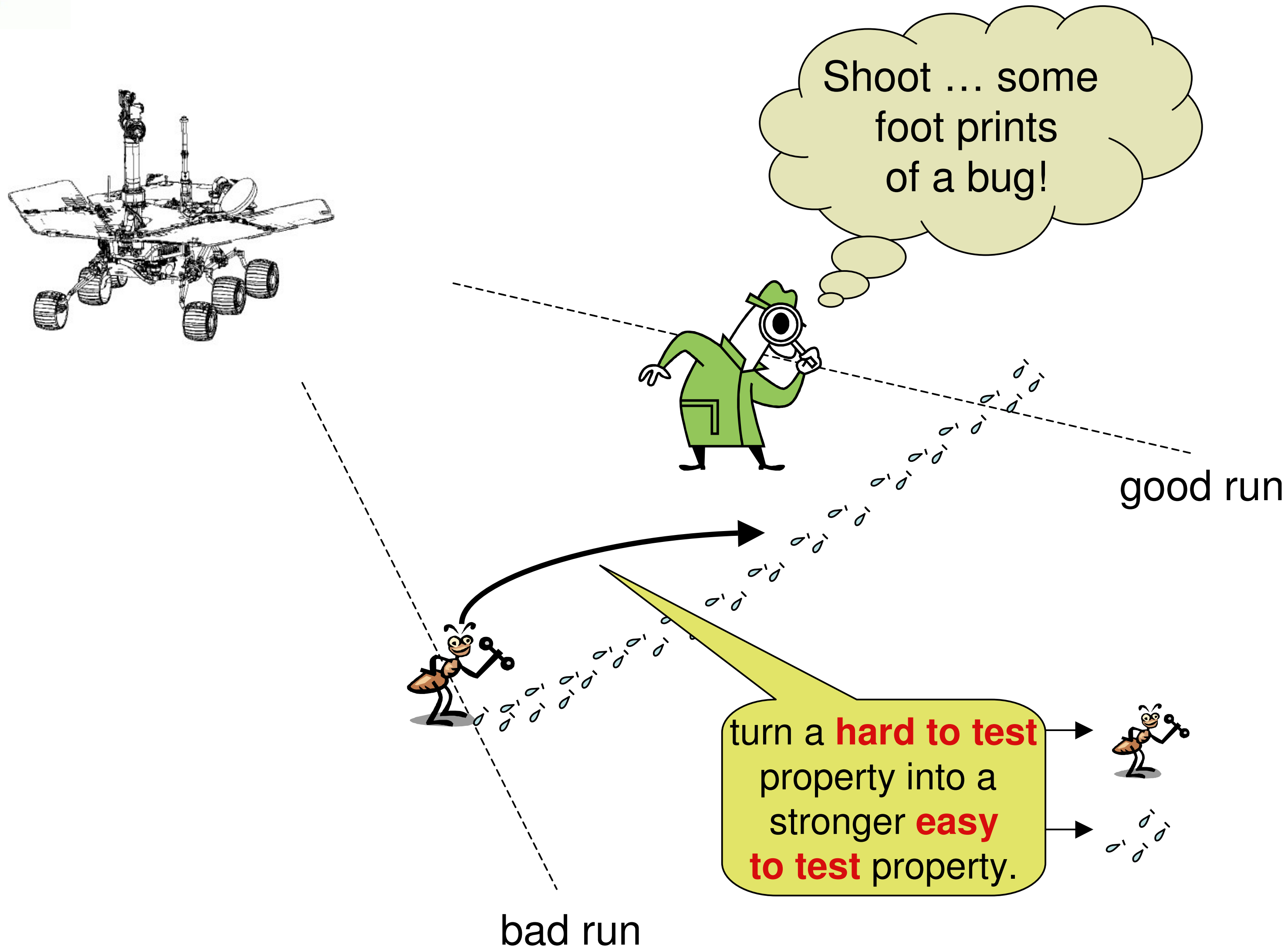
So Many Traces
So Little Time



good run

bad run

Algorithm-Based "Potential" Analysis



Pros and Cons

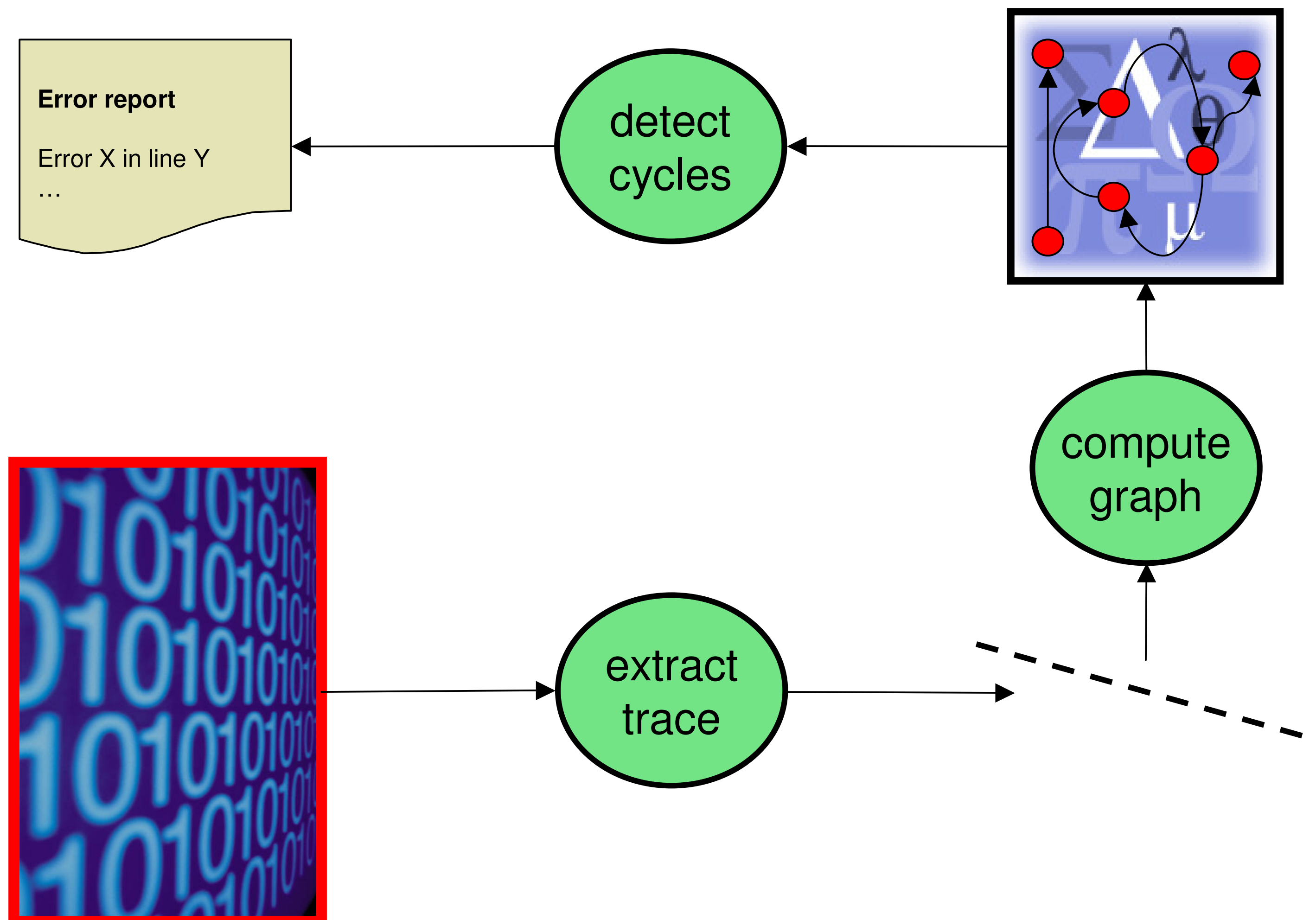
+ Scales well (one trace)
+ Often finds the bugs it is supposed to find

- Gives false positives
- Gives false negatives
- Only special bugs

Two Forms of Trace Analysis

- Build lock graph from trace and detect cycles in it.
- Build transition system from trace and model check it.

Detecting Cycles in Lock Graphs



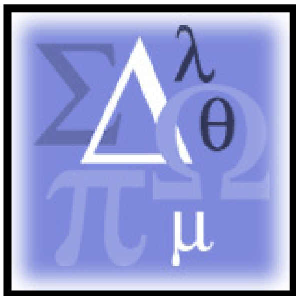
Basic Algorithm
Visiual Threads (Harrow 2000)

T1:

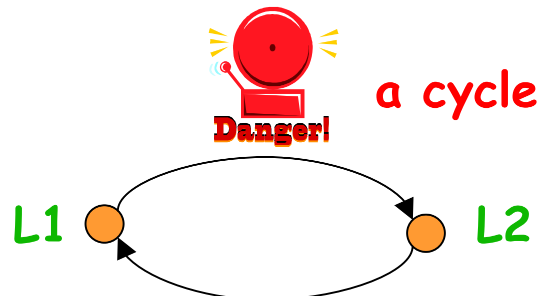
```
synchronized(L1){
   ...
   synchronized(L2){
      ...
   }
   ...
}
```

T2:

```
synchronized(L2){
   ...
   synchronized(L1){
      ...
   }
   ...
}
```



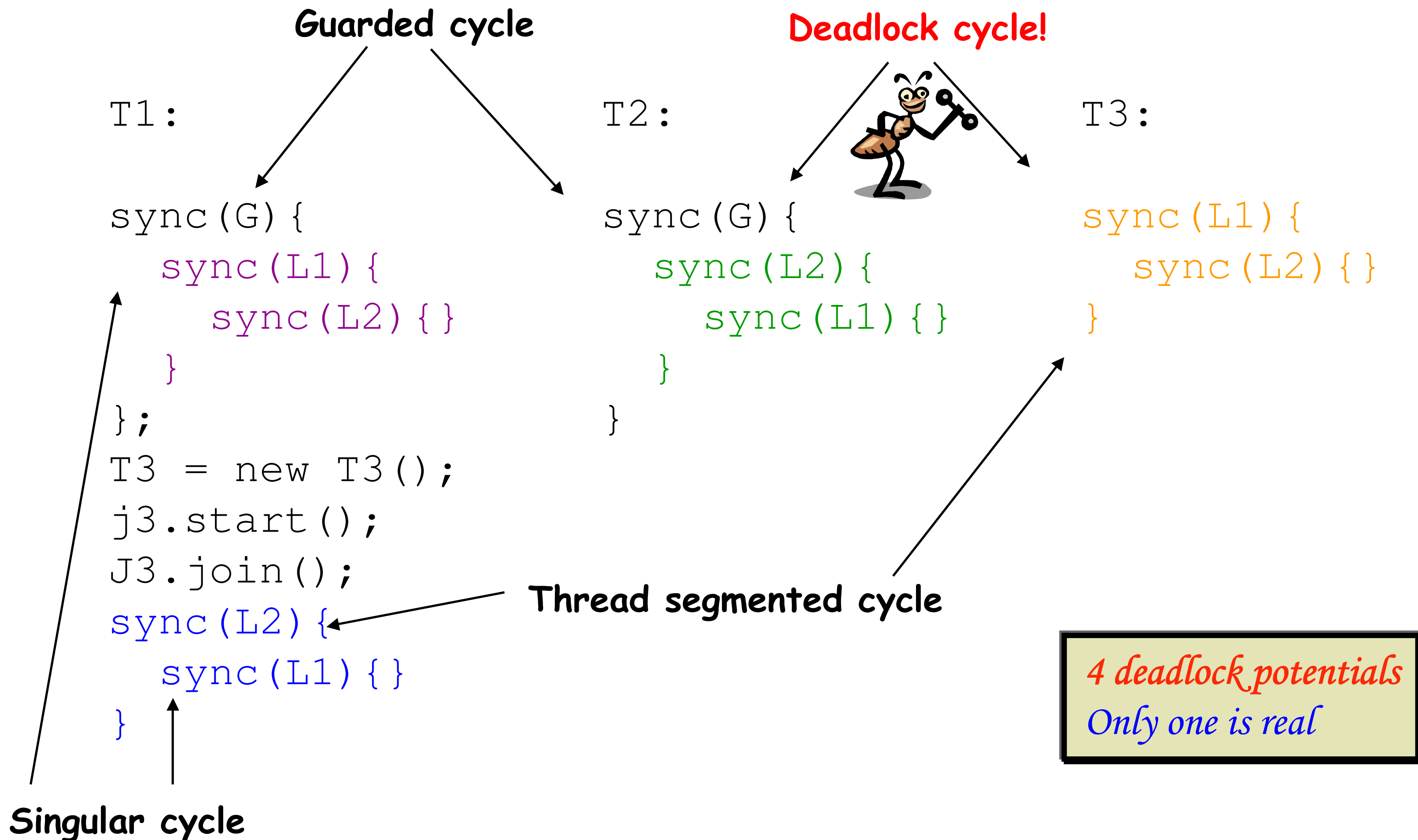
=



Basic Algorithm

```
Input: an execution trace δ

G_L : (Lock × Lock)-set        lock graph
C_L : [Thread → Lock-set]      lock context

for(i = 1..|δ|) do
  case δ[i] of
    l(t,o) →
      G_L := G_L ∪ { (o',o) | o' ∈ C_L(t) };
      C_L := C_L ⊕ [t ☞ C_L(t) ∪ {o} ];
    u(t,o) →
      C_L := C_L ⊕ [t ☞ C_L(t) \ {o} ];
  endcase
enddo

foreach c in cycles(G_L)
  print("deadlock potential: " + c);
```

Simple Algorithm Gives False Positives (False Warnings)



4 deadlock potentials
Only one is real

Extract 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:

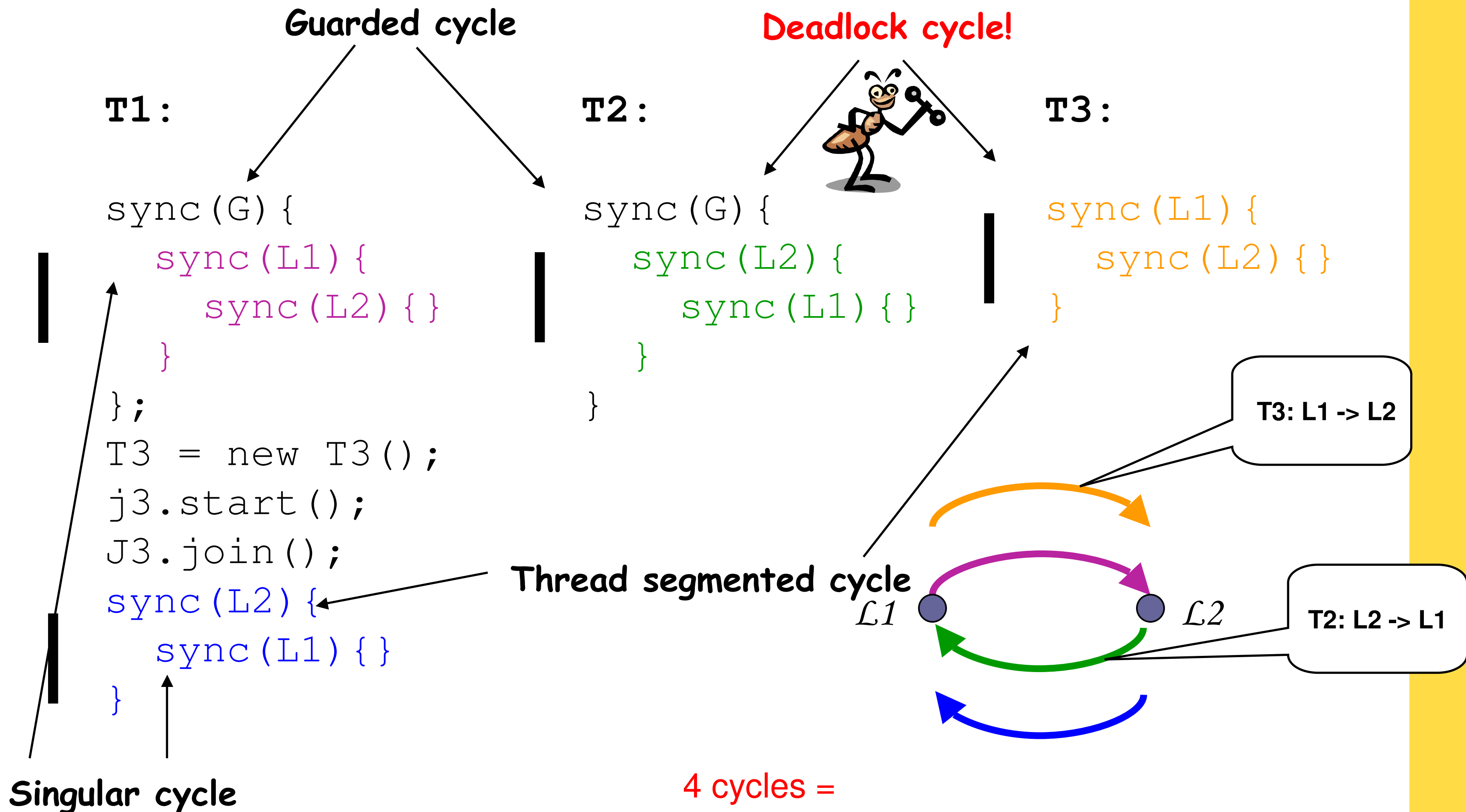
```
l(<thread>,<lock>)      - lock
u(<thread>,<lock>)      - unlock
s(<thread>,<thread>)    - start
j(<thread>,<thread>)    - join
```

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)
```

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

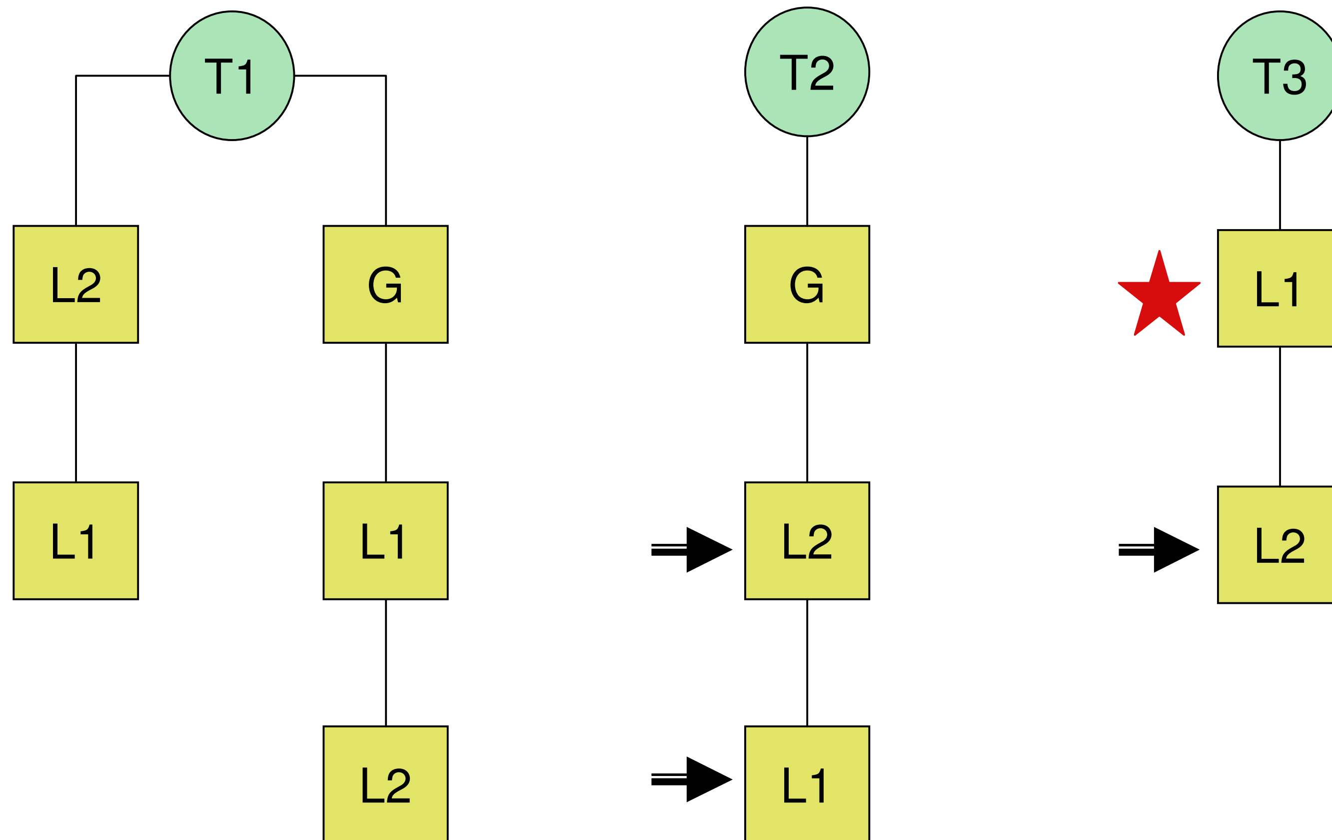
Basic Algorithm



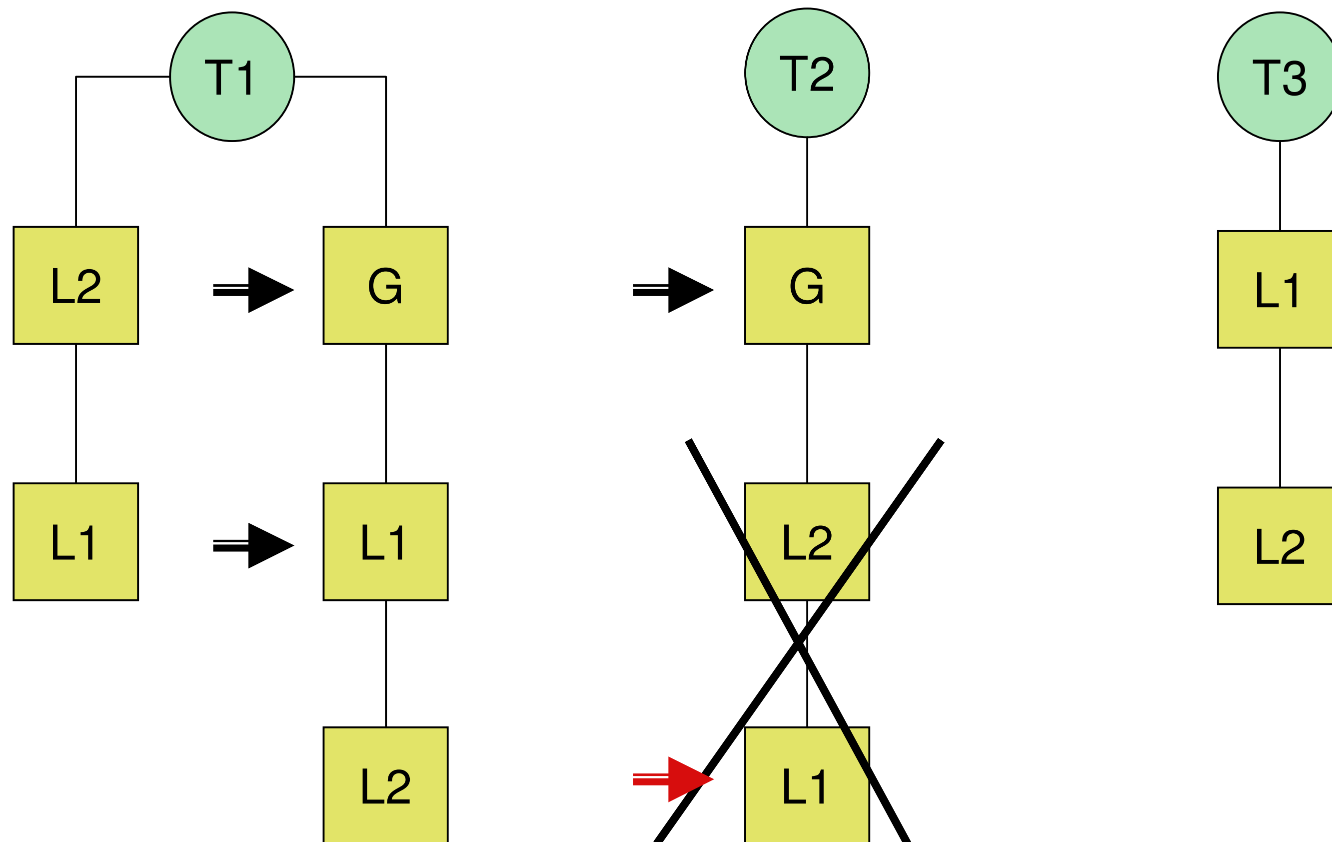
4 cycles =
4 deadlock potentials reported (Visual Threads).

1 real deadlock! (3 false positives)

The GoodLock Algorithm (Havelund 2000)



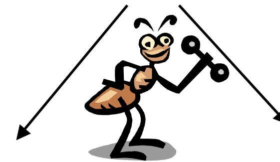
The GoodLock Algorithm (Havelund 2000)



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

Guarded Algorithm

Deadlock cycle!



T1:

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

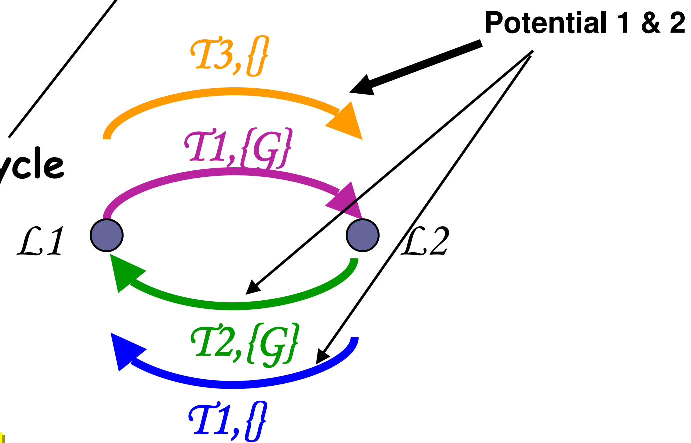
T2:

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

T3:

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

Thread segmented cycle



2 cycles =
2 deadlock potentials reported.

Valid Cycles:

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

Guarded Algorithm

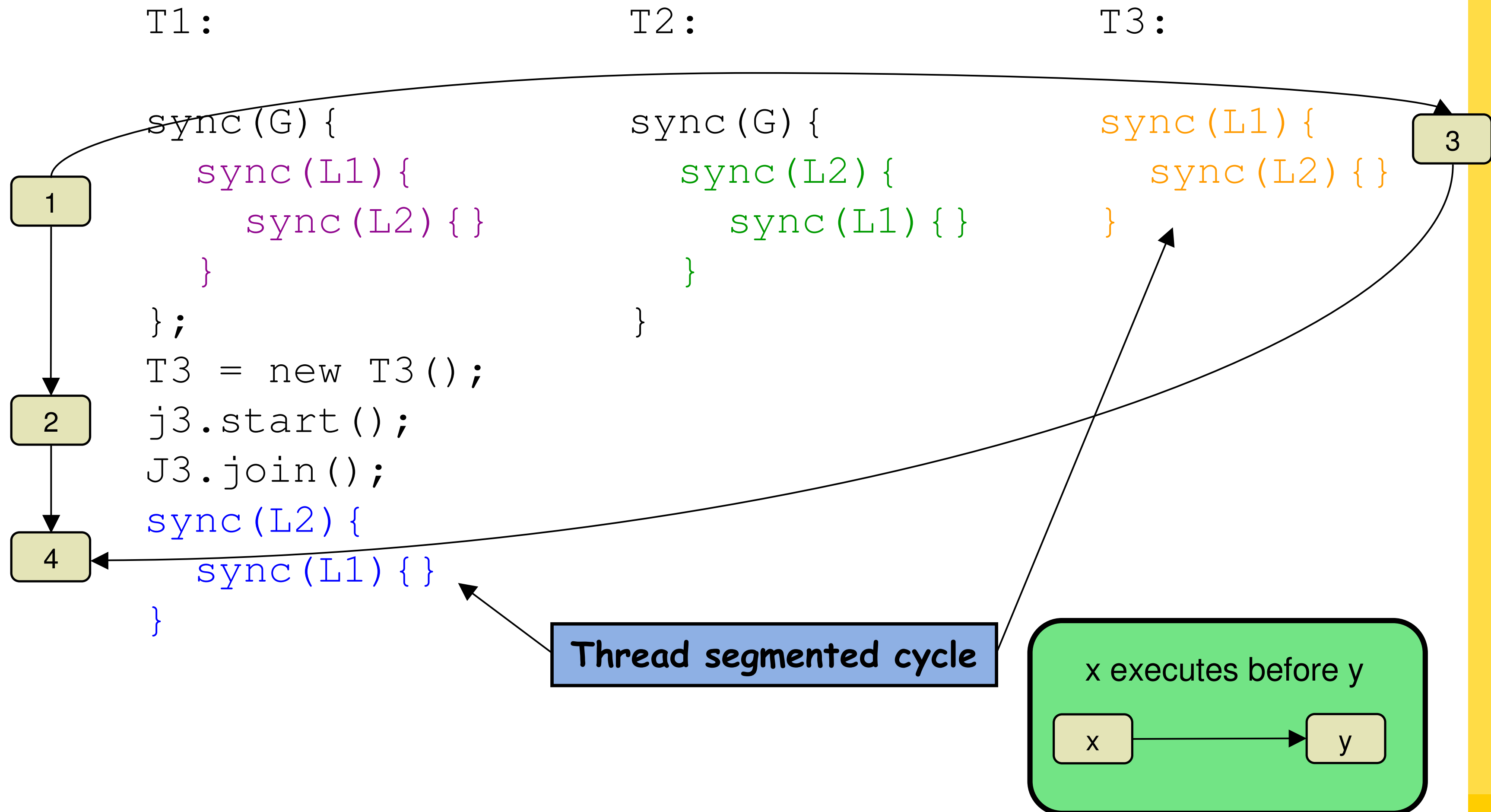
```
Input: an execution trace δ
type Label = Thread × Lock-set
G_L : (Lock × Label × Lock)-set            lock graph
C_L : [Thread → Lock-set]                  lock context

for(i = 1..|δ|) do
  case δ[i] of
    l(t,o) →
      G_L := G_L ∪ { (o',(t,g),o) |
                       o' ∈ C_L(t) ∧
                       g = { o'' | o'' ∈ C_L(t) }};
      C_L := C_L ⊕ [t ☞ C_L(t) ∪ {o} ];
    u(t,o) →
      C_L := C_L ⊕ [t ☞ C_L(t) \ {o} ];
  endcase
enddo

foreach c in valid-cycles(G_L)
  print("deadlock potential: " + c);
```

thread : Label → Thread
thread(t,g) = t

guards : Label → Lock-set
guards(t,g) = g

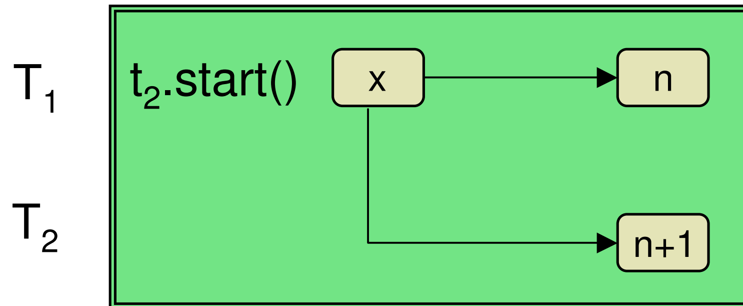
valid-cycle(c) =
$\forall e_1, e_2 \in c \bullet$
$thread(e_1) \neq thread(e_2) \wedge$
$guards(e_1) \cap guards(e_2) = \{\ \}$

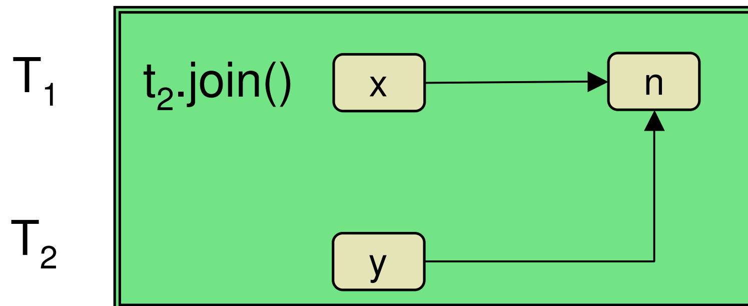
Dividing Code Into Segments



Effect of start() and join() on the Segmentation Graph 'S'

n : next free segment





Let S^* be the transitive closure of segmentation graph S.

The happens-before relation is defined as:

x ➲ y = $(x,y) \in S^*$

The in-parallel relation is defined as:

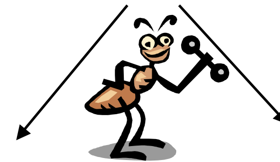
x || y = ¬(x ➲ y) ∧ ¬(y ➲ x)

Algorithm: extend lock graph with segment labels: which segments of a thread locks are taken.

Final Algorithm

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

Deadlock cycle!

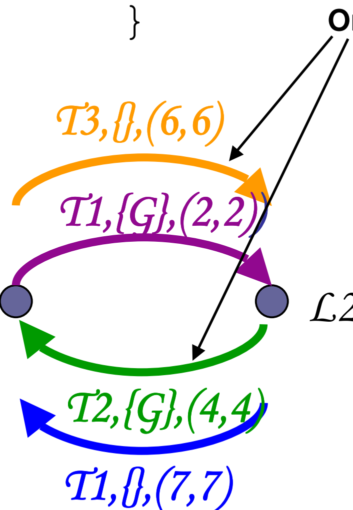


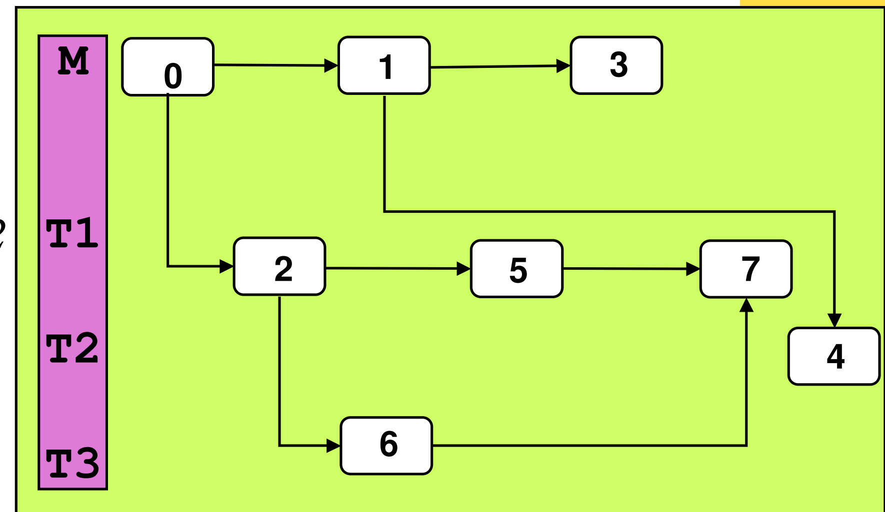
```
T1:

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){}
}
```

One potential left, the real deadlock!





Valid Cycles:

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

Final Algorithm

```
Input: an execution trace δ
type Label = nat × Thread × Lock-set × nat
G_L : (Lock × Label × Lock)-set         lock graph
G_S : (nat × nat)-set                   seg. Graph
C_L : [Thread → (Lock × nat)-set]       lock context
C_S : [Thread → nat]                    segm. Context
n : nat = 1;                            next available segm.

for(i = 1..|δ|) do
  case δ[i] of
    l(t,o) →
      G_L := G_L ∪ { (o',(s_1,t,g,s_2),o) |
                     (o', s_1) ∈ C_L(t) ∧
                     g = { o'' | (o'', *) ∈ C_L(t) } ∧
                     s_2 = C_S(t)};
      C_L := C_L ⊕ [t ☞ C_L(t) ∪ {(o,C_S(t))} ];
    u(t,o) →
      C_L := C_L ⊕ [t ☞ C_L(t) \ {(o,*)} ];
    s(t_1, t_2) →
      G_S := G_S ∪ { (C_S(t_1),n), (C_S(t_1),n+1)}
      C_S := C_S ⊕ [t_1 ☞ n, t_2 ☞ n+1];
      n := n + 2;
    j(t_1, t_2) →
      G_S := G_S ∪ { (C_S(t_1),n), (C_S(t_2),n)};
      C_S := C_S ⊕ [t_1 ☞ n];
      n := n + 1;
  endcase
enddo

foreach c in valid-cycles(G_L)
  print("deadlock potential: " + c);
```

thread : Label → Thread
$thread(s_1,t,g,s_2) = t$

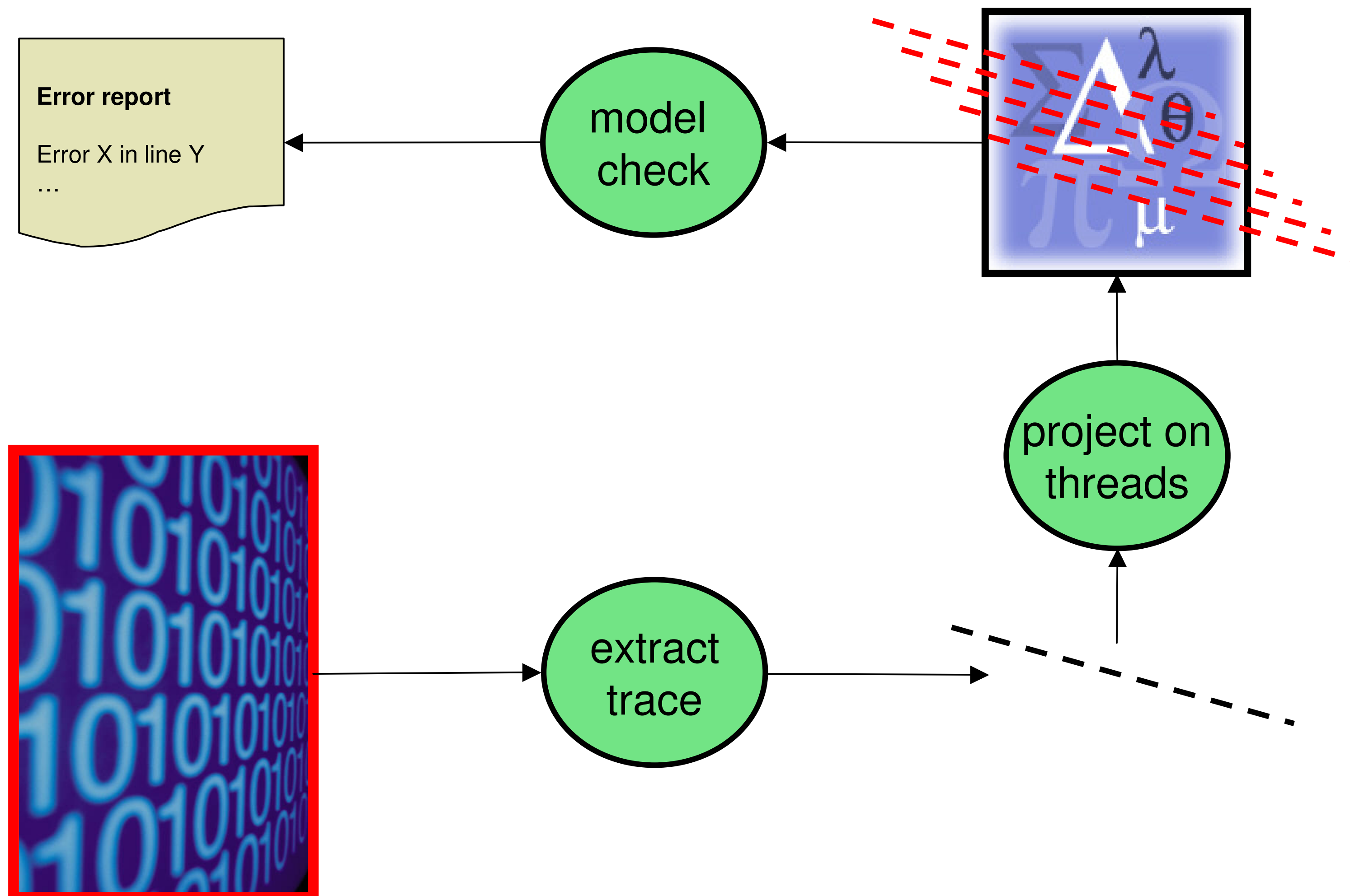
guards : Label → Lock-set
$guards(s_1,t,g,s_2) = g$

source : Label → nat
$source(s_1,t,g,s_2) = s_1$

target : Label → nat
$target(s_1,t,g,s_2) = s_2$

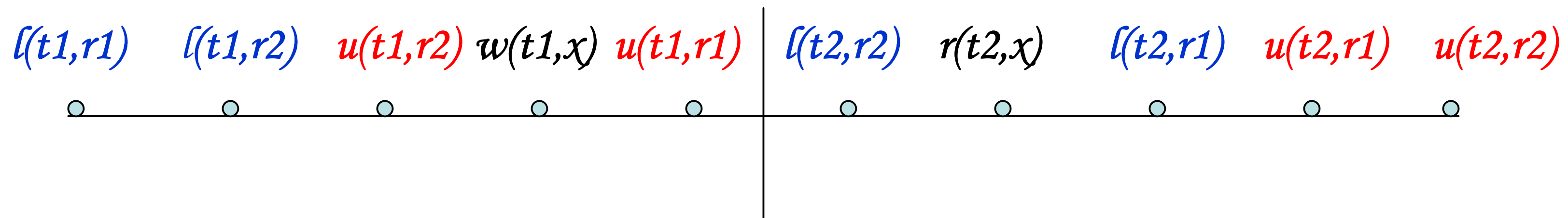
valid-cycle(c) =
$\forall e_1,e_2 \in c \bullet$
$thread(e_1) \neq thread(e_2) \wedge$
$guards(e_1) \cap guards(e_2) = \{ \} \wedge$
$target(e_1) \neg\supset target(e_2)$

Model Checking Traces



Model Checking Traces

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



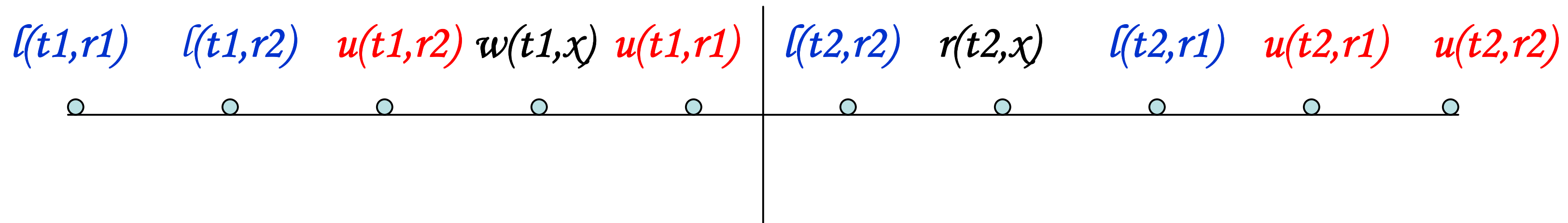
T1:

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

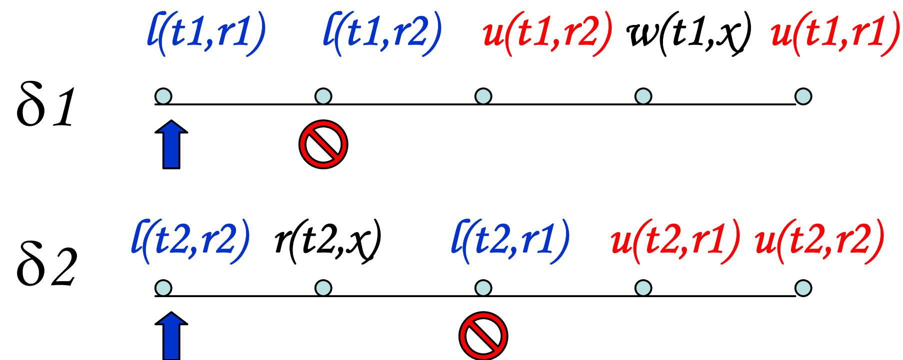
T2:

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

Model Checking Traces



project on threads



KESTREL TECHNOLOGY

Verimag

Deadlocking Transition Systems

An execution path ρ of $||S_i$ is a sequence:

$$\rho = s_0 \xrightarrow{e_0} s_1 \xrightarrow{e_1} s_2 \xrightarrow{e_2} \ldots \xrightarrow{e_{k-1}} s_k$$

such that s_0 is the initial state (1,1,…,1,{})
and there are no further moves from s_k.

deadlocking(ρ) = *last_state(ρ) $\neq$ ($k_1, k_2, \ldots, k_M$,{})*
That is: not all threads have reached their final state.

Deadlocking($||S_i$) = $\exists\ \rho \in ||S_i \bullet$ deadlocking(ρ)

DEADLOCKING(δ) = Deadlocking($||S_i$)

Experiment with Model Checking of Traces

better

- Deadlocking philosophers: for N=47 deadlock found in 5 minutes
- For *deadlock free* philosophers (recall that each eats 10 times):
  - for N=3 verified correct in 38 seconds
  - For N=4 out of memory (1.5 GB memory)

N=3: faster
N=4: out of mem.

Correctness Theorem

Let δ be an execution trace.
Let G_L be its lock graph.
Let $||S_i$ be its transition system.

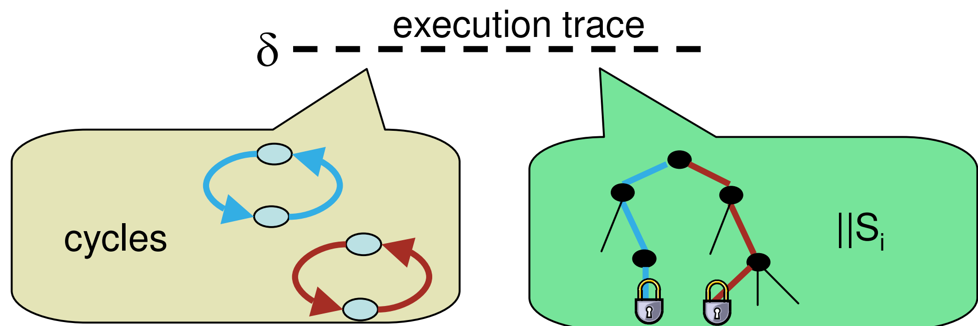
reflects : Cycle $\times$ Path $\rightarrow$ **Bool**
reflects(c,ρ) =
  $\forall$ (l_1,(_,t,_,_),l_2) $\in$ c •
   t **holds** l_1 **in** ρ $\wedge$
   t **next wants** l_2 **in** ρ

soundness wrt. execution trace δ:

$$\forall\ c \in \text{valid-cycles}(G_L) \bullet \exists\ \rho \in ||S_i \bullet \text{Deadlocking}(\rho) \wedge \text{reflects}(c,\rho)$$

completeness wrt. execution trace δ:

$$\forall\ \rho \in ||S_i \bullet \text{Deadlocking}(\rho) \Rightarrow \exists\ c \in \text{valid-cycles}(G_L) \bullet \text{reflects}(c,\rho)$$



KESTREL TECHNOLOGY

Verimag

Missed Warnings

Guarded cycle?

T1:

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

T2:

```
if(p)
  X := G
else
  X := H;
sync(X){
  sync(L2){
    sync(L1){}
  }
}
```

T3:

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

If p evaluates to true
Cycle becomes guarded
and error is missed.

KESTREL TECHNOLOGY

Verimag

The Problem of False Negatives

- Added covererage module reporting synchronization statements not executed per thread in a run.
- Will work better when combined with test case generation and more sophisticated coverage criteria.
- However, tool works surprisingly well on tried test cases (finds known bugs immediately).

The Testing Problem: Non-determinism

Prog(k,n):

```
synchronized(L1){
  if(random(1,n)==1)
    synchronized(L2){}
}

synchronized(L2){
  if(random(1,n)==1)
    synchronized(L3){}
}

...

synchronized(Lk){
  if(random(1,n)==1)
    synchronized(L1){}
}
```

Probability for deadlock to occur:

$P_D(k,n)$ = P(**Prog(k,n)** deadlocks in a run)

= 1/n * P(deadlock | random == 1)

= 1/n * (1 * (k-1)/k * (k-2)/k * … * 1/k)

= 1/n * k!/k^k

Example:

$P_D(k=4,n=3) = 0.03$

Runtime Verification on this Example is 10 times more Effective

Prog(k,n):

```
synchronized(L1){
  if(random(1,n)==1)
    synchronized(L2){}
}

synchronized(L2){
  if(random(1,n)==1)
    synchronized(L3){}
}

...

synchronized(Lk){
  if(random(1,n)==1)
    synchronized(L1){}
}
```

Probability for deadlock to occur:

$P_C(k,n)$ = P(**Prog(k,n)** generates cycle in a run)

= 1/n * P(cycle | random == 1)

= 1/n * 1

= 1/n

Example:

$P_C(k=4,n=3) = 0.33$

$= 10 * P_D(k=4,n=3)$ < 1

KESTREL TECHNOLOGY

Verimag

The Dining Philosophers

model checking

Deadlocking version:
- N=15 (32 seconds),
- N=20 (2.51 minutes),
- N=21 (out of mem.)

Deadlock free version:
- N=3 (3 minutes),
- N=4 (out of mem.)

trace model checking

Deadlocking version:
- N=47 (5 minutes),

Deadlock free version:
- N=3 (38 seconds),
- N=4 (out of mem.)

runtime analysis

Deadlocking version:
- N=100 (8 seconds),
- N=300 (22 seconds)
- …

Deadlock free version:
- N=4 (7 seconds)
- N=100 (30 seconds)
- N=300 (2 minutes)
- …

Real Case Studies

NASA's K9 Rover controller:

35,000 lines of C++

Found 1 deadlock in first run.

NASA K9 Java Rover controller:

8,000 lines of Java (small Java version of C++ version)

Found 2 planted deadlocks in first run. Did well in "competition".

NASA IDEA plan-runner:

C++

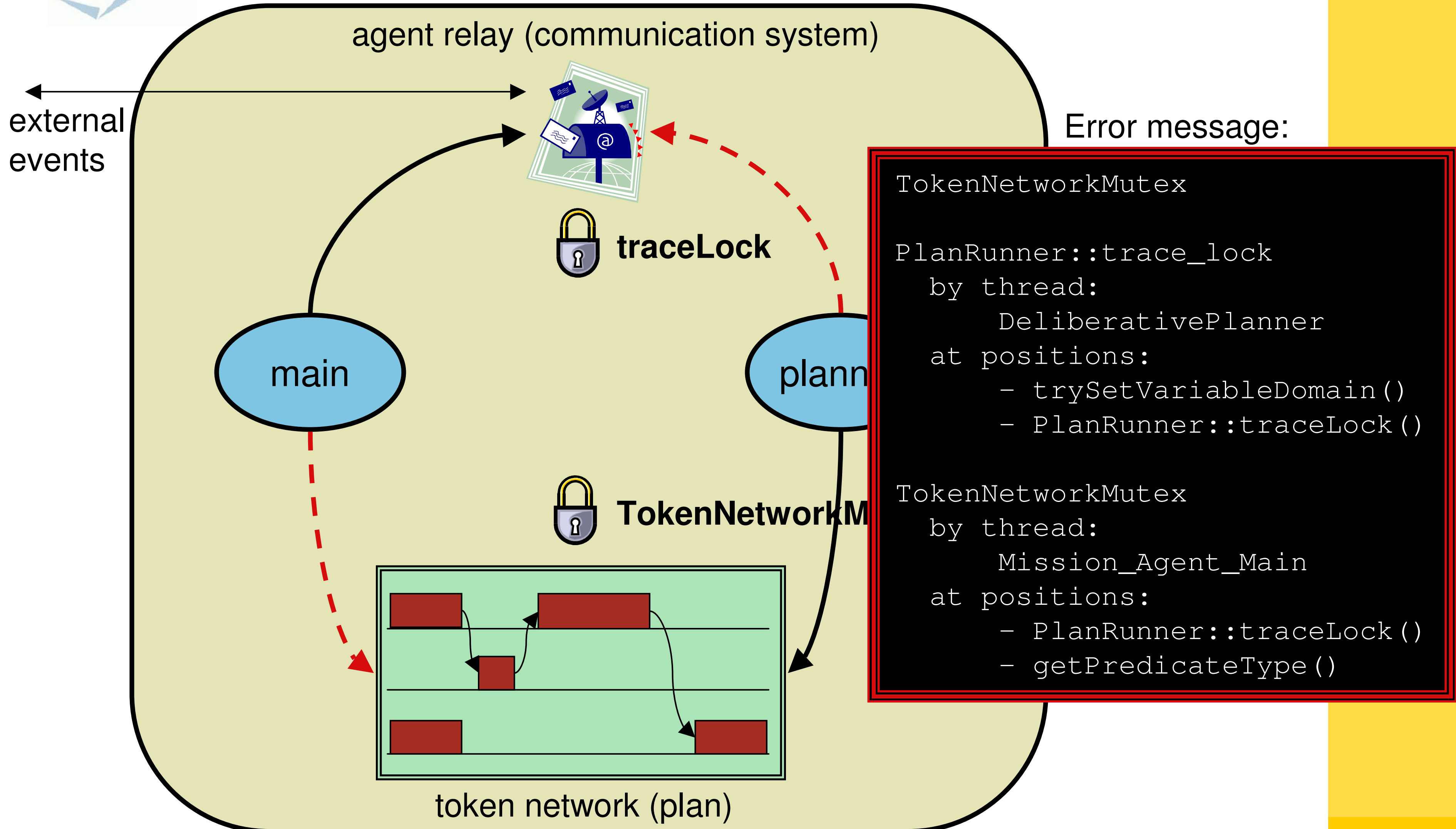
Found 2 deadlocks.

NASA attitude controller:

No deadlocks found (but Java version of Eraser found a data race).

Deadlock Analysis of the NASA 'IDEA' Planner

With Chuck Fry,
QSS, NASA Ames



Observations

- **Dynamic Analysis**
  - It's very easy to instrument code for this analysis. Static analysis is much more complicated to implement (consider fx. C++, it took programmer only 20 min. to instrument rover code for dynamic analysis).
  - Detects bugs very efficiently, usually in first run.
  - However, combine with test case generation for better results.
  - Study is needed to explore relationship with coverage criteria.

- **Static analysis:**
  - Some dynamic cases seem hard to catch statically (Dining Phil. modulo).
  - However, static analysis might be used to detect "as much as possible". Combine with dynamic analysis. Systematic comparison needed.
  - Type systems.

- **Preventive meassures:**
  - Multiple locking operations:

    ```
    synchronized(A,B){…}
    ```

  - A planning oriented approach, where programmer states intentions.

End