Software Model Checking Tools and Trends at NASA

Klaus Havelund Recom / QSS / NASA Ames
Charles Pecheur RIACS / NASA Ames
Reid Simmons Carnegie Mellon University
Willem Visser RIACS / NASA Ames

Contact Info

Klaus Havelund <havelund@ptolemy.arc.nasa.gov>
NASA Ames Research Center, M/S 269-2
Moffett Field, CA 94035, U.S.A.

Charles Pecheur <pecheur@ptolemy.arc.nasa.gov>
NASA Ames Research Center, M/S 269-2
Moffett Field, CA 94035, U.S.A.

Reid Simmons <reids@cs.cmu.edu>
Robotics Institute, Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213, U.S.A.

Willem Visser <wvisser@ptolemy.arc.nasa.gov>
NASA Ames Research Center, M/S 269-2
Moffett Field, CA 94035, U.S.A.

Outline

- Model Checking for Autonomy Software
  - **SMV (And Compiling to It)** *Charles*
  - **Verification of Autonomy Software** *Reid*

- Model Checking for Programming Languages
  - **Model Checking Programs** *Willem*
  - **Runtime Analysis of Programs** *Klaus*

SMV
And Compiling to It

Charles Pecheur

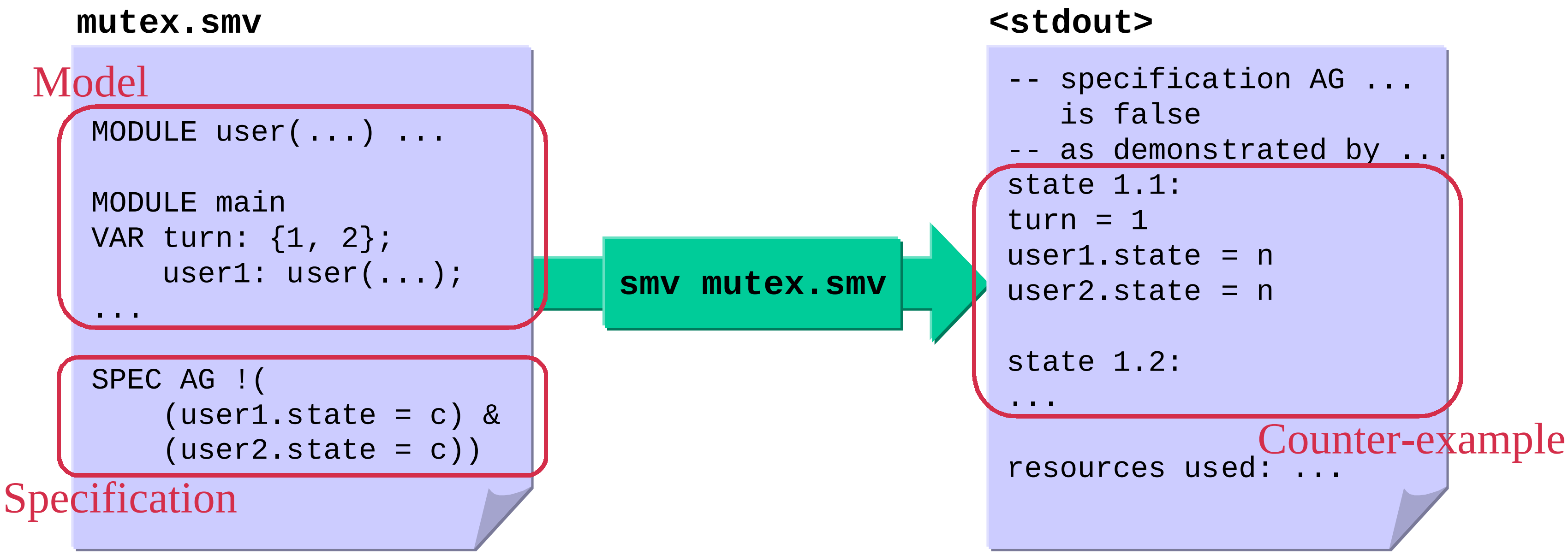
RIACS / NASA Ames

partially based on material from *Marius Minea*

Overview

- **SMV** = **S**ymbolic **M**odel **V**erifier.
- Developed by Ken McMillan at Carnegie Mellon University.
- Modeling language for transition systems based on parallel assignments.
- Specifications in temporal logic CTL.
- BDD-based symbolic model checking: can handle very large state spaces.

What SMV Does



mutex.smv

Model

```
MODULE user(...) ...

MODULE main
VAR turn: {1, 2};
    user1: user(...);
...
```

```
SPEC AG !(
    (user1.state = c) &
    (user2.state = c))
```

Specification

smv mutex.smv

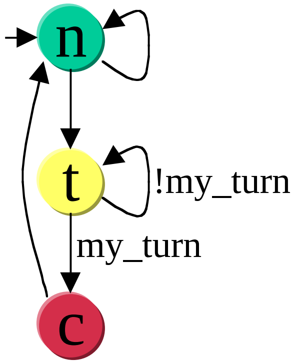
<stdout>

```
-- specification AG ...
   is false
-- as demonstrated by ...
state 1.1:
turn = 1
user1.state = n
user2.state = n

state 1.2:
...

resources used: ...
```

Counter-example

SMV Program Example (1/2)

```
MODULE user(turn,id,other)
VAR state: {n, t, c};
DEFINE my_turn :=
   (other=n) | ((other=t) & (turn=id));
ASSIGN
init(state) := n;
next(state) := case
   (state = n) : {n, t};
   (state = t) & my_turn: c;
   (state = c) : n;
   1 : state;
esac;

SPEC AG((state = t) -> AF (state = c))
```



SMV Program Example (2/2)

```
MODULE main
VAR turn: {1, 2};
    user1: user(turn, 1, user2.state);
    user2: user(turn, 2, user1.state);
ASSIGN
init(turn) := 1;
next(turn) := case
   (user1.state=n) & (user2.state=t): 2;
   (user2.state=n) & (user1.state=t): 1;
   1: turn;
esac;

SPEC AG !((user1.state=c) & (user2.state=c))
SPEC AG !(user1.state=c)
```

Diagnostic Trace Example

```
-- specification AG (state = t -> AF state = c) (in
   module user1) is true
-- specification AG (state = t -> AF state = c) (in
   module user2) is true
-- specification AG (!(user1.state = c & user2.state =
   c)... is true
-- specification AG (!user1.state = c) is false
-- as demonstrated by the following execution sequence
state 1.1:
turn = 1
user1.state = n
user2.state = n

state 1.2:
user1.state = t

state 1.3:
user1.state = c
```

The Essence of SMV

- The SMV program defines:
  - a finite transition model M (Kripke structure),
  - a set of possible initial states I (may be several),
  - specifications P_1 .. P_m (CTL formulas).
- For each specification P, SMV checks that

$$\boxed{\forall\, s_o \in I \;.\; M, s_o \models P}$$

Note: `SPEC !P` is not the negation of `SPEC P`:
both can be false (in some initial states),
both can be true (vacuously when $I=\emptyset$).

Variables and Transitions (Assignment Style)

```
VAR state: {n, t, c};
ASSIGN
init(state) := n;
next(state) := case
   (state = n) : {n, t}; ...
esac;
```

- Finite data types (incl. numbers and arrays).
- Usual operations `x&y`, `x+y`, etc., `case` statement.
- All assignments are evaluated in parallel.
- No control flow (must be simulated with vars).
- SMV detects circular and duplicate assignments.

Defined Symbols

```
DEFINE my_turn :=
   other=n | (other=t & turn=id);
ASSIGN
next(state) := case ...
   (state = t) & my_turn:  c; ...
esac;
```

- Defines an abbreviation (macro definition).
- No new state variable is created

  => no added complexity for model checking.
- No type declaration is needed.

Modules

```
MODULE user(turn,id,other)
VAR ...
ASSIGN ...
MODULE main
VAR user1: user(turn,1,user2.state);
    ...
```

- Parameters passed by reference.
- Top-level module `main`.
- Composition is synchronous by default: all modules move at each step.

Records

Modules without parameters and assignments.

```
MODULE point
VAR   x : {0,1,2,3,4,5};
      y : {0,1,2,3,4,5};

MODULE main
VAR   p : point;
ASSIGN
   init(p.x) := 0; init(p.y) := 0;
   ...
```

Processes

```
VAR node1: process node(1);
    node2: process node(2);
```

- Composition of processes is asynchronous: one process moves at each step.
- Boolean variable `running` in each process
  - `running=1` when that process is selected to run.
  - Used for fairness constraints (see later).

Specifications

`SPEC AG ((state = t) -> AF (state = c))`

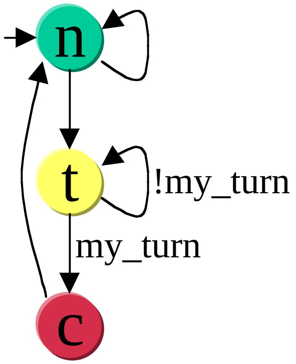
"Whenever state `t` is reached, state `c` will always eventually be reached."

- Standard CTL syntax:

  `AX p`, `AF p`, `AG p`, `A[p U q]`, `EX p`, ...
- Can be added in any module.
- Each specification is verified separately.

Fairness

```
MODULE user(turn,id,other)
VAR ...
ASSIGN ...
SPEC AG AF (state = c)
FAIRNESS (state = t)
```



- Check specifications, assuming fairness conditions hold repeatedly (infinitely often).
- Useful for liveness properties.
- Fair scheduling: `FAIRNESS running`

Variables and Transitions (Constraint Style)

```
VAR pos: {0,1,2,3,4,5};
INIT pos < 2
TRANS (next(pos)-pos) in {+2,-1}
INVAR !(pos=3)
```

- Any propositional formula is allowed
  => flexible for translation from other languages.
- `INVAR p` is equivalent to

  ```
  INIT p
  TRANS next(p)
  ```

  but implemented more efficiently.
- Risk of inconsistent models (`TRANS p & !p`).

Well-Formed Programs?

- In assignment style, by construction:
  - always at least one initial state,
  - all states have at least one next state,
  - non-determinism is apparent (unassigned variables,set assignments, interleaving).
- In constraint style:
  - `INIT` and `TRANS` constraints can be inconsistent,
  - the level of non-determinism is emergent from the conjunction of all constraints.

Inconsistency

- Inconsistent `INIT` constraints
  => inconsistent model: no initial state.
  - `SPEC 0` (or any `SPEC P`) is vacuously true.
- Inconsistent `TRANS` constraints
  => deadlock state: state with no next state
  => transition relation is not complete.
  - SMV does not work correctly in this case.
  - SMV will detect and report it.

Variable Ordering

- BDDs require a fixed variable ordering .
  - Critical for performance (BDD size).
  - Best one is hard to find (NP-complete).
- SMV does not optimize by default but
  - can read, write ordering in a file,
  - can search for better ordering on demand.

Re-ordering Variables

Using command line options:

```
smv -o demo.var
```

Outputs variable ordering to `demo.var`.
`demo.var` is text, can be re-ordered manually.

```
smv -i demo.var
```

Inputs variable ordering from `demo.var`.

```
smv -reorder
```

Does variable re-ordering when BDD size exceeds a certain (configurable) limit.

```
smv -reorder -oo demo.var
```

Outputs to `demo.var` after each change.

Re-ordering Variables Method for Tough Cases

Problem (Livingstone ISPP model):

```
smv ispp.smv
```

-> Memory overflow.

```
smv -reorder ispp.smv
```

-> keeps re-ordering again and again...

Solution:

```
smv -reorder -oo ispp.var ispp.smv
```

Wait until "enough" re-ordering (statistics).

```
^C
smv -i ispp.var ispp.smv
```

-> Goes to completion (10^{50} states).

Availability

- Freely downloadable.
- Source or binaries for Unix (SunOS4, SunOS5, Linux x86, Ultrix).
- Windows NT port (Dong Wang).
- see http://www.cs.cmu.edu/~modelcheck/smv.html

NuSMV

- From ITC-IRST (Trento, Italy) and CMU.
- New version of SMV, completely rewritten:
  - Same language as SMV.
  - Modular, documented APIs, easily customized.
  - Specifications in CTL or LTL.
  - Graphical User Interface.
  - Usually faster but uses more memory.
- See http://sra.itc.it/tools/nusmv/index.html

Other Related Tools

- **Cadence SMV** (Cadence Berkeley Labs)
  - From Ken McMillan, original author of SMV.
  - Supports refinement, compositional verification.
  - New language but accepts CMU SMV.
  - see http://www-cad.eecs.berkeley.edu/~kenmcmil/smv/
- **BMC** = Bounded Model Checker (CMU)
  - Uses SAT procedures instead of BDDs: bounded depth but usually faster, less memory.
  - Simple SMV-like language (no modules).
  - Early beta version.
  - see http://www.cs.cmu.edu/~modelcheck/bmc.html

References

Ken McMillan. Symbolic Model Checking. Kluwer Academic Publishers, 1993.

Based on Ken McMillan's PhD thesis on SMV.

J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic model checking: 10^20 states and beyond. Information and Computation, vol. 98, no. 2, 1992.

The reference survey paper on the principles of SMV.

Ken L. McMillan. The SMV System (draft). February 1992. http://www.cs.cmu.edu/~modelcheck/smv/smvmanual.r2.2.ps

The (old) user manual provided with the SMV program.

Verification of Autonomy Software

Reid Simmons
Carnegie Mellon University

Autonomous Systems

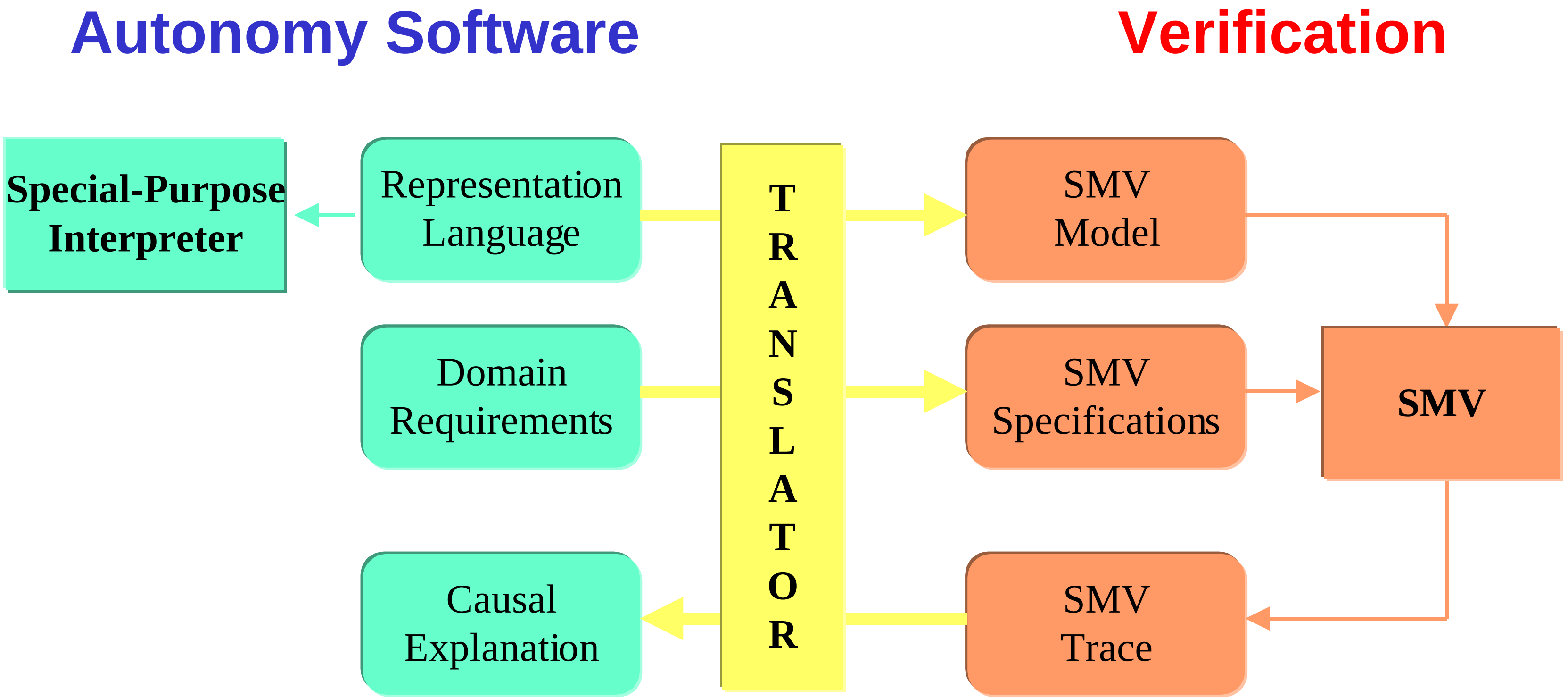
- Achieve complex tasks in *uncertain, unstructured* environments
  - Combine deliberative and reactive behaviors
  - Highly conditional; Non-local flow of control
  - Feedback loops at multiple levels of abstraction

- Architectures for Autonomy
  - Specialized representations and algorithms
  - ***Model-based programming***

Aspects of Verification

- Verifying the Interpreter
  - Special-purpose languages
- Verifying for Internal Correctness
  - Check for deadlock, safety, resource conflict, …
- Verifying for External Correctness
  - How the system interacts with the environment

Architecture for Verification of Autonomy Software



Autonomy Software

Verification

Special-Purpose Interpreter

Representation Language

Domain Requirements

Causal Explanation

TRANSLATOR

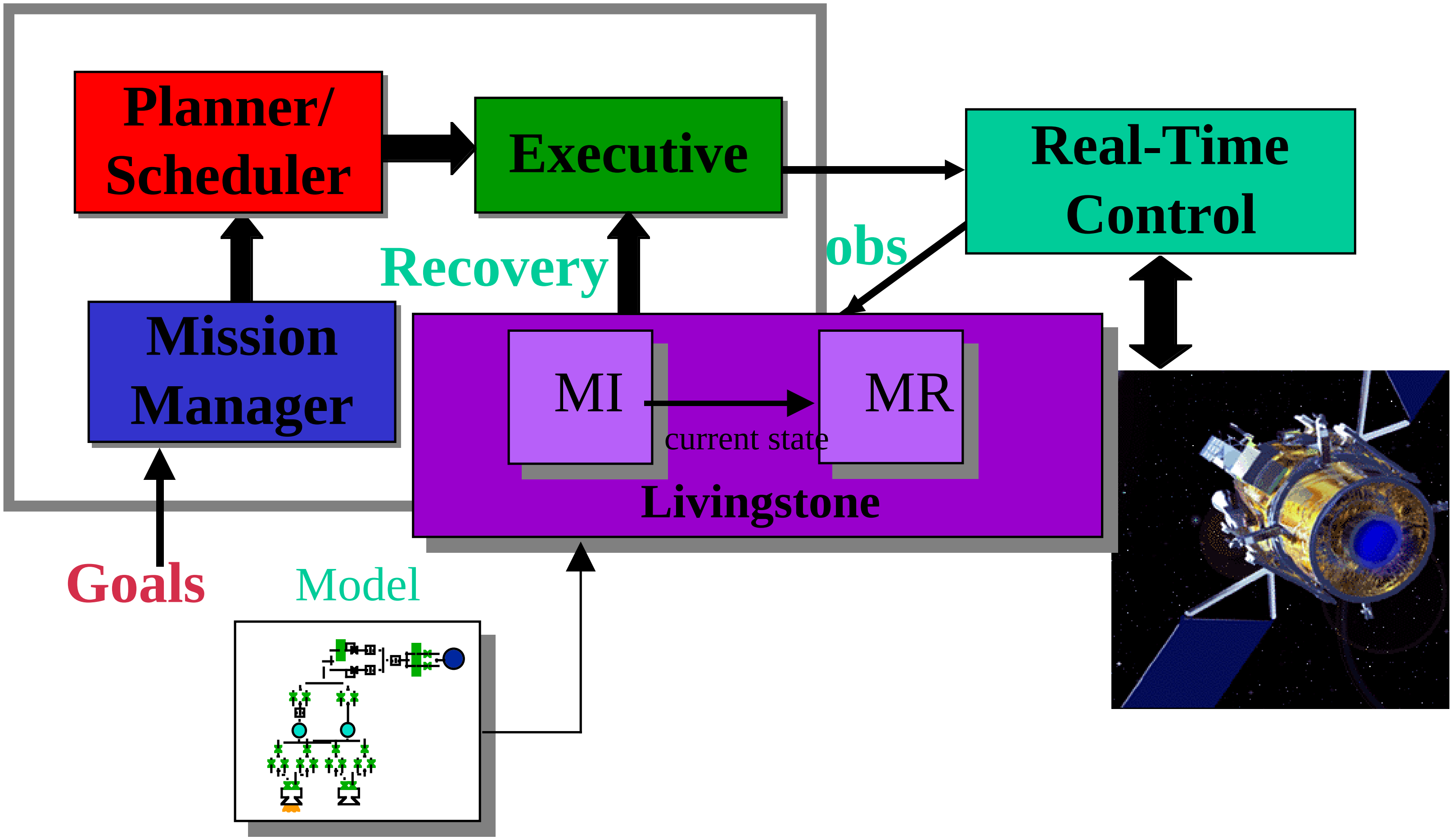
SMV Model

SMV Specifications

SMV Trace

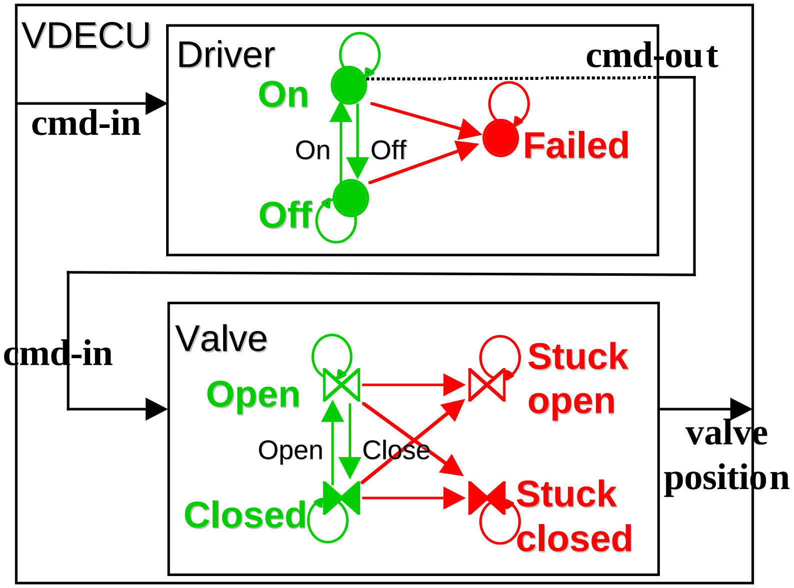
SMV

Remote Agent

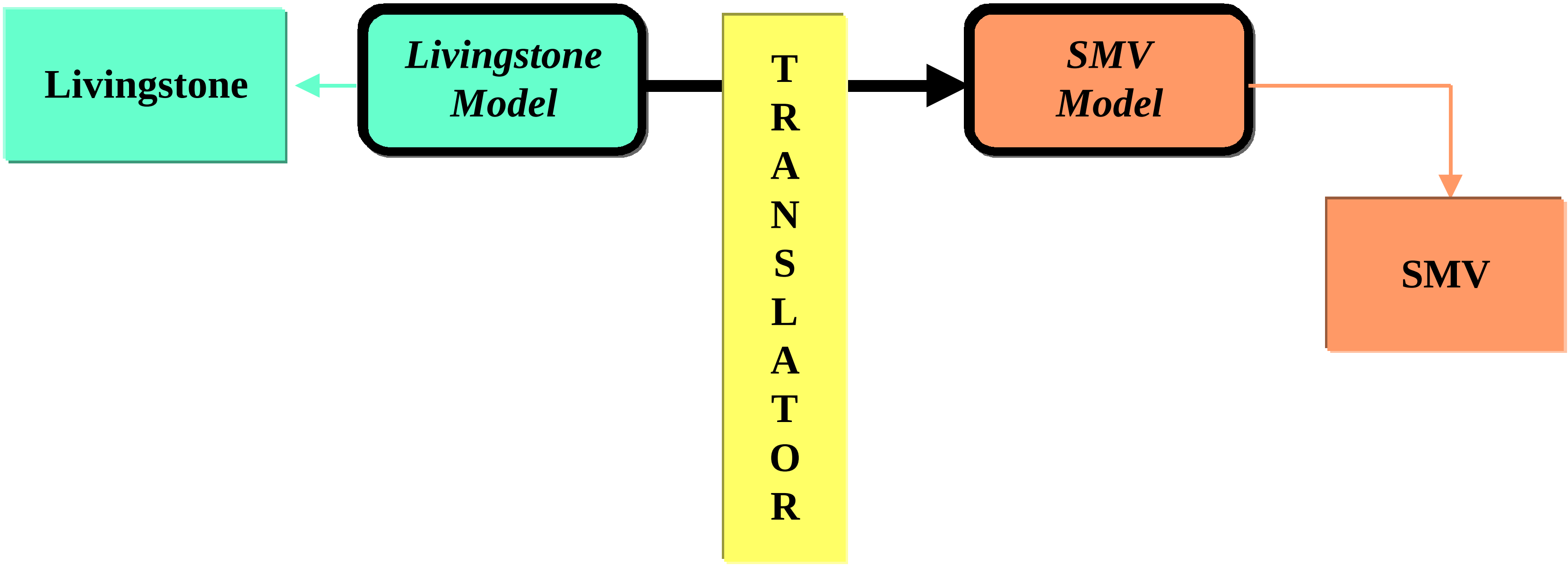


Livingstone

- Model-based system for fault diagnosis
  - Detects conflicts between observed and predicted state variables
  - Diagnoses inconsistencies (nominal/fault modes)
  - Finds recovery actions
  - Qualitative
  - Hierarchical
  - Lisp-based



Translation



Livingstone

Livingstone Model

TRANSLATOR

SMV Model

SMV

Formalizing the Model

MPL	**SMV**
component	module
module	module
variables	scalar variables
structures	module variables
mode transitions	TRANS
model constraints	INVAR
initial state	INIT

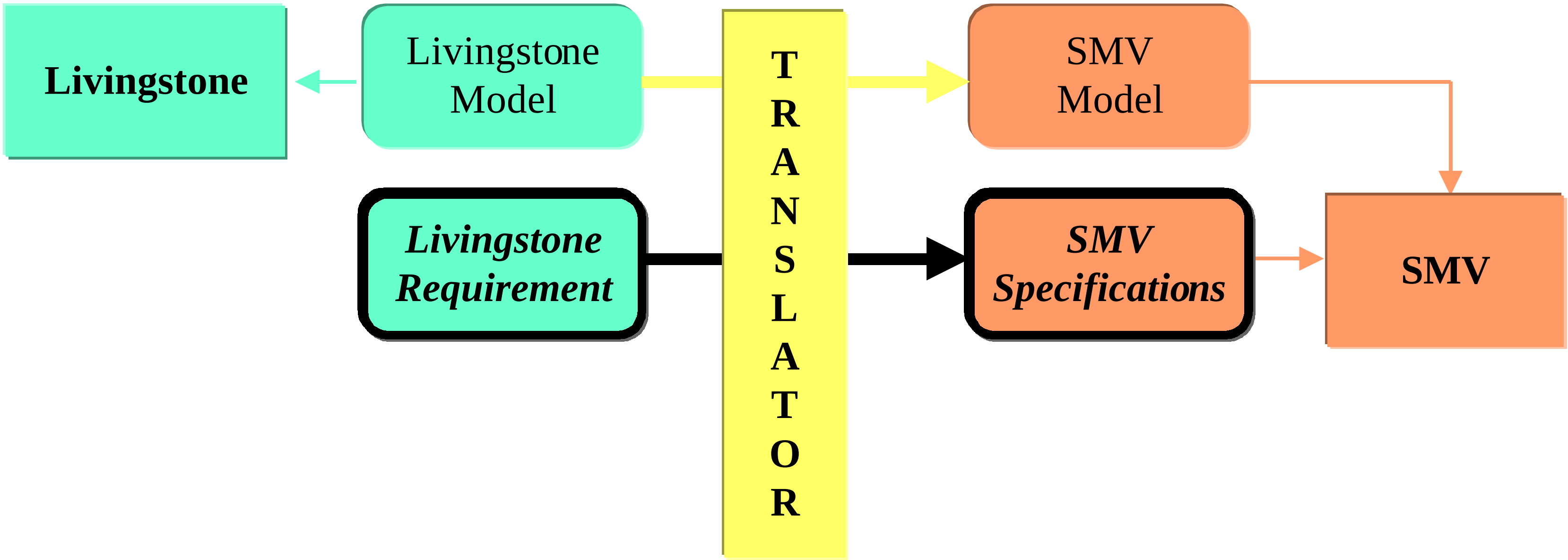
Main difficulty is translating Livingstone's flat name space

Livingstone to SMV

```
(defcomponent valve ()
  (:inputs (cmd-in :type valve-cmd))
  (:outputs (valve-position
                :type open-closed-type))
 ...
  (Closed :type ok-mode
    :model (open (valve-position))
    :transitions
      ((do-open :when (open cmd)
        :next Open) ...))
  (StuckC :type :fault-mode ...)
  ...)
```

```
MODULE valve
VAR   mode: {Open,Closed,
                     StuckO,StuckC};
          valve-position: {Open, Closed};
          cmd-in: {open,close};
DEFINE faults:={StuckO,StuckC};
TRANS
  (mode=Closed & cmd-in=open) ->
    (next(mode) in {Open union faults})
INVAR
 (mode=Open -> valve-position=Open)
```

Requirements


Livingstone

Livingstone Model

Livingstone Requirement

TRANSLATOR

SMV Model

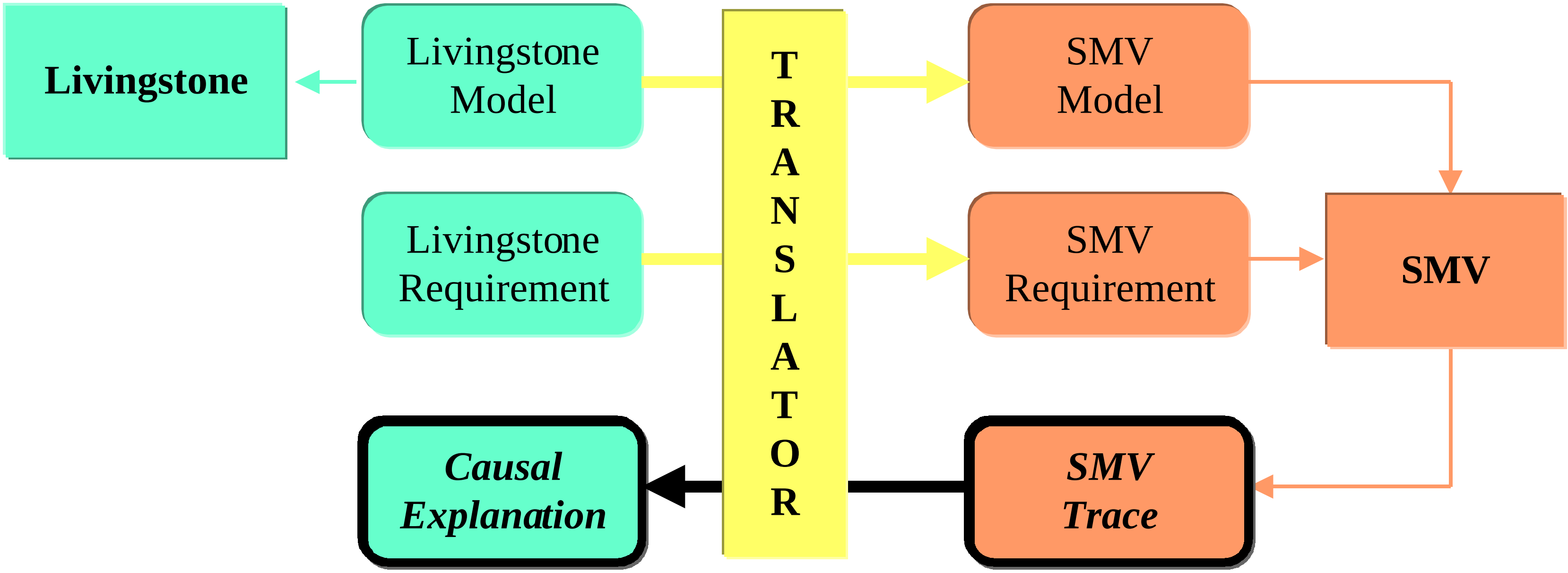
SMV Specifications

SMV

Specifying Properties

- Extend Livingstone to specify CTL properties directly
  - (all (globally (implies (off (admittance outlet))
    (off (flow z-flow-module)))))
- Add high-level properties
  - Completeness, consistency, reachability, …
- Add auxiliary predicates
  - broken, failed, multibroken, ...

Explanations



Livingstone

Livingstone Model

Livingstone Requirement

Causal Explanation

TRANSLATOR

SMV Model

SMV Requirement

SMV

SMV Trace

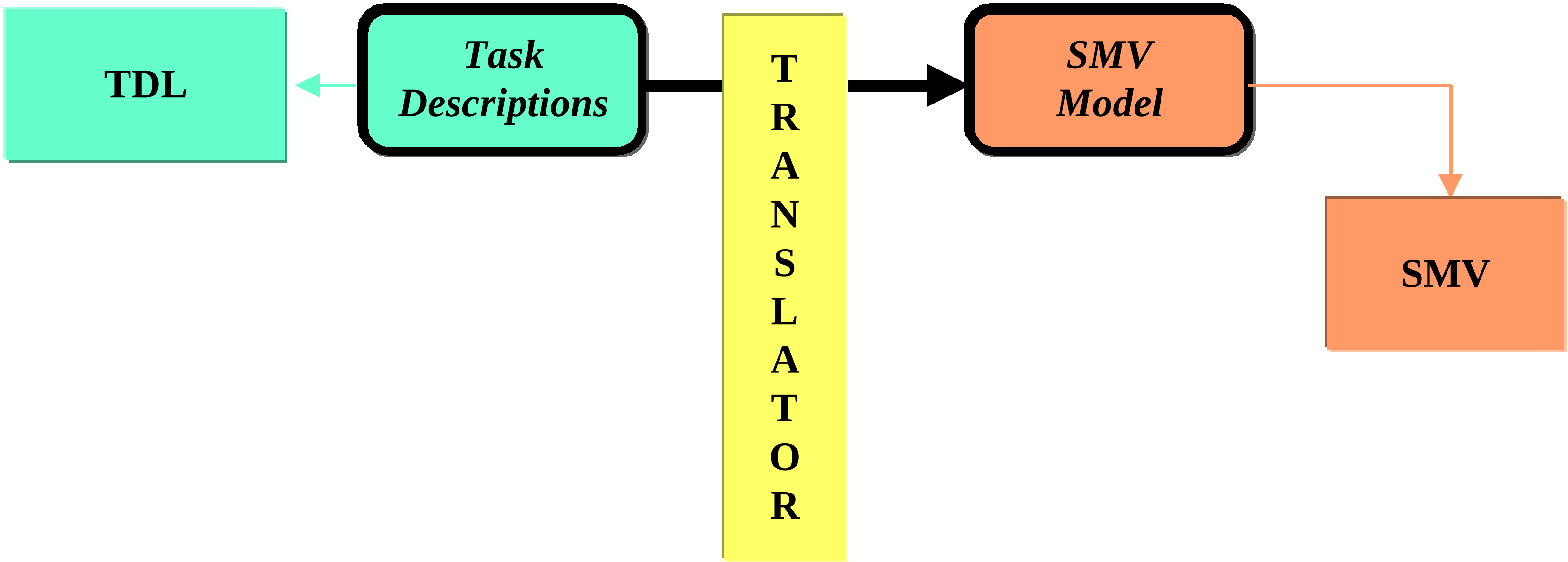
Explaining Witnesses

- Use Truth Maintenance System (TMS)
  - Recreate chain of inferences
  - Record dependencies
  - Generate explanation

```
(AG (NOT (EQ VDECU.DRIVER.MODE FAILED))) is false because
In State 1
  1. VDECU.DRIVER.MODE is initially OFF
  2. VDECU.DRIVER.CMD-OUT is NO-COMMAND
     based on 1 and
     vdecu.driver.mode = off -> vdecu.driver.cmd-out = no-command
  3. VDECU.DRIVER.CMD-OUT is not ON
     based on 2 and EXCLUSIVE-VALUE
In State 2
  4. VDECU.DRIVER.MODE non-deterministically transitions to FAILED
     based on 1, 3, and
     vdecu.driver.mode = off_ & !vdecu.driver.cmd-out = on_ -> next(vdecu.driver.mode) in (off union failed)
  5. (NOT (EQ VDECU.DRIVER.MODE FAILED)) is FALSE
      based on 4
```

Translating TDL

- TDL: *Task Description Language*
  - Extension of C++
  - Task decomposition, task synchronization, monitoring, exception handling

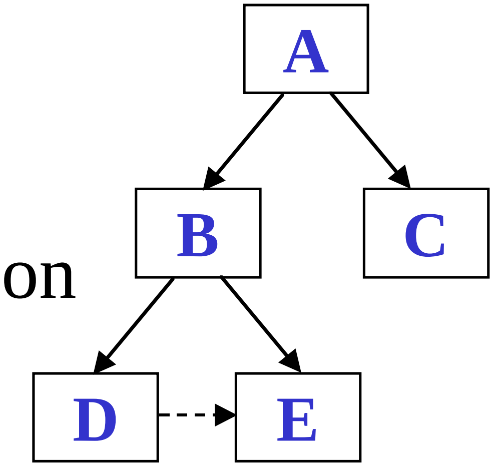


Formalizing the Model

TDL	**SMV**
task	module
task/subtask relationship	module variables
task state	scalar variables
state transitions	ASSIGN
temporal constraints	INVAR and parameters
asynchronous nature	***PROCESS variables and FAIRNESS constraints***

Verifying Task Descriptions

- Can verify temporal properties of hierarchical tasks
  - deadlock, safety, liveness, …
  - can handle conditional execution



- Working on:
  - monitoring and exception handling
  - iteration and recursion

Verifying the ESL Engine

- *Executive Support Language* (ESL)
  - – Built on top of multi-threaded Lisp
- Verify whether implementation matches requirements
  - – Create abstract model of code in PROMELA
  - – Verify properties of interest over all possible execution traces
  - – Found several subtle bugs in the code
    - See paper in LFM 2000 proceedings!

Example: Property Locks

- **Property Lock**: Similar to a semaphore
  - Must be released when task terminates
- The Bug:
  - Task body is wrapped by code to catch exceptions and to release locks (***in that order***)
  - Problem arises if exception is raised while trying to release locks
  - Solution: Surround lock-release code in a critical section

Summary

- Automatic Translation of Special-Purpose Languages for Autonomy Software

- Extensions for Specifying Requirements Directly

- Tools for Analyzing Counter-Examples

References

- C. Pecheur and R. Simmons. “From Livingstone to SMV: Formal Verification for Autonomous Spacecrafts”. *First Goddard Workshop on Formal Approaches to Agent-Based Systems*, NASA Goddard, April 5-7, 2000.
- R. Simmons and C. Pecheur. “Automating Model Checking for Autonomous Systems”. *AAAI Spring Symposium on Real-Time Autonomous Systems*, Stanford CA, March 2000.
- K. Havelund, M. Lowry, S. Park, C. Pecheur, J. Penix, W. Visser, J.L. White. “Formal Analysis of the Remote Agent - Before and After Flight”. *Fifth NASA Langley Formal Methods Workshop*, Virginia, June 2000.

Model Checking Programs

Willem Visser

RIACS / NASA Ames

Model Checking Programs

- Model checking usually applied to designs
  - Some errors get introduced after designs
  - Design errors are missed due to lack of detail
  - Sometimes there is no design
- Can model checking find errors in real programs?
  - Yes, many examples in the literature
- Can model checkers be used by programmers?
  - Only if it takes real programs as input

Main Issues

- Memory
  - Explicit-state model checking’s Achilles heel
  - State of a software system can be complex
  - Require efficient encoding of state, or,
  - State-less model checking
- Input notation not supported
  - Translate to existing notation
  - Custom-made model checker
- State-space Explosion

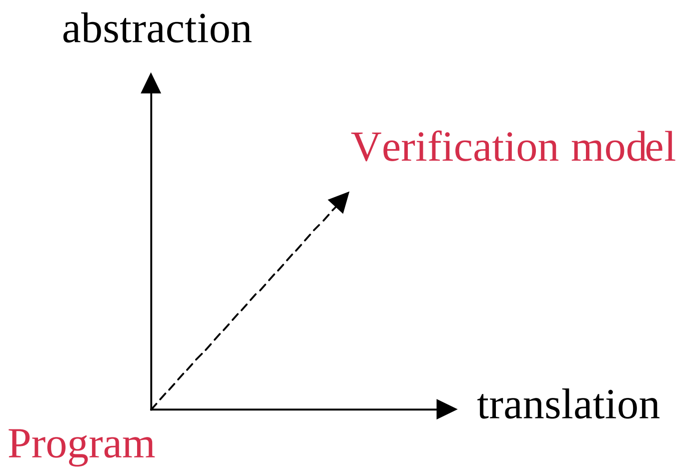
State-less Model Checking

- Must limit search-depth to ensure termination
- Based on partial-order reduction techniques
- Annotate code to allow verifier to detect "important" transitions
- Examples include
  - VeriSoft
    - http://www1.bell-labs.com/project/verisoft/
  - Rivet
    - http://sdg.lcs.mit.edu/rivet.html

Traditional Model Checking

- Translation-based using existing model checker
  - Hand-translation
  - Semi-automatic translation
  - Fully automatic translation
- Custom-made model checker
  - Fully automatic translation
  - More flexible

Hand-Translation



- Hand translation of program to model checker’s input notation
- “Meat-axe” approach to abstraction
- Labor intensive and error-prone

Hand-Translation Examples

- Remote Agent – Havelund,Penix,Lowry 1997
    - http://ase.arc.nasa.gov/havelund
    - Translation from Lisp to Promela (most effort)
    - Heavy abstraction
    - 3 man months
- DEOS – Penix *et al.* 1998/1999
    - http://ase.arc.nasa.gov/visser
    - C++ to Promela (most effort in environment)
    - Limited abstraction - programmers produced sliced system
    - 3 man months

Semi-Automatic Translation

- Table-driven translation and abstraction
  - Feaver system by Gerard Holzmann
  - User specifies code fragments in C and how to translate them to Promela (SPIN)
  - Translation is then automatic
  - Found 75 errors in Lucent’s PathStar system
  - http://cm.bell-labs.com/cm/cs/who/gerard/
- Advantages
  - Can be reused when program changes
  - Works well for programs with long development and only local changes

Fully Automatic Translation

- Advantage
  - – No human intervention required
- Disadvantage
  - – Limited by capabilities of target system
- Examples
  - – Java PathFinder 1- http://ase.arc.nasa.gov/havelund/jpf.html
    - Translates from Java to Promela (Spin)
  - – JCAT - http://www.dai-arc.polito.it/dai-arc/auto/tools/tool6.shtml
    - Translates from Java to Promela (or dSpin)
  - – Bandera - http://www.cis.ksu.edu/santos/bandera/
    - Translates from Java bytecode to Promela, SMV or dSpin

Custom-made Model Checkers

- Allows efficient model checking
  - Often no translation is required
  - Algorithms can be tailored
- Translation-based approaches
  - dSpin
    - Spin extended with dynamic constructs
    - Essentially a C model checker
    - http://www.dai-arc.polito.it/dai-arc/auto/tools/tool7.shtml
  - Java Model Checker (from Stanford)
    - Translates Java bytecode to SAL language
    - Custom-made SAL model checker
    - http://sprout.stanford.edu/uli/

Java PathFinder 2

- Based on new Java Virtual Machine
  - Handle all of Java, since it works with bytecodes
- Written in Java
  - 1 month to develop version with only integers
- Efficient encoding of states
  - Complex states are translated to integer vector
  - Garbage collection
  - Canonical heap representation
- http://ase.arc.nasa.gov/jpf

Reducing the State Space

- Partial-order reductions
  - Vital for efficient explicit-state model checking
  - Must be able to identify independent transitions
    - Static analysis
- Abstraction
  - Under-approximations
    - Slicing, i.e. a cultured “meat-axe”
  - Over-approximations
    - Predicate abstraction
    - Type-based abstraction

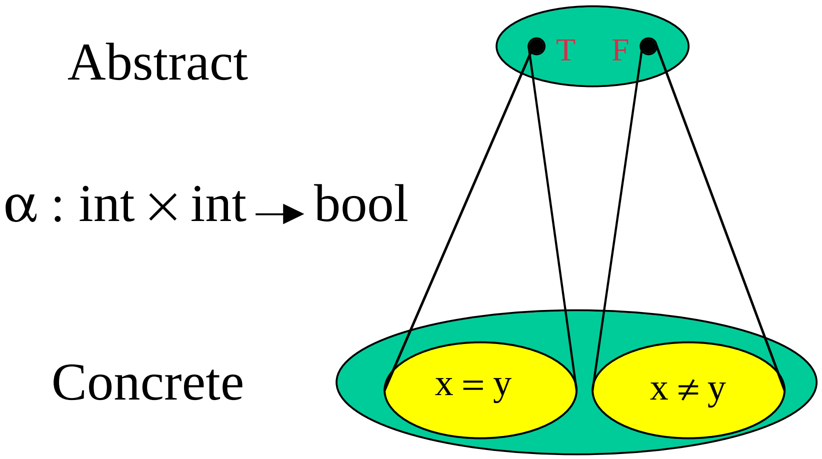
Slicing in JPF

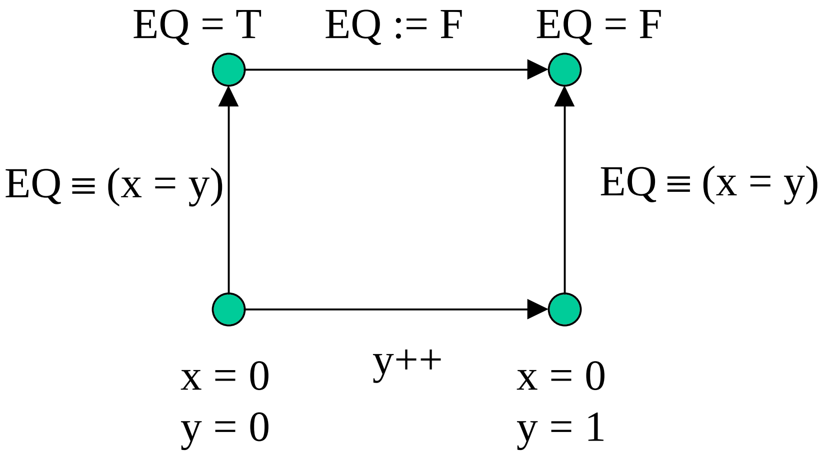
- JPF uses Bandera's slicer
- Bandera slices w.r.t.
  - Deadlock - i.e. communication statements
  - Variables occurring in temporal properties
  - Variables participating in race-violations
    - Used with JPF's runtime analysis
- More examples of slicing for model checking
  - Slicing for Promela (Millet and Teitelbaum)
    - http://netlib.bell-labs.com/netlib/spin/ws98/program.html
  - Slicing for Hardware Description Languages (Shankar *et al.*)
    - http://www.cs.wisc.edu/~reps/

Predicate Abstraction

- Create abstract state-space w.r.t. set of predicates defined in concrete system
  - Abstract interpretation
- First proposed by Graf and Saidi
  - http://www.csl.sri.com/~saidi/
  - http://www-verimag.imag.fr/~graf/
  - see also http://theory.stanford.edu/people/uribe/
- Only applies to static programs, that manipulates global variables
  - Not directly applicable to object-oriented programs

Predicate Abstraction

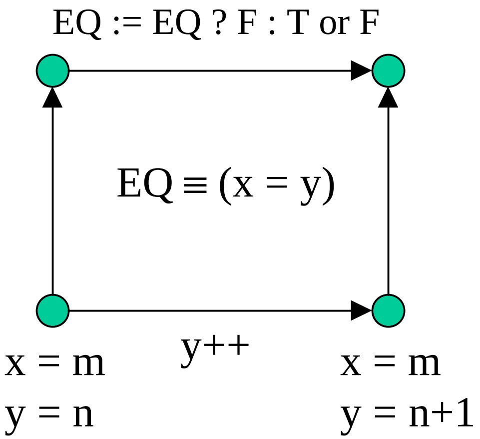




- Mapping of a concrete system to an abstract system, whose states correspond to truth values of a set of predicate
- Create abstract state-graph during model checking, or,
- Create an abstract transition system before model checking

JPF Abstraction Technique

- Find abstraction mapping (α) by user guidance
- Use decision procedures to automatically compute *abstract interpretation* of concrete transitions
- Validity checking of pre-images
- Over approximation with nondeterminism



EQ := EQ ? F : T or F

EQ ≡ (x = y)

x = m
y = n

y++

x = m
y = n+1

JPF’s Java Abstraction

- Annotations used to indicate abstractions

  -
    ```
    Abstract.remove(x);
    Abstract.remove(y);
    Abstract.addBoolean(ÒEQÓ, x==y);
    ```
- Tool generates abstract Java program
  - Using Stanford Validity Checker (SVC)
  - JVM is extended with nondeterminism to handle over approximation
- Abstractions can be local to a class or global across multiple classes
  - `Abstract.addBoolean(ÒEQÓ, A.x==B.y);`
  - Dynamic predicate abstraction, since it works across instances

Conclusions

- Model checking programs is an active field
  - At least 5 groups are checking Java
- Model checking needs some help
  - Static analysis
  - Abstraction – abstract interpretation
  - Runtime analysis
    - Gathering information during one run through the code to guide the model checker towards errors
    - Next talk

Runtime Analysis of Programs

Klaus Havelund

Recom / QSS / NASA Ames

The State Space Explosion Problem

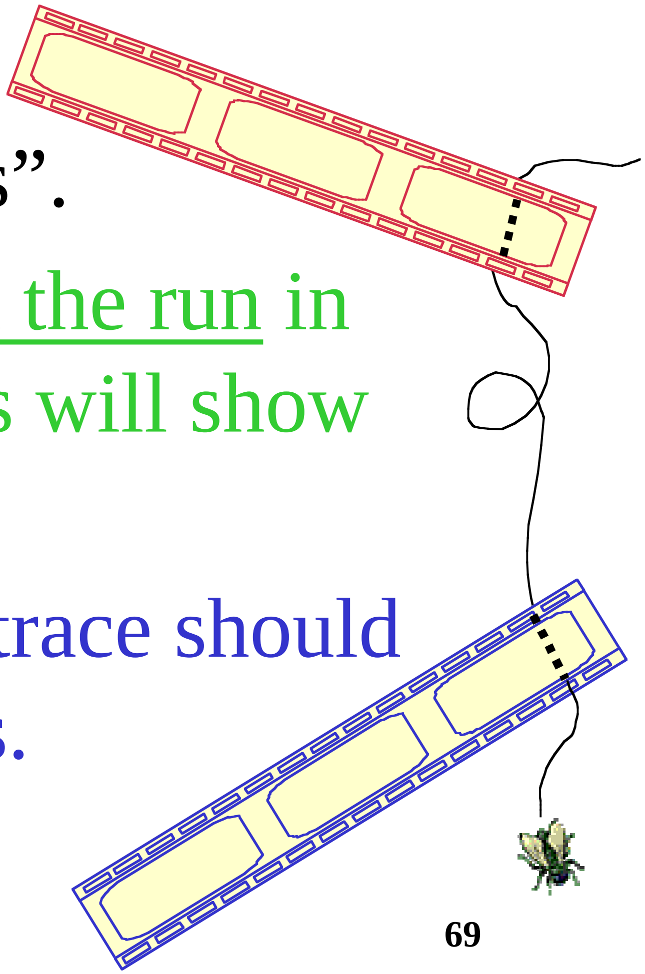
- Real programs have too many states for unfocused model checking.
- The model checker needs to be focused on program fragments that “matter”.
- Abstraction is the solution.
- However, we probably need complementary techniques which can examine the state space in a less complete way.
- We also need guided model checking.

Are there Other Solutions?

Solutions which can find errors in multi threaded programs, and which do not require repeated test runs?

Yes: Runtime Analysis!

- Conclude properties of a program from a single run of the program.
- Look for the bug’s “foot prints”.
- Bug does not have to occur in the run in order to be detected. Examples will show this.
- Goal: the choice of execution trace should not influence result of analysis.



How to do Runtime Analysis

- Run the program once.
- Collect information about run in a database. What information depends on the property being analyzed.
- Database is analyzed “on-the-fly” or after (a forced) program termination.
- Warnings are issued in case the contents of the database suggests that properties can be violated in this or other runs.

Runtime Analysis Plusses and Minuses

+ Scales well (one trace) + Often finds the bugs it is supposed to find	- Gives false positives - Gives false negatives - Limited to special classes of bugs

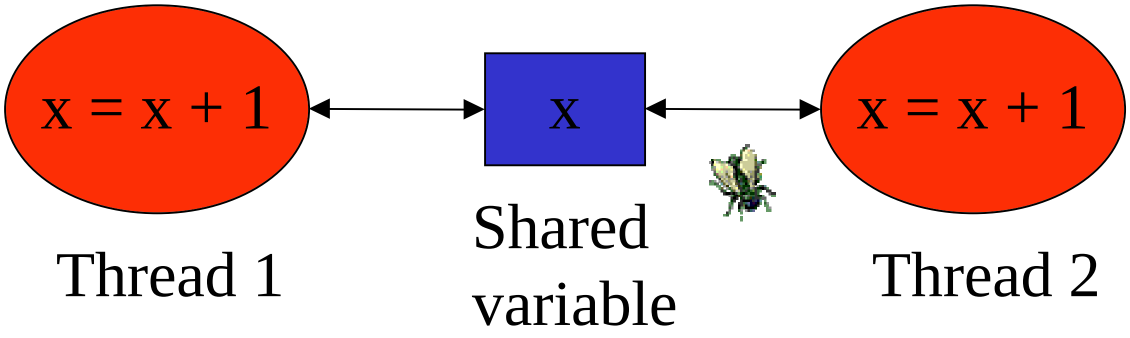
Two Examples of Runtime Analysis

- **Data race detection**: detects simultaneous access to unprotected variables by several threads.
- **Deadlock detection**: detects deadlocks between threads that access shared resources.

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, …

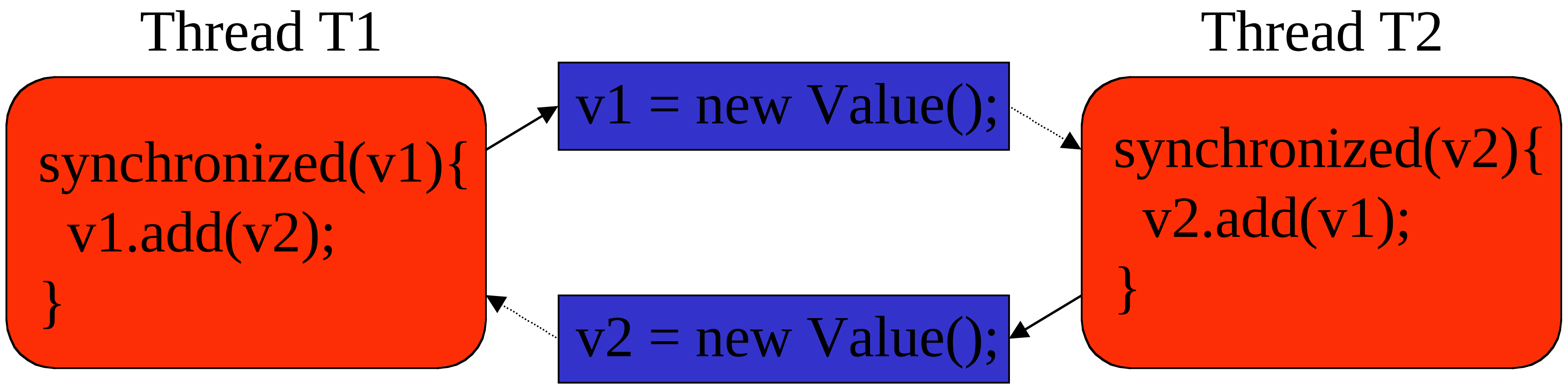


Data Race Detection

- The **Eraser** algorithm (Savage,Burrows,Nelson,Sobalvarro).
- Detects data race potentials by observing execution trace - keeping track of which locks are active when variables are accessed.

Example Java Program

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



Thread T1

```
synchronized(v1){
  v1.add(v2);
}
```

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

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

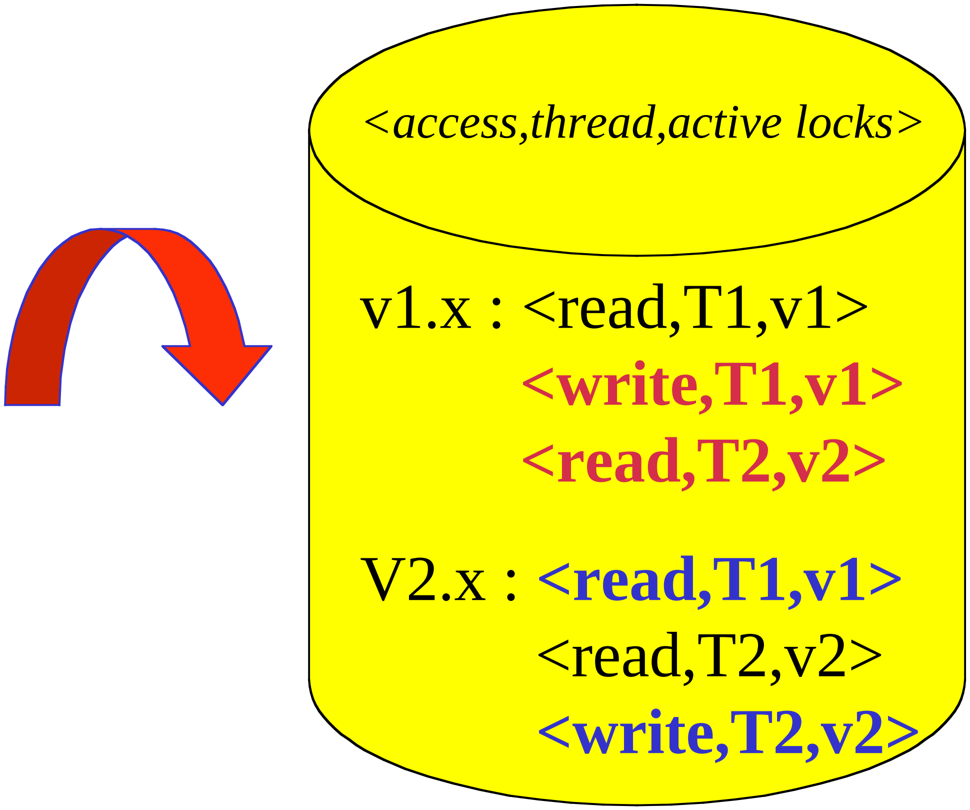
Thread T2

```
synchronized(v2){
  v2.add(v1);
}
```

Examining a Run

0: T1.monitorenter(v1);
1: T1.getfield(v1.x);
2: T1.getfield(v2.x);
3: T1.putfield(v1.x);
4: T1.monitorexit(v1);

5: T2.monitorenter(v2);
6: T2.getfield(v2.x);
7: T2.getfield(v1.x);
8: T2.putfield(v2.x);
9: T2.monitorexit(v2);



<access,thread,active locks>

v1.x : <read,T1,v1>
<write,T1,v1>
<read,T2,v2>

V2.x : **<read,T1,v1>**
<read,T2,v2>
<write,T2,v2>

The Basic Algorithm

set(t) : set of locks owned by thread t
set(x) : set of locks protecting variable x

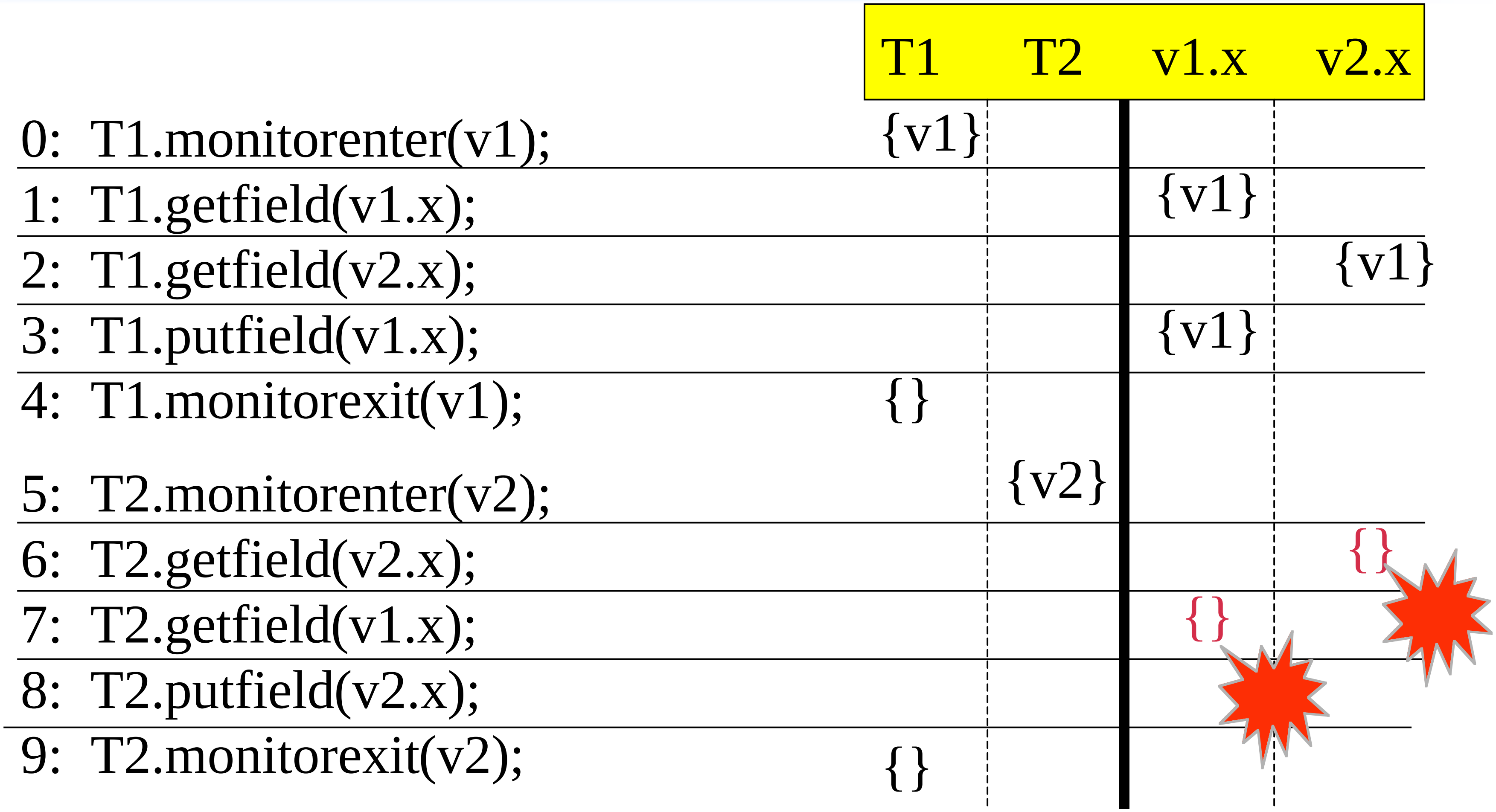
```
takeLock(t,l)
  set(t) = set(t) union {l}

releaseLock(t,l)
  set(t) = set(t) minus {l}
```

```
firstAccess(t,x)
  set(x) = set(t)
```

Lock refinement

```
laterAccess(t,x)
  set(x) = set(x) intersect set(t);
  if set(x) == {} then warning
```

Examining Run using Basic Algorithm

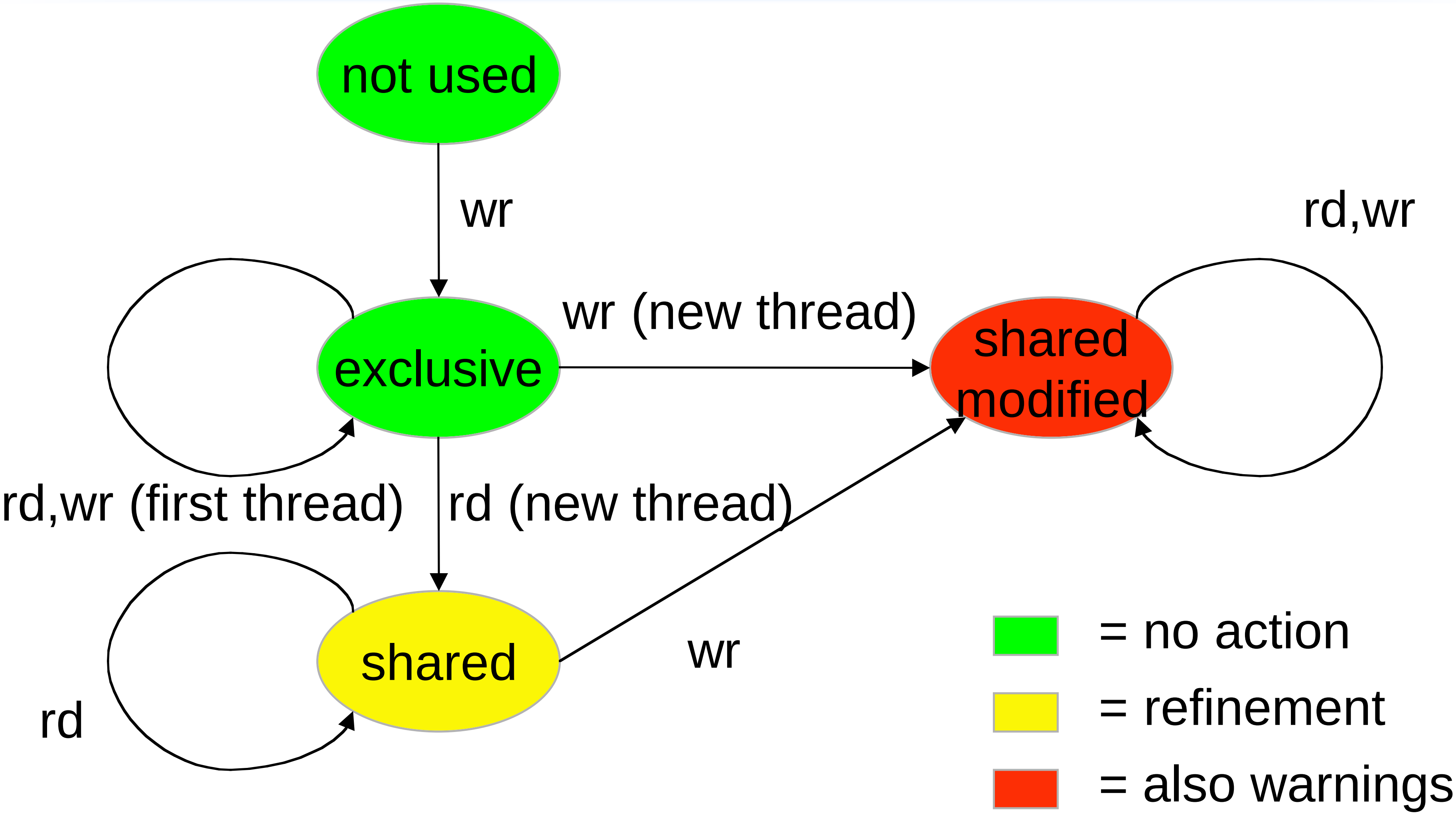


	T1	T2	v1.x	v2.x
0: T1.monitorenter(v1);	{v1}			
1: T1.getfield(v1.x);			{v1}	
2: T1.getfield(v2.x);				{v1}
3: T1.putfield(v1.x);			{v1}	
4: T1.monitorexit(v1);	{}			
5: T2.monitorenter(v2);		{v2}		
6: T2.getfield(v2.x);				{}
7: T2.getfield(v1.x);			{}	
8: T2.putfield(v2.x);				
9: T2.monitorexit(v2);	{}			

Basic Algorithm Yields False Positives

- **Initialization/single threaded use:** usually done without locks.
- **Shared read access:** several threads should be allowed to read if no-one writes after the initialization.

The Extended Algorithm



not used

wr

exclusive

wr (new thread)

shared modified

rd,wr

rd,wr (first thread)

rd (new thread)

shared

wr

rd

🟩 = no action

🟨 = refinement

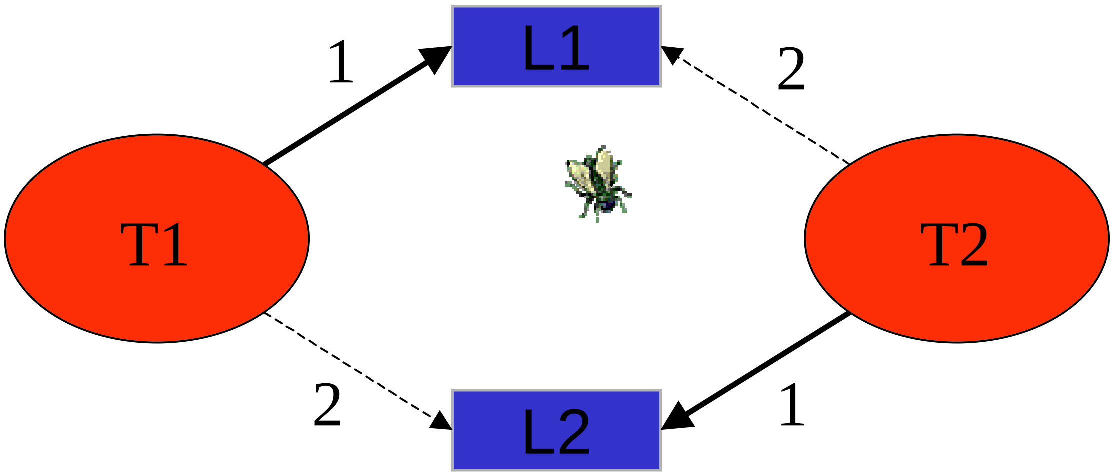
🟥 = also warnings

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

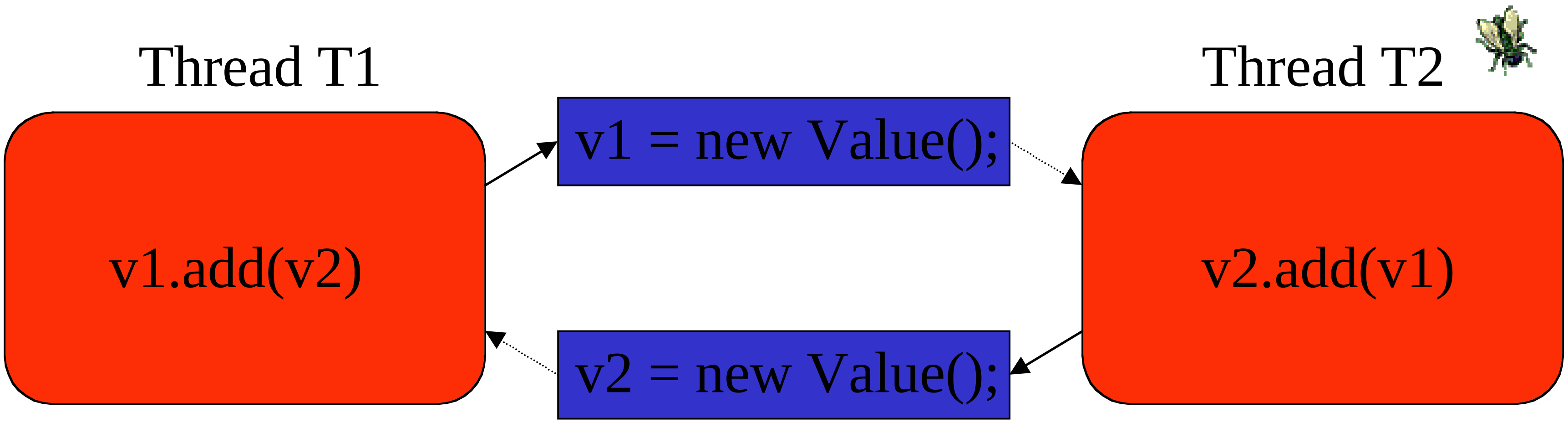


Deadlock Detection

- The **GoodLock** algorithm (Havelund).
- Detects deadlock potentials by observing execution trace - keeping track of which locks are taken by threads, and in which order they are taken.

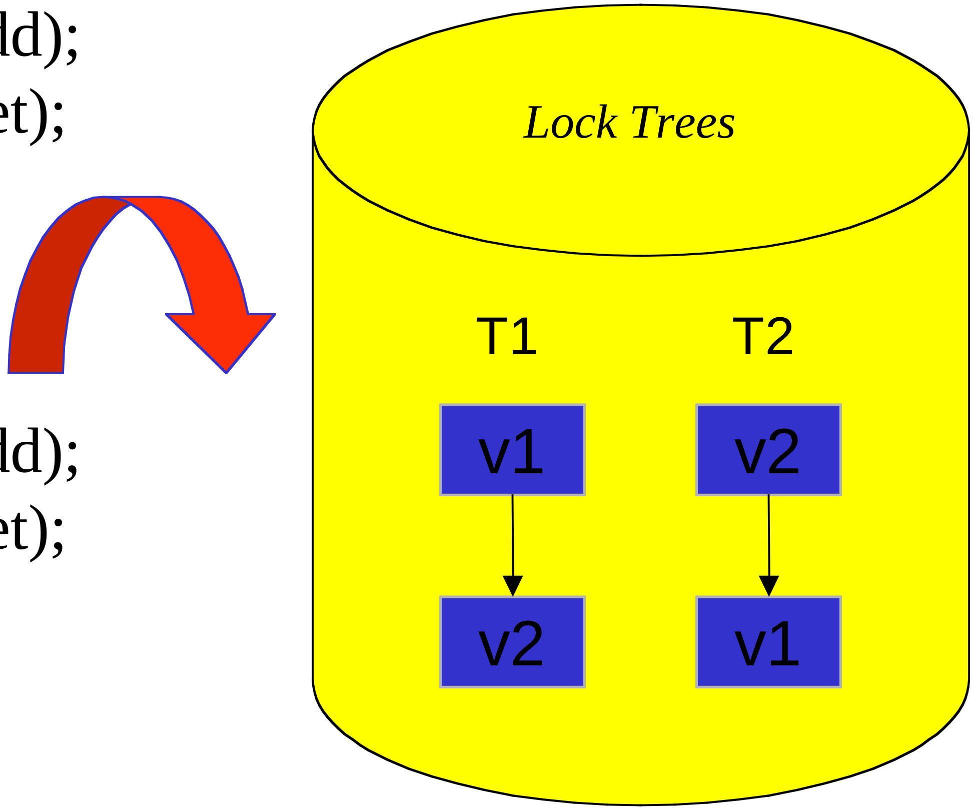
Modified Java Program

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



Examining a Run

```
0:  T1.invokevirtual(v1.add);
1:  T1.invokevirtual(v2.get);
2:  T1.return(v2.get);
3:  T1.return(v1.add);

4:  T2.invokevirtual(v2.add);
5:  T2.invokevirtual(v1.get);
6:  T2.return(v1.get);
7:  T2.return(v2.add);
```



More Elaborate Example

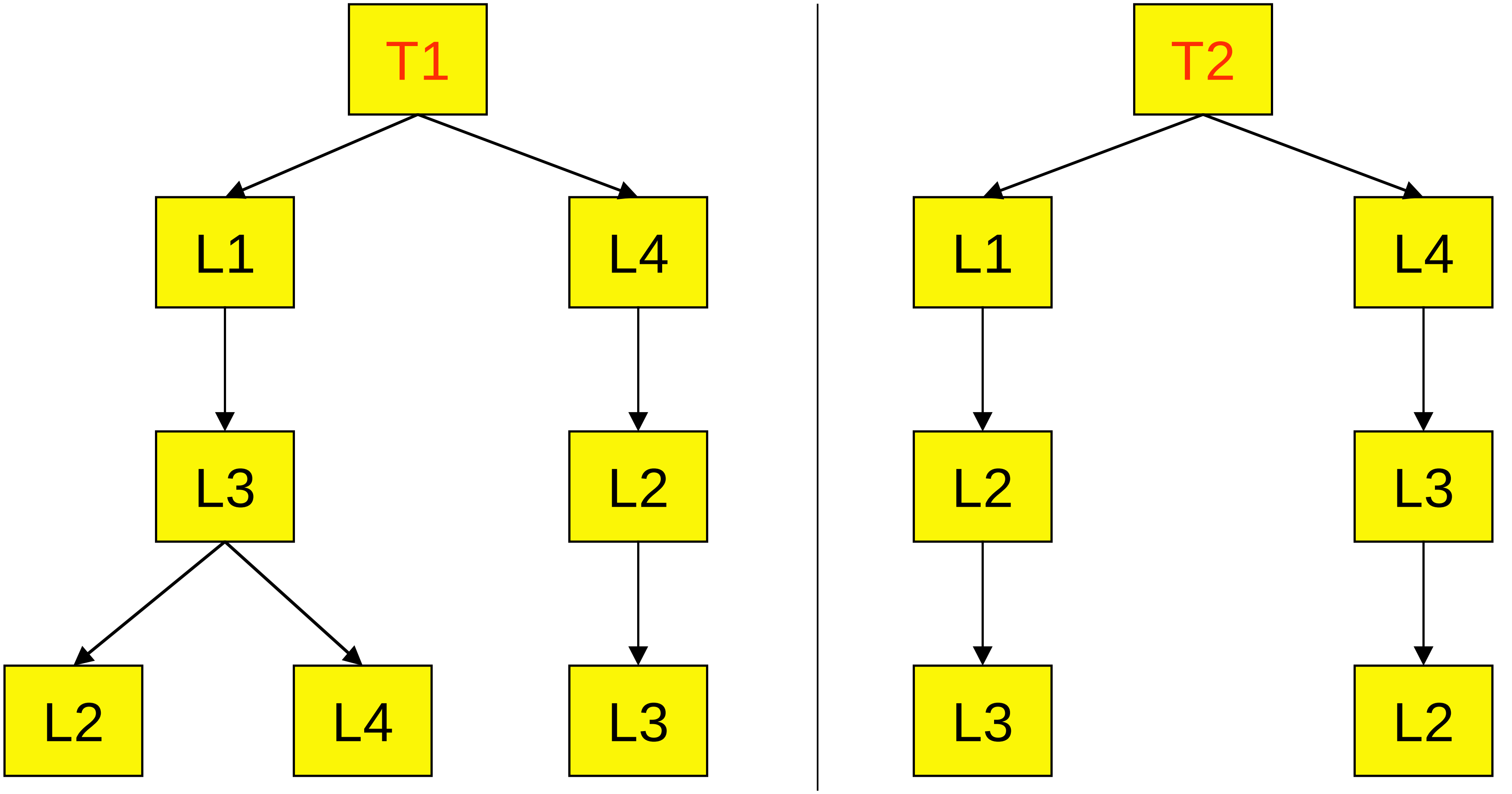
Thread T1:

```
synchronized(L1){
  synchronized(L3){
    synchronized(L2){};
    synchronized(L4){}
  }
};
synchronized(L4){
  synchronized(L2){
    synchronized(L3){}
  }
}
```

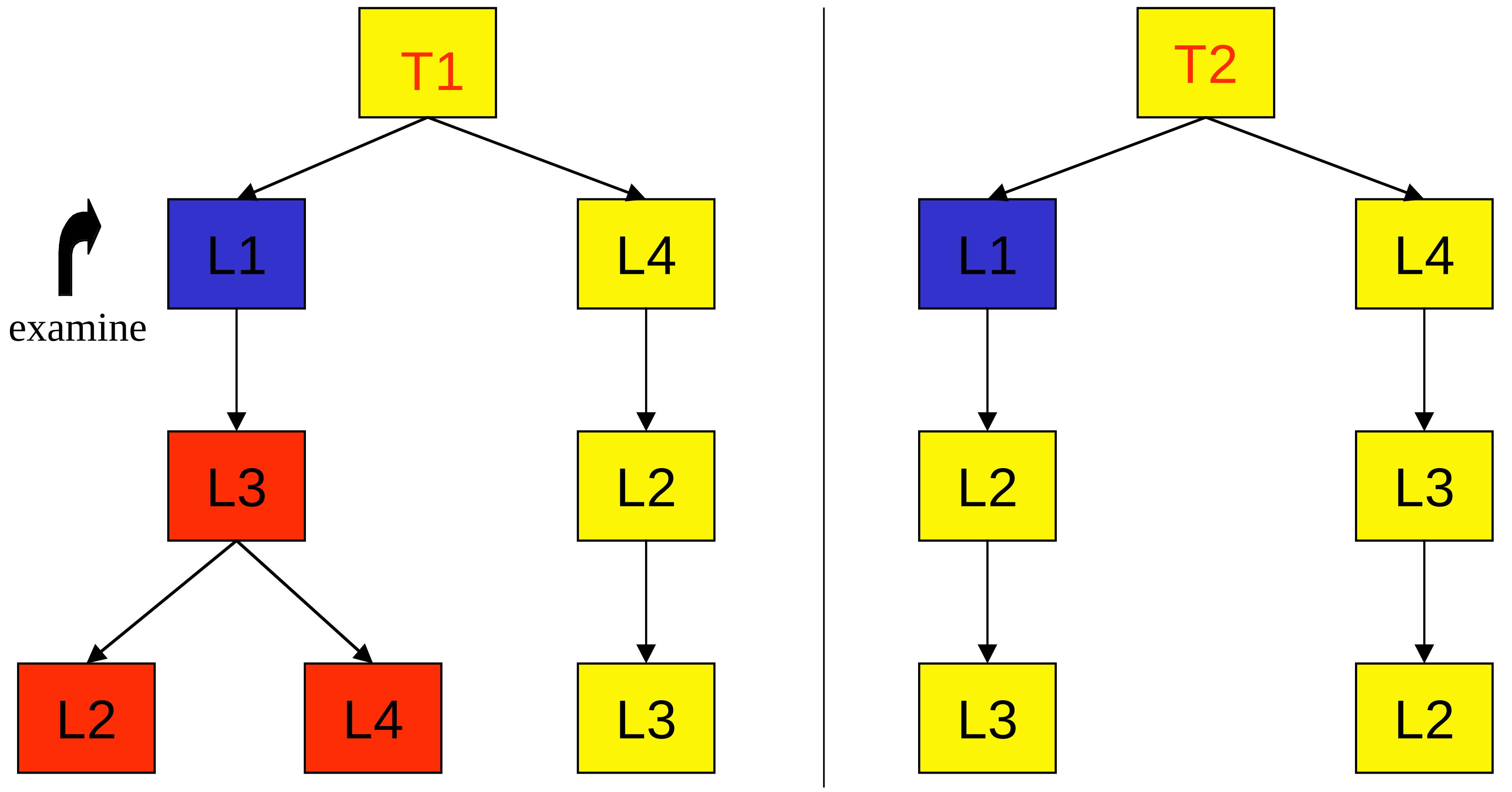
Thread T2:

```
synchronized(L1){
  synchronized(L2){
    synchronized(L3){}
  }
};
synchronized(L4){
  synchronized(L3){
    synchronized(L2){}
  }
}
```

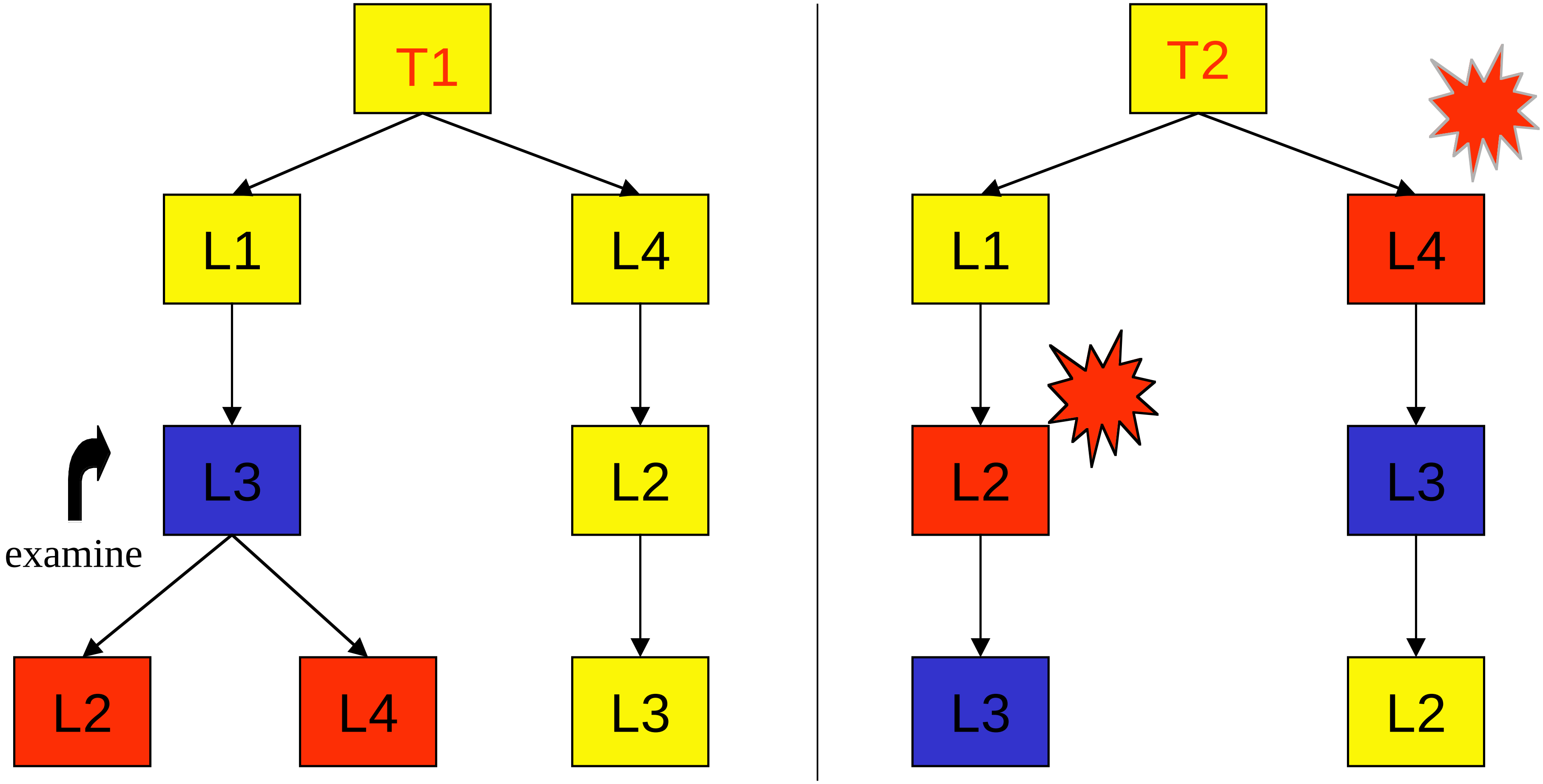
Create Lock Trees During Run



Analyze Lock Trees After Run



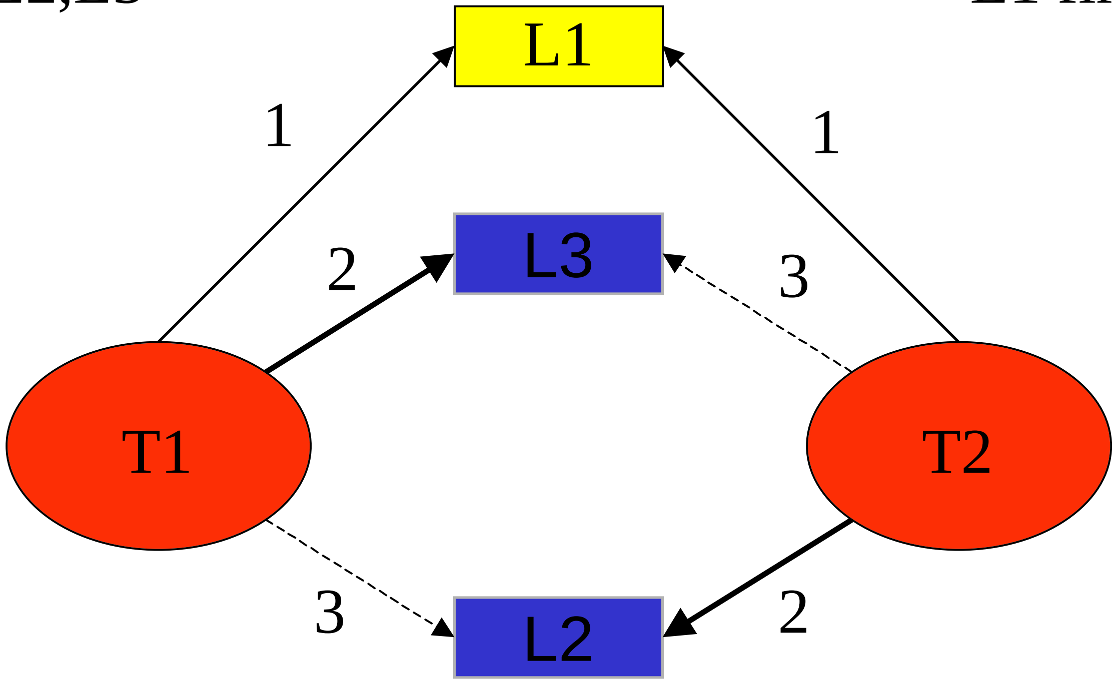
Examine L3 in T1’s Left Branch



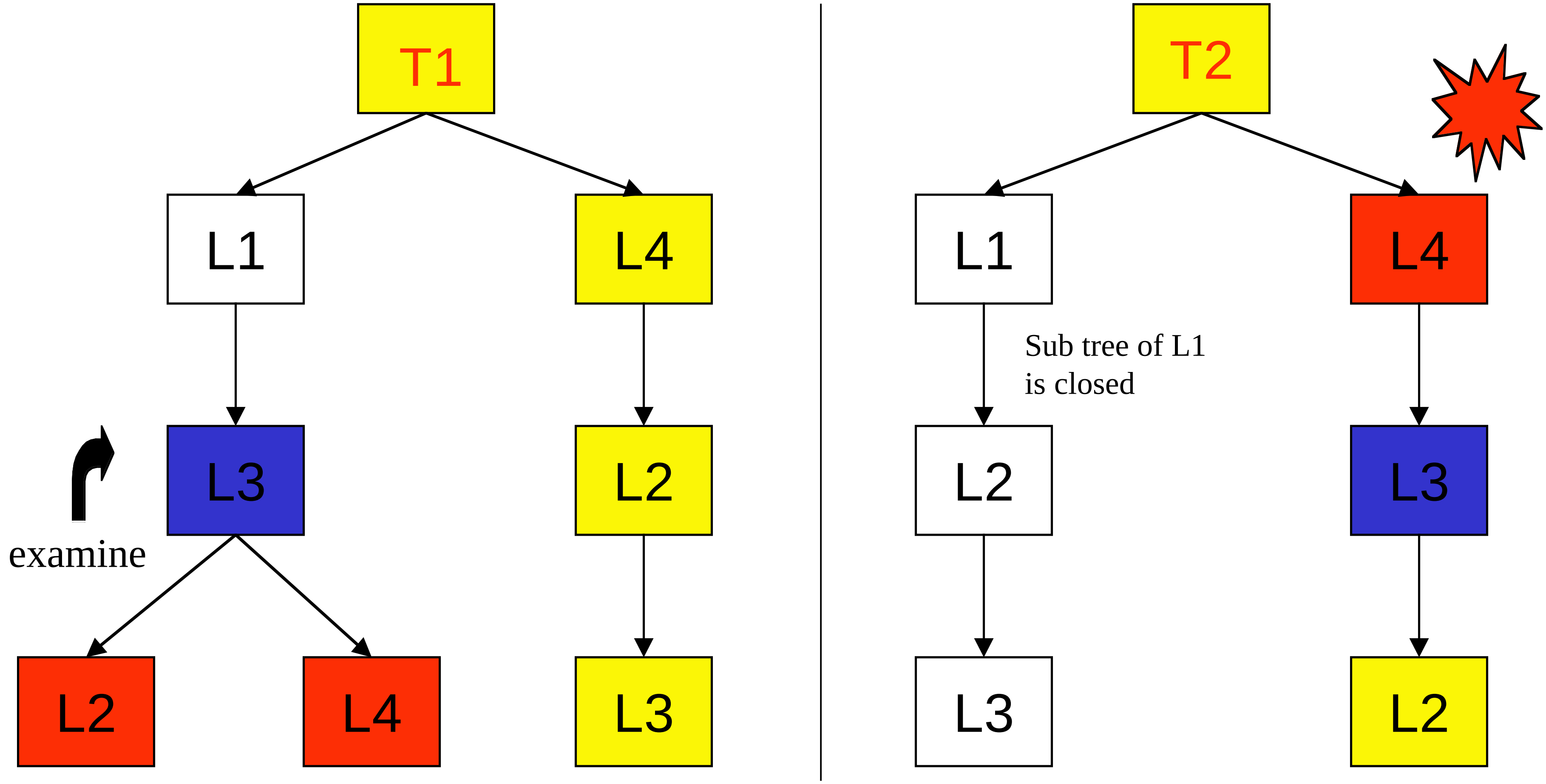
Basic Algorithm Yields False Positives

The lock L1 protects against <L2,L3> deadlock:

Both threads take L1 first.

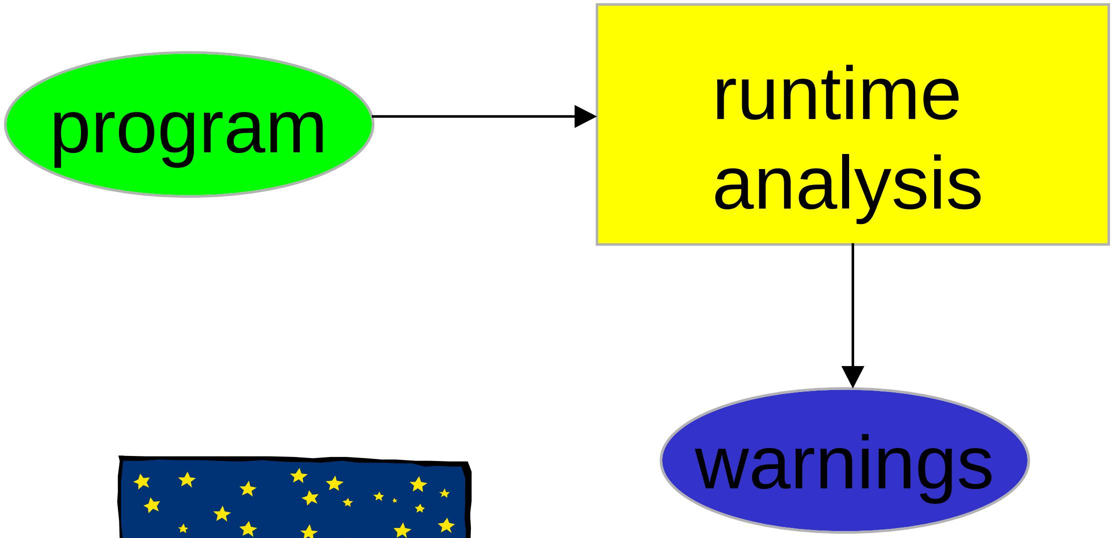


Close L1 Tree in T2 After Examination of L1



Sub tree of L1 is closed

How to Interpret Warnings from Analysis

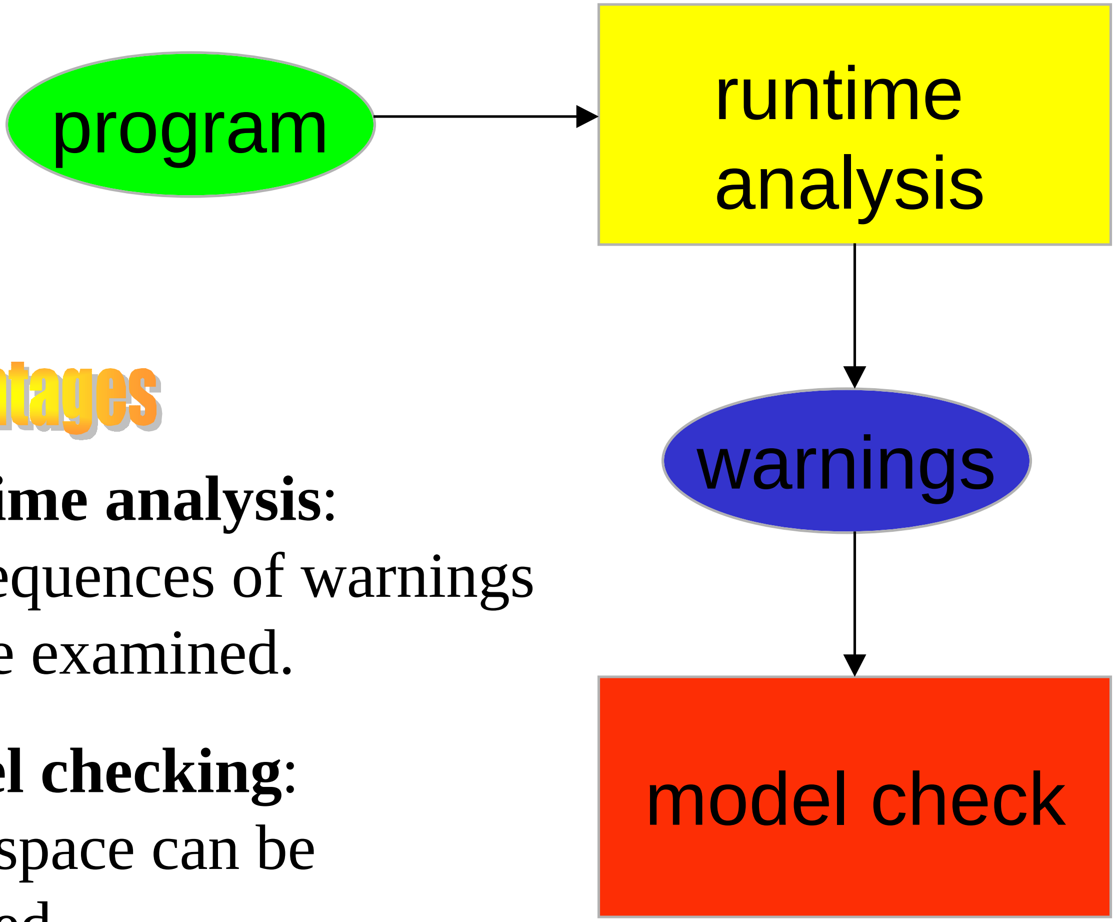


program

runtime analysis

warnings

Runtime Analysis Can Guide Model Checking


program

runtime analysis

warnings

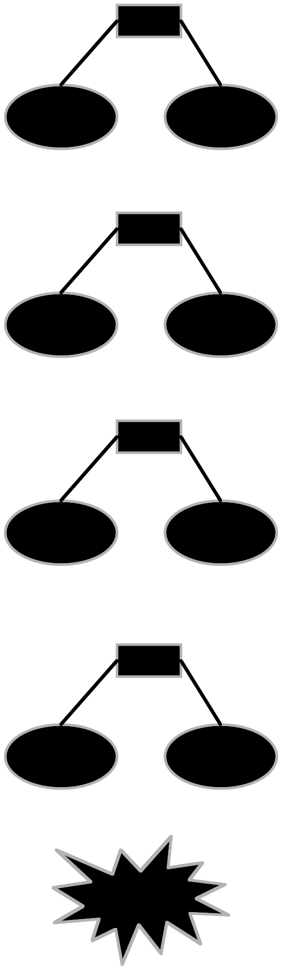
model check

Advantages

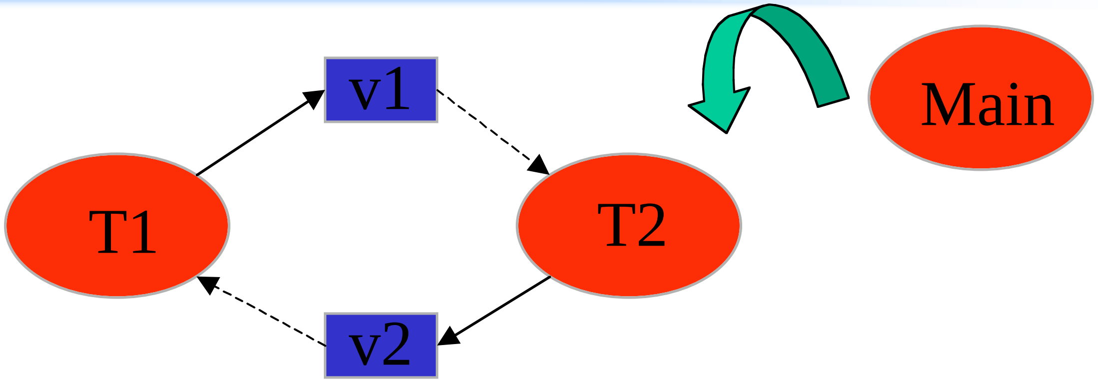
Runtime analysis: Consequences of warnings can be examined.

Model checking: State space can be reduced.

Analyzing a Big State Space



20 groups
in total



Environment: 40 threads, each performing 10.000 assignments to shared variable.
More than 10^{160} states!

Record dependency information:

- Which threads start which threads?
- Which threads read/write which objects?
- Calculate smallest window from warnings!

Result of Running JPF2 on Example

```
Runtime Analysis:
----------------------
…
Thread T1 takes lock on v1
…
EXECUTION INTERRUPTED!
…
```

27 seconds

```
Lock Trees:
--------------
Thread T1:
0   v1
0.0  v2

Thread T2:
0  v2
0.0  v1
…
```

```
Lock Order Conflict:
------------------------
Locks on v1 and v2 are
taken in opposite order.

Lock on v2 is taken last by T1
  Value.add line 4
  Task.run line 17
Lock on v1 is taken last by T2
  Value.add line 4
  Task.run line 17
```

```
Dependencies:
------------------
Task T1:
  creater : Main
  reads   :  v1,v2
  writes  :  v1
…
```

```
Window Extension:
-----------------------
Warning Window   : T1, T2
Extended Window : Main, T1, T2
```

```
Model Checking of
Extended Window:
-----------------------
*** Deadlock ***
… error trail …
```

2 seconds

Conclusions and Future Work

- Deadlock occurred on board Deep-Space 1 due to missing critical section. Eraser can find the error.
- Minimize false positives.
- Generalize deadlock algorithm to N.
- Alternative kinds of runtime analysis.
- Runtime analysis *during* model checking.
- Optimize: only analyze shared objects, …
- Feed warnings to static slicing tool (Bandera).
- Investigate how useful runtime analysis is, and generalize.

References

- "Eraser: A Dynamic Data Race Detector for Multithreaded Programs", S. Savage, M. Burrows, G. Nelson, P. Sobalvarro. http://camars.kaist.ac.kr/etc/SOSP16/PAPERS/SAVAGE/SAVAGE.HTM
- "Using Runtime Analysis to Guide Model Checking of Java Programs", K. Havelund. http://ase.arc.nasa.gov/havelund