


**tutorial:  
hardware and software  
model checking**

*gerard holzmann and anuj puri*

**{ gerard | anuj } @research.bell-labs.com**

*Bell Labs, USA*

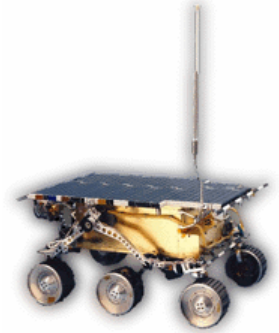
# outline

- introduction (15 mins) 
- theory and algorithms
  - system modeling and specifications (25 mins)
  - algorithms and data structures (25 mins)
- practice (25 minutes)
  - the role of abstraction
  - model checking tools
  - some applications

# what is model checking ?

- a **model** of the hardware or software system
- a **specification**
- Question: does system satisfy specification?
- key is to build a **model** that is an appropriate **abstraction** of the system
- a model checking tool answers:
  - *yes* -- if the model definitely satisfies the specification
  - *no* -- in which case it provides the user a **counter-example**

# the mars pathfinder

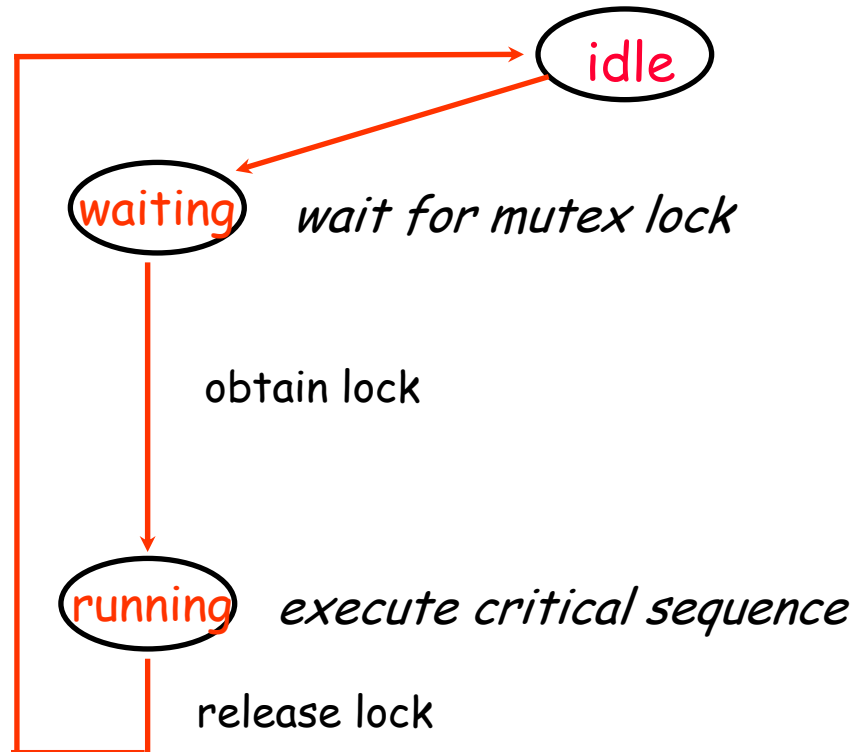


## An Example:

The required properties for the Mars Pathfinder control software included:

- mutual exclusion rules
  - a process cannot access the databus unless it owns a mutual exclusion lock
- scheduling priority rules
  - a lower priority process cannot execute when a higher priority process is ready to execute
  - saving data to memory, for instance, has higher priority than processing data

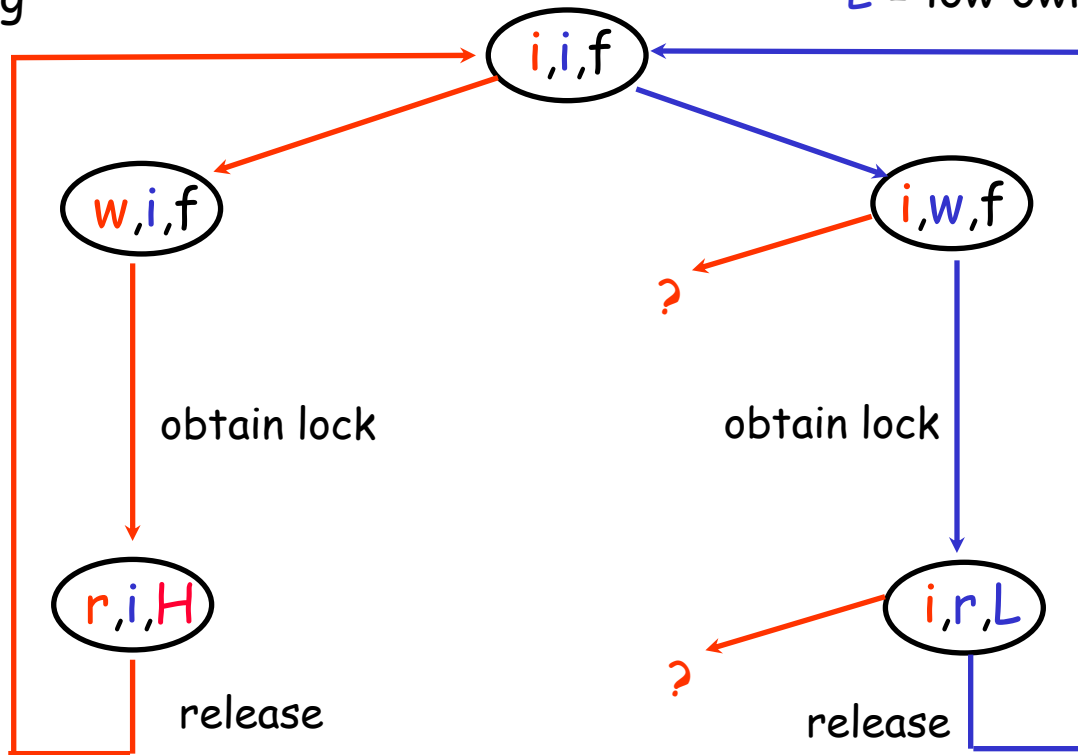
# the basic sequence for access to the shared bus



# priority rule

i = idle  
w = waiting  
r = running

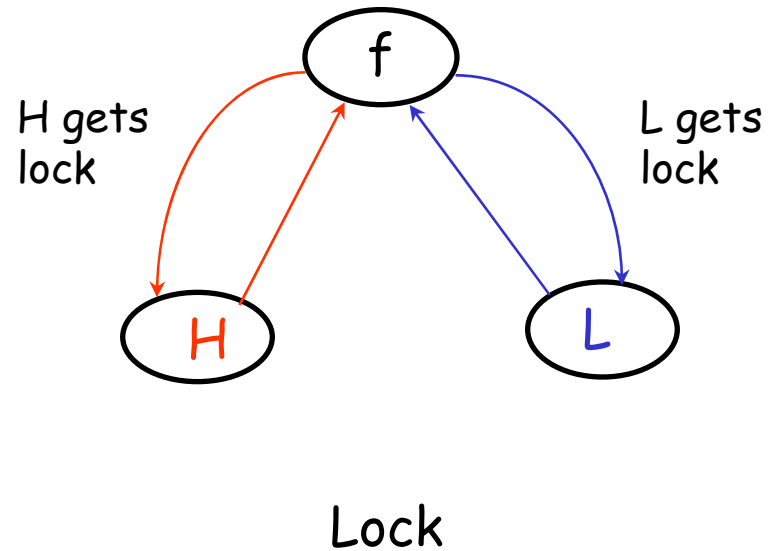
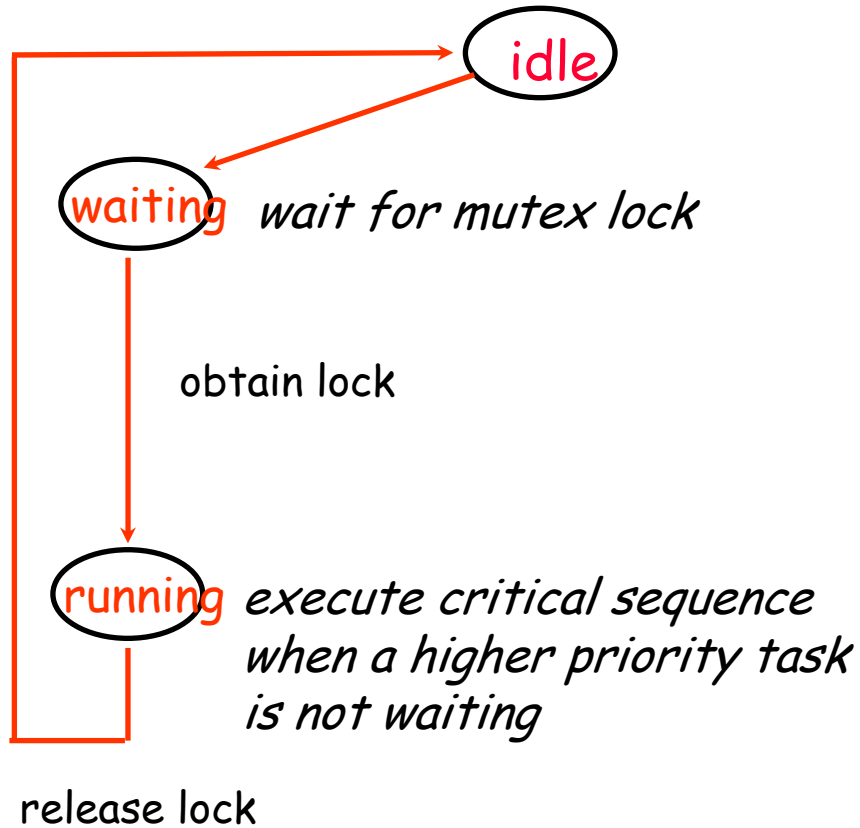
f = mutex lock free  
H = high owns lock  
L = low owns lock



High priority process  
(dumping measurements)  
**restricted by lock**

Low priority process  
(processing data)  
cannot run unless: i,-,-

# system model

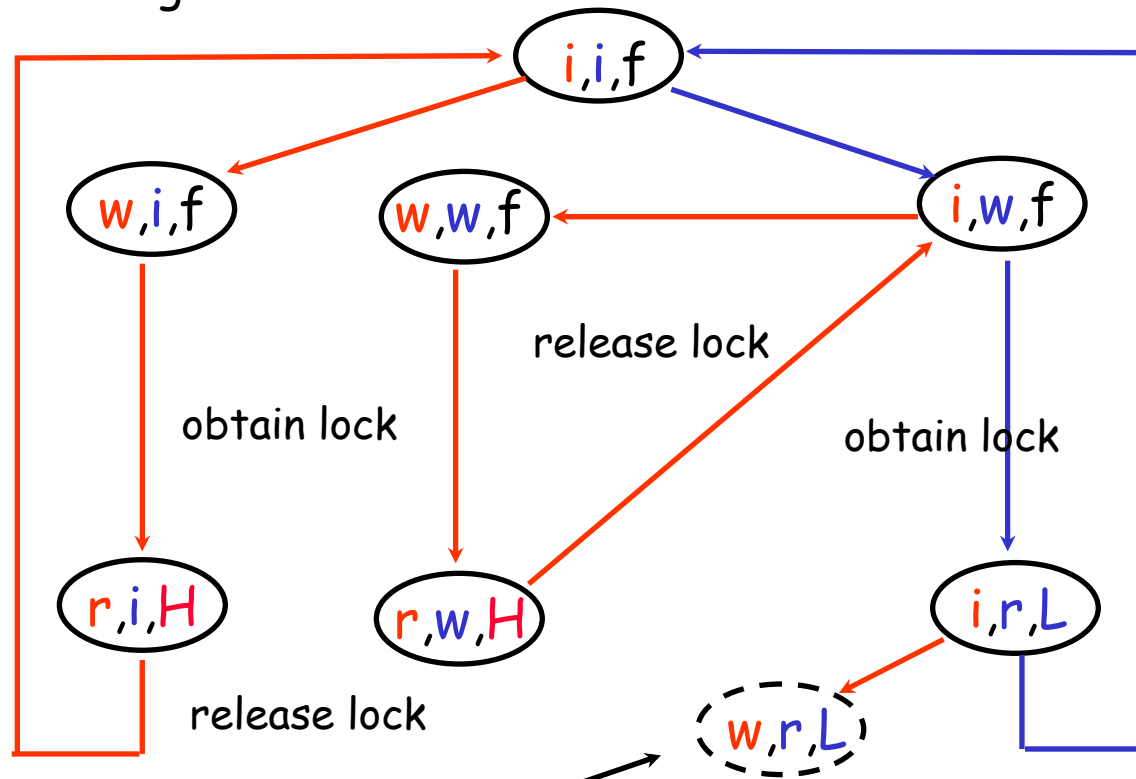


high priority and low priority tasks

# model checking run

i = idle  
w = waiting  
r = running

f = mutex lock free  
H = high owns lock  
L = low owns lock



the hangup problem  
from July 1997


# model checking procedure

- build a model of the system abstracting out irrelevant details
- write a specification
  - deadlocks, reachability issues, system invariants etc.
- run the model checker
  - performs some variation of reachability analysis on the *finite state* model
- the possible answers are:
  - yes -- specification satisfied
  - no -- a counter-example provided

# defining issues

- how do you **formalize**
  - modeling issues
  - specification
- how do you **search for counter-examples**
  - explicit state model checking
  - symbolic model checking
- how do you **control complexity**
  - avoiding or containing state space explosion
  - abstraction and reduction techniques

# outline

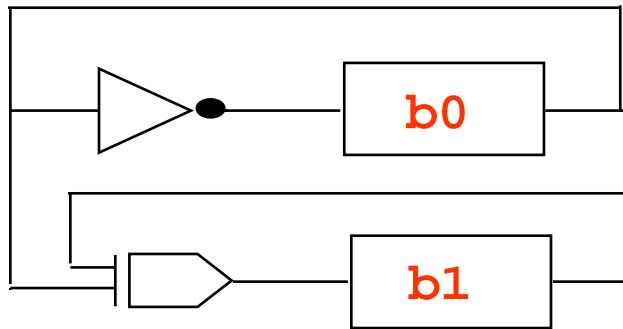
- introduction (15 mins)
- theory and algorithms 
  - system modeling and specifications (25 mins)
  - algorithms and data structures (25 mins)
- practice (25 minutes)
  - the role of abstraction
  - model checking tools
  - some applications

# system modeling

hardware:

- typically synchronous, clock-driven
- bit-registers, signal wires, gates
- modeled with hardware description language

2 bit counter:



**b1 b0** : 00 -> 01 -> 10 -> 11 -> 00

# system modeling (cont.)

## software:

- asynchronous, distributed processes
- message passing, shared variables, higher level data structures, semaphores, rv-ports

## SPIN model for the Pathfinder:

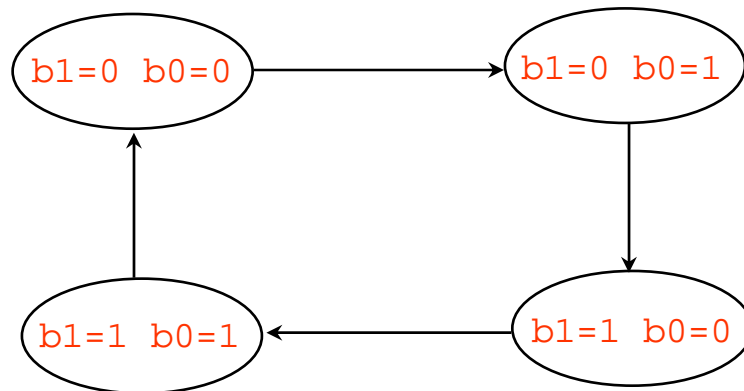
```
active proctype low_priority()
  provided (h_state == idle) {
end: do
  :: l_state = waiting;
  atomic { mutex == free ->
          mutex = busy };
  l_state = running;
  /* consumes data */
  atomic { l_state = idle;
          mutex = free };
od }
```

```
active proctype high_priority (){
end: do
  :: h_state = waiting;
  atomic { mutex == free ->
          mutex = busy };
  h_state = running;
  /* produces data */
  atomic { h_state = idle;
          mutex = free };
od }
```

# system modeling (cont.)

- underlying model is finite automata
- models (hardware description language models or promela models) can be **translated to finite automata**

2-bit counter:



# formalizing specifications

- PLTL (temporal logic)
  - propositional **linear time** logic
  - first suggested by Amir Pnueli in late 70s
- $\omega$ -automata
  - finite automata accepting infinite sequences
- many other logics
  - CTL, u-calculus, many variations

# Syntax of PLTL

- `formula ::=`
  - `true, false`
  - `propositional symbols p, q, r, ...`
  - `( f )`
  - `unary_operator f`
  - `f binary_operator f`
- `unary_operator ::=`
  - `[]` --- `always, henceforth`
  - `<>` --- `eventually`
  - `!` --- `logical negation`
- `binary_operator ::=`
  - `U` --- `strong until`
  - `&&` --- `logical and`
  - `||` --- `logical or`
  - `->` --- `implication`
  - `<->` --- `equivalence`

# semantics of PLTL

**Model:**  $s = s[0] s[1] s[2] s[3] \dots$

for each proposition  $p$ ,  
 $s[i] \models p$  is defined

temporal formulas:

$s[i] \models [ ]f$  provided  $\forall j \geq i s[j] \models f$

$s[i] \models <>f$  provided  $\exists j, j \geq i s[j] \models f$

$s[i] \models f \cup g$

$s \models f$  provided  $s[0] \models f$

# typical PLTL formulae

- frequently used properties:

- $[] p$  --- invariance
- $\langle \rangle p$  --- guarantee
- $p \rightarrow (\langle \rangle q)$  --- response
- $p \rightarrow (q \cup r)$  --- precedence
- $[] \langle \rangle p$  --- recurrence (progress)
- $\langle \rangle [] p$  --- stability (non-progress)
- $\langle \rangle p \rightarrow \langle \rangle q$  --- correlation

- two useful equivalences:

$$![] p \Leftrightarrow \langle \rangle !p$$

$$!\langle \rangle p \Leftrightarrow [] !p$$

# model checking PLTL formulae

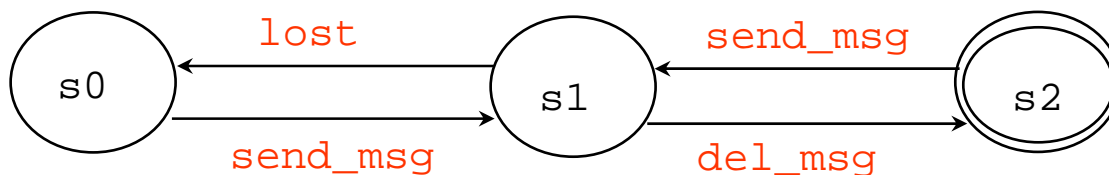
System model:  $S$

Specification: PLTL formula  $f$

How do we model check ?

we need to learn about  $\omega$ -automata ....

# automata theoretic approach



## standard finite automata

$\sigma = \text{send\_msg lost send\_msg del\_msg}$

run = s0            s1            s0            s1            s2

$\sigma$  is **accepted** when s2 is final

## $\omega$ -automata (buchi automata)

$\sigma = \text{send\_msg lost send\_msg del\_msg } \dots$

run = s0            s1            s0            s1            s2     $\dots$

$\sigma$  is **accepted** when s2 is visited **infinitely often**

**language:** set of accepted strings or sequences

# automata theoretic approach(cont.)

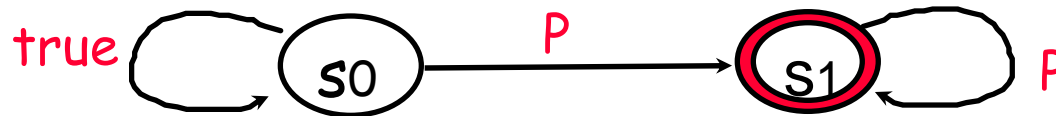
- deterministic vs. non-deterministic
- $\omega$ -automata are closed under all **boolean operations**
  - automata **A** with language **Lang(A)**
  - there exists **C** with **Lang(C) =  $\Sigma^\omega \setminus \text{Lang(A)}$**
  - for **A and B**, there exists **C** with **Lang(C) =  $\text{Lang(A)} \cap \text{Lang(B)}$**
- checking non-emptiness: is there a reachable strongly connected component which contains a final state ?

# automata theoretic approach (cont.)

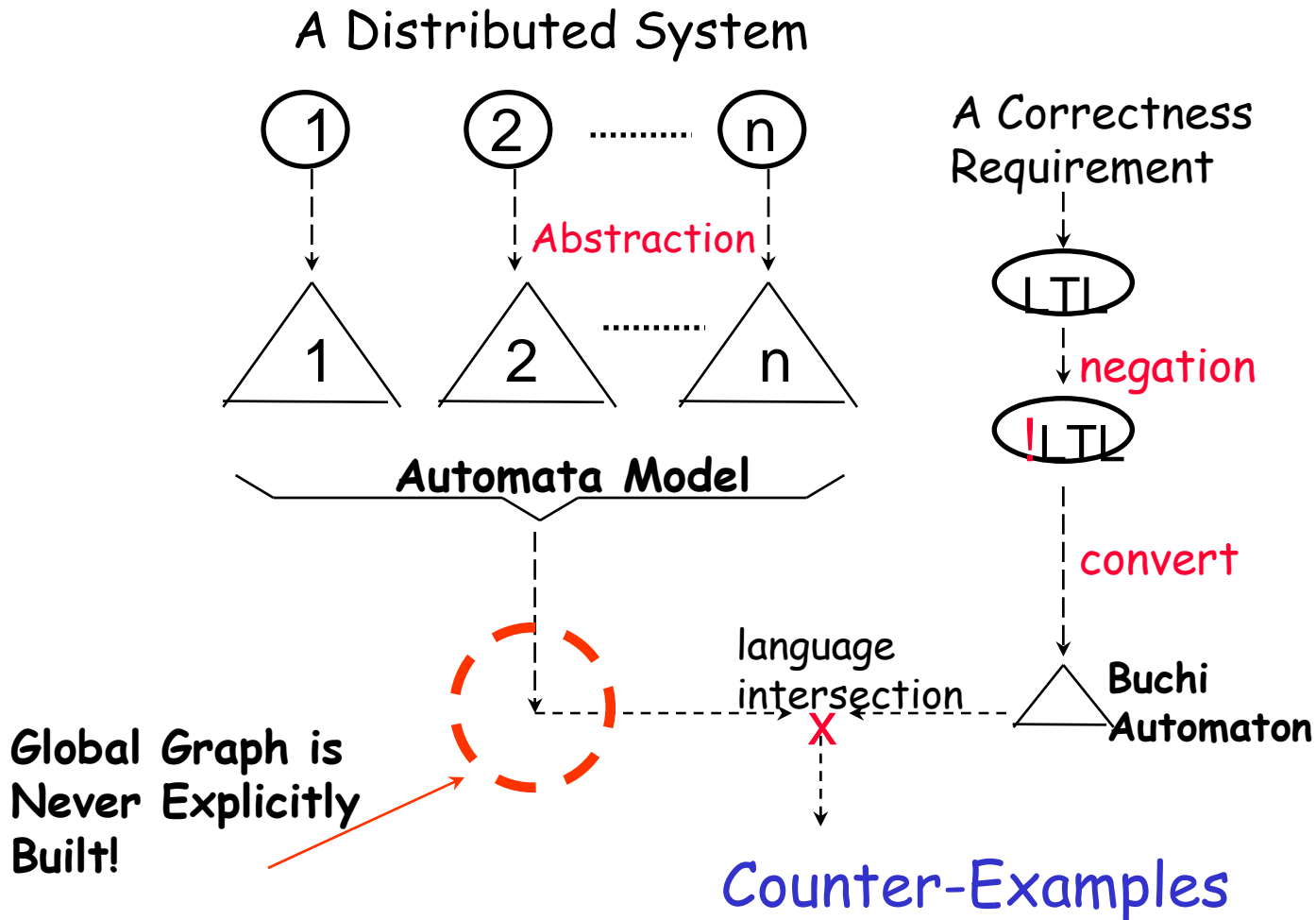
- Model:  $\text{Lang}(\text{Model})$
- Specification - another automaton -  $\text{Lang}(\text{Spec})$
- Model Checking Question:
  - $\text{Lang}(\text{Model}) \subset \text{Lang}(\text{Spec})$
- Algorithm:
  - $\text{Lang}(\text{Model}) \subset \text{Lang}(\text{Spec})$  iff  
 $\text{Lang}(\text{Model}) \cap (\Sigma^\omega \setminus \text{Lang}(\text{Spec})) = \emptyset$
  - Yes/No(counter-example)

# from logic to automata

- The truth of an PLTL formula is defined over execution sequences where  $\square\square\square\square$
- For any LTL formula  $f$  there exists a buchi automaton that accepts precisely those executions for which  $f$  is true
- Example: the LTL formula  $\langle\rangle[] P$  corresponds to the Buchi automaton:



# spin's on-the-fly verification procedure



# outline

- introduction (15 mins)
- theory and algorithms
  - system modeling and specifications (25 mins)
  - algorithms and data structures (25 mins) ←
- practice (25 minutes)
  - the role of abstraction
  - model checking tools
  - some applications

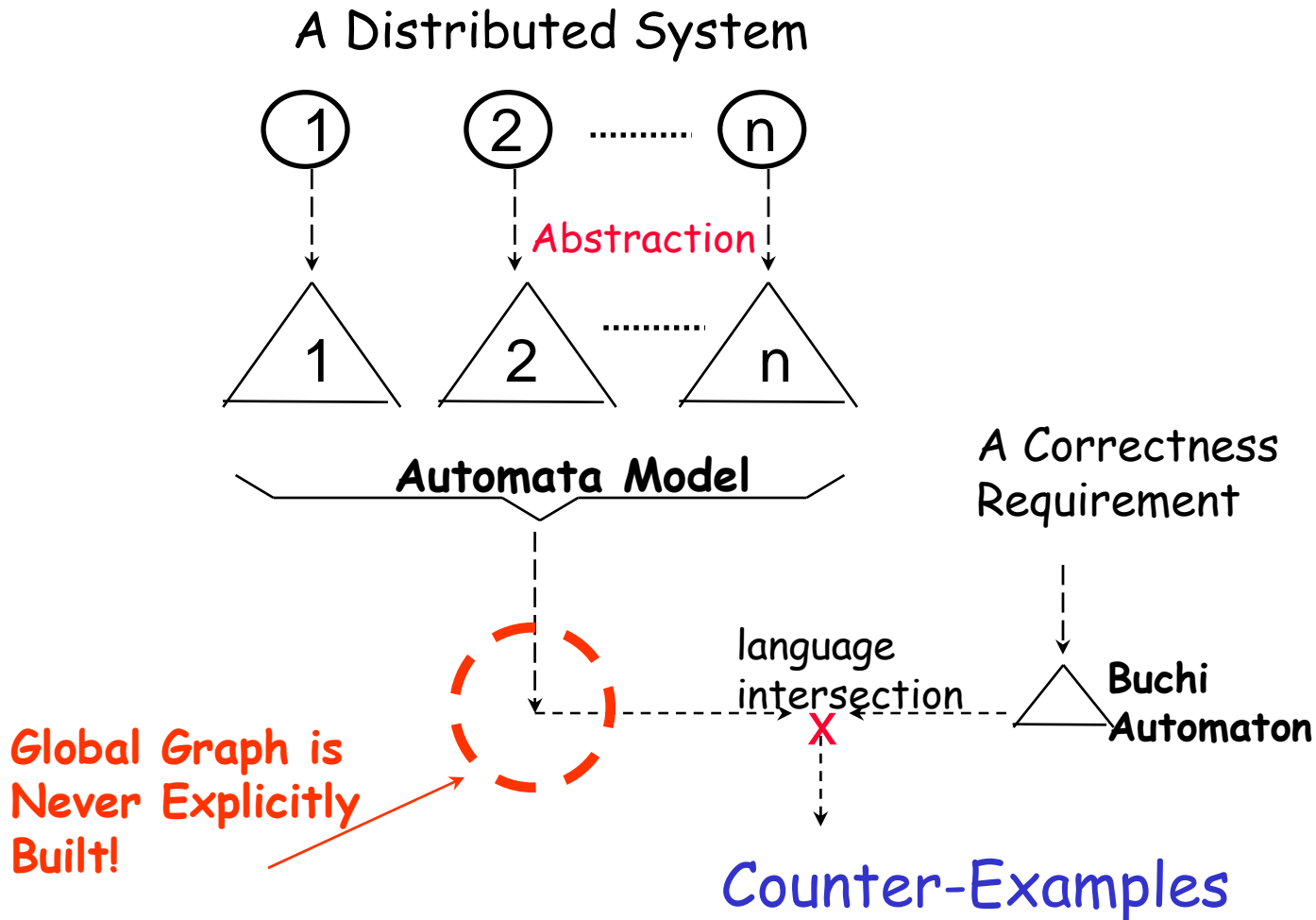
# algorithms and data structures

- basic issues
  - reachability
  - cycle detection
- **explicit** state enumeration
  - dfs
- **implicit** state enumeration
  - bfs
- data structures for storing states

# explicit state enumeration

- **problem:** is there a cycle containing a final state ?
- depth first search (dfs)
  - compute strongly connected components
- nested depth first search (used in Spin)
- state storage
  - compression techniques

# spin's on-the-fly verification procedure (language intersection)



# depth first search

## dfs algorithm:

```
store =  $\emptyset$ 
```

```
dfs(q)
```

```
{
```

```
  if  $q \in$  store
```

```
    return;
```

```
  else
```

```
    store = store  $\cup$  {q};
```

```
    for each successor s of q
```

```
      dfs(s);
```

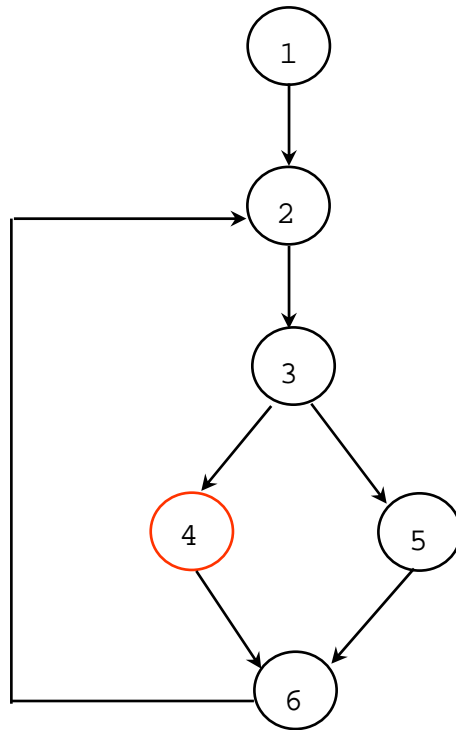
```
}
```

← Memory Bottleneck

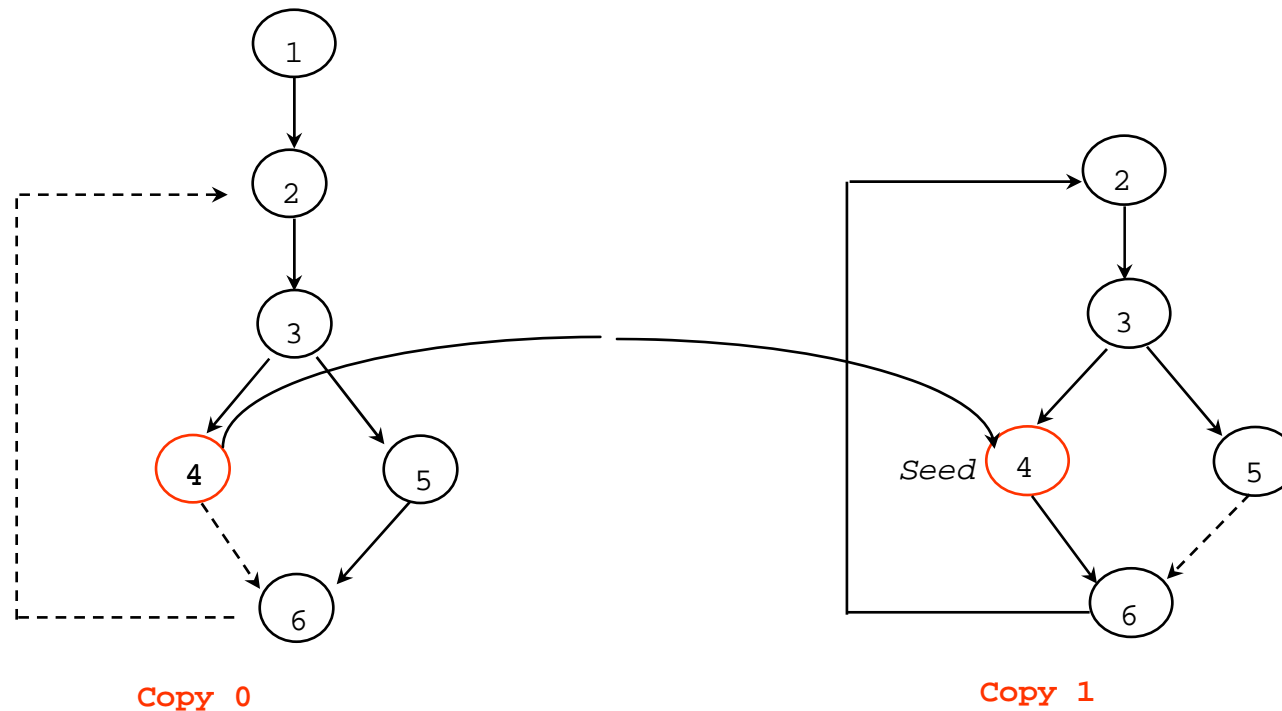
- recursively explore state graph
- store as little data as possible about previously visited states
- no need to store transitions

# Example

**Problem:** Is there a cycle through the final state ?



# Nested Depth-First Search



- Memory Overhead: **2 bits** per state
  - Tarjan's dfs requires **2 x 32 bits** per state
- Worst case time overhead: **2 x dfs**

# computational cost

Number of States: R

Size of each state: S

R x S dominates all costs

1) try to reduce R

- model reduction (abstraction)
- partial order reduction

2) try to reduce S

- store each state explicitly (**fast** - default)
- compress states (-DCOLLAPSE)
  - lossy compression (ie, using hash function)
- compute **minimized DFA** recognizer for the states (-DMA=N)

# implicit state enumeration

- boolean functions
  - canonical representation using minimized dfa (obdds)
- basic boolean operations
- symbolic representation of transition relation
- symbolic breadth first search (bfs)

# Boolean Functions

**Example:**

Boolean Variables:  $a, b, c$

$$f = ab + bc + ac$$

n-variable boolean function:

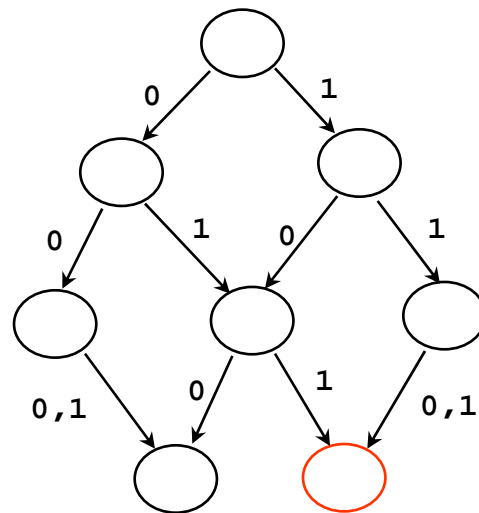
$$f : \{0,1\}^n \rightarrow \{0,1\}$$

**Truth Table:**

a	b	c	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

# Minimized DFA representation of Boolean Functions

$$f = ab + bc + ac$$



- Almost same as OBDDs

# Basic Boolean Operations

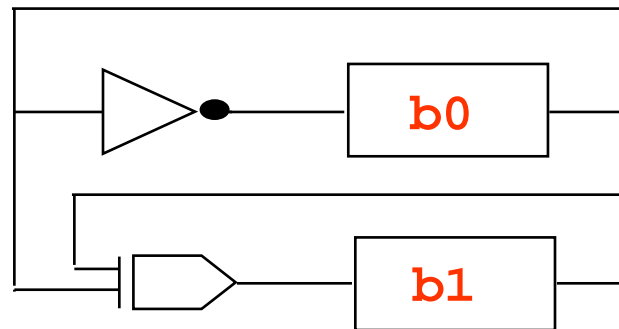
- All boolean operations can be performed **efficiently** using the minimized DFA representation
- Disjunction:  $h = f + g$
- Negation:  $h = \sim f$
- Existential quantification:
  - $h(y) = \exists x f(x,y) = f(0,y) + f(1,y)$
- Canonical form
  - equivalence of two boolean functions

# Symbolic Transition Relation

## Transition Relation:

$$R \subset Q \times Q$$

$(q,r) \in R$  iff there is an edge from  $q$  to  $r$



## symbolic transition relation:

$$(b0' = \sim b0) \wedge (b1' = b1 \oplus b0)$$

boolean function representation:

$$R : \{0,1\}^2 \times \{0,1\}^2 \rightarrow \{0,1\}$$

# symbolic breadth first search

symbolic transition relation:  $R(q, q')$

initial state:  $s_0(q)$

symbolic breadth first search:

$$\text{states}_0(q) = s_0(q)$$

$$s_1(q') = \exists q ( s_0(q) \wedge R(q, q') )$$

$$\text{states}_1(q) = \text{states}_0(q) + s_1(q)$$

$$s_2(q') = \exists q ( s_1(q) \wedge R(q, q') )$$


$$\text{states}_2(q) = \text{states}_1(q) + s_2(q)$$

⋮

stop when  $\text{states}_i(q) = \text{states}_{i-1}(q)$



# outline

- introduction (15 mins)
- theory and algorithms
  - system modeling and specifications (25 mins)
  - algorithms and data structures (25 mins)
- practice (25 minutes) 
  - the role of abstraction
  - model checking tools
  - some applications

# exponential effects in the state space

- **Example:**
  - k integers, each taking values from -N to N
  - number of states  $\sim (2N)^k$
- **Example:**
  - k buffers, s slots per buffer, m different items
  - number of states  $\sim m^{sk}$

# importance of abstraction

- model checkers are intended to be applied to models of systems, not directly to implementations
  - a model is a design abstraction
- find the **appropriate abstraction**
  - modeling the interfaces between modules without modeling the internal details of modules
- **without abstractions**, simple problems can become intractable
- and **with abstractions**, even hard problems can be solved if you know which parameters to tune

# importance of abstractions (cont.)

- if the problem is complex:
  - there is no free lunch: you have to find the right design abstraction
  - be suspicious of variables, counters, integer data items
  - think of how you would explain the principles of the design on a blackboard to a friend
  - model only the relevant aspects

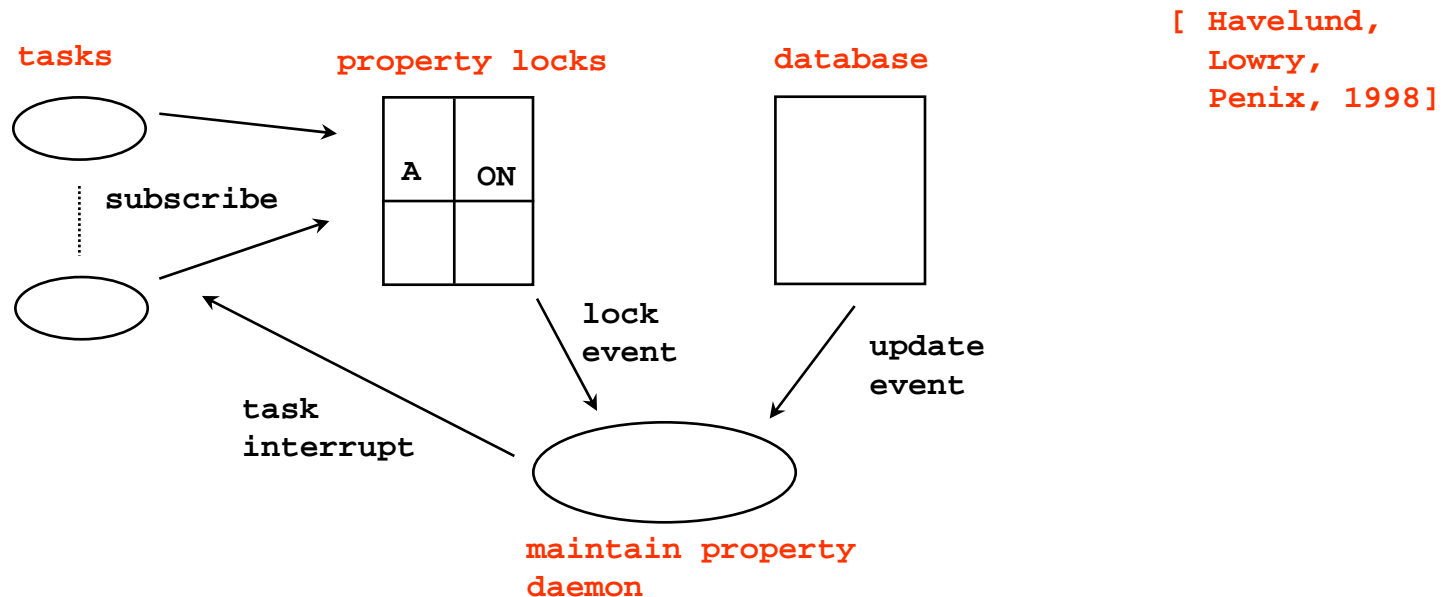
# tools

- **implicit state enumeration**
  - oriented towards hardware verification
  - smv (CTL), vis, cospan, ...
- **explicit state enumeration**
  - oriented towards software verification
  - spin (LTL), murphi, ...
- **commercial model checking tools**



# deep space 1 mission to mars

remote agent executive resource manager  
- verification of a new experimental  
distributed controller design using  
SPIN



# deep space 1 - results

- modeling effort - 2 people from NASA/Ames Research center about 6-8 weeks (354 lines in SPIN)
- 1 week to run and document verification (approx. 300K states, 10 seconds/run)
- 2 properties checked, both failed
- 4 serious errors found
- developer's reaction
  - "you've found a number of bugs that i am sure would not have been found otherwise. One of the bugs revealed a major design flaw. So I'd say you have had a substantial impact."

# processor verification

- pentium floating-point division bug cost Intel **\$475 million**
- plenty of **dollars** to do a lot of formal verification
- specialized techniques
  - ALU verification
  - pipelines etc.
- much commercial activity by EDA tool vendors and chip design companies

# intel looking to hire ...

although formal verification methods have been in widespread use at Intel for several years, **sequential formal verification (using model checking)** has seen its first wide deployment on the Merced project in the Santa Clara Processor Division

we have trained a large number of engineers in the techniques of formal specification and proof, made great strides in mastering the technology, improving the tools and developing methodologies of usage; but most importantly, **we have found numerous bugs, some of them very sneaky,** and established FV as a valuable contributor to the quality of logic design.

we are looking for a small number of exceptional candidates to own the **formal verification of blocks of Merced logic design.**

The work involves defining a comprehensive set of assertions which describe the behavior of the blocks, formulating them in a formal language, and proving these properties using our automated proof tools. Our **current focus is on model checking as the primary proof mechanism,** but we have access to and will make use of other tools as appropriate.

# summary

- real problem: find **bugs** in design
- model checking: build a model of the system, then the model checking tool will **automatically** check whether the specification is satisfied
- **abstractions** are key in building tractable models
- **theory** of model checking
- good **tool** support
- both software and hardware model checking are becoming **mainstream**, after having proven their value in academic and industrial use

## some references

- **Gerard Holzmann**, *Design and Validation of Computer Protocols*, Prentice-Hall, 1991.
  - **good introduction on spin and protocol verification**
- **Ken McMillan**, *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.
  - **good introduction to SMV, hardware and symbolic verification**
- **K.Havelund, M.Lowry, J.Penix**, *Formal Analysis of a Space Craft Controller using SPIN*, 1998.
  - **nice application of model checking**