# How to Verify a CSP Model?

**February 28, 2009**

# Previously

Given a process, a Labeled Transition System can be built by repeatedly applying the operational semantics.

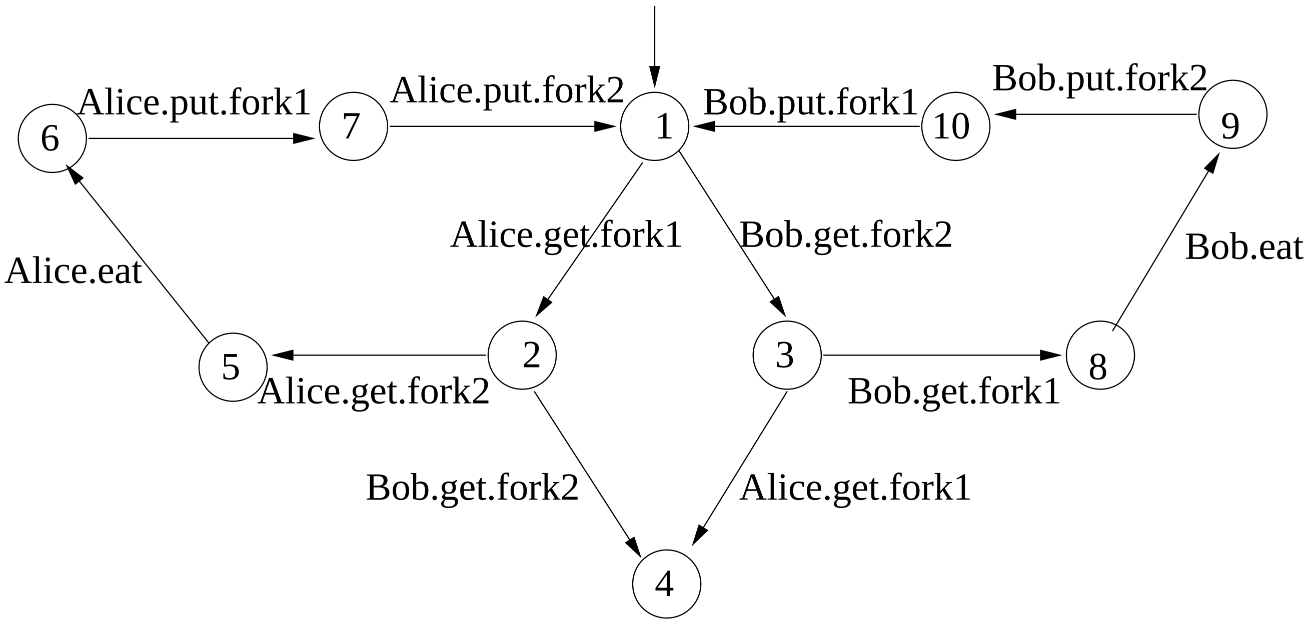
- Given,

$$
\begin{aligned}
Alice &= Alice.get.fork1 \rightarrow Alice.get.fork2 \rightarrow Alice.eat \\
&\rightarrow Alice.put.fork1 \rightarrow Alice.put.fork2 \rightarrow Alice \\
\\
Bob &= Bob.get.fork2 \rightarrow Bob.get.fork1 \rightarrow Bob.eat \\
&\rightarrow Bob.put.fork2 \rightarrow Bob.put.fork1 \rightarrow Bob \\
\\
Fork1 &= Alice.get.fork1 \rightarrow Alice.put.fork1 \rightarrow Fork1 \,\square \\
&\quad Bob.get.fork1 \rightarrow Bob.put.fork1 \rightarrow Fork1 \\
\\
Fork2 &= Alice.get.fork2 \rightarrow Alice.put.fork2 \rightarrow Fork2 \,\square \\
&\quad Bob.get.fork2 \rightarrow Bob.put.fork2 \rightarrow Fork2 \\
\\
College &= Alice \parallel Bob \parallel Fork1 \parallel Fork2
\end{aligned}
$$

# Previously (cont'ed)

Given a process, a Labeled Transition System can be built by repeatedly applying the operational semantics.

- We built,



# Outline

- What are the questions you can ask about a system?
  - Safety: *something bad never happens*
  - Liveness: *something good eventually happens*
  - Liveness under fairness: *what if the world is fair, can something good happen eventually?*
- Case study: multi-lift system
  - modeling,
  - verifying using PAT

# What is safety?

Safety $\approx$ *something bad never happens*

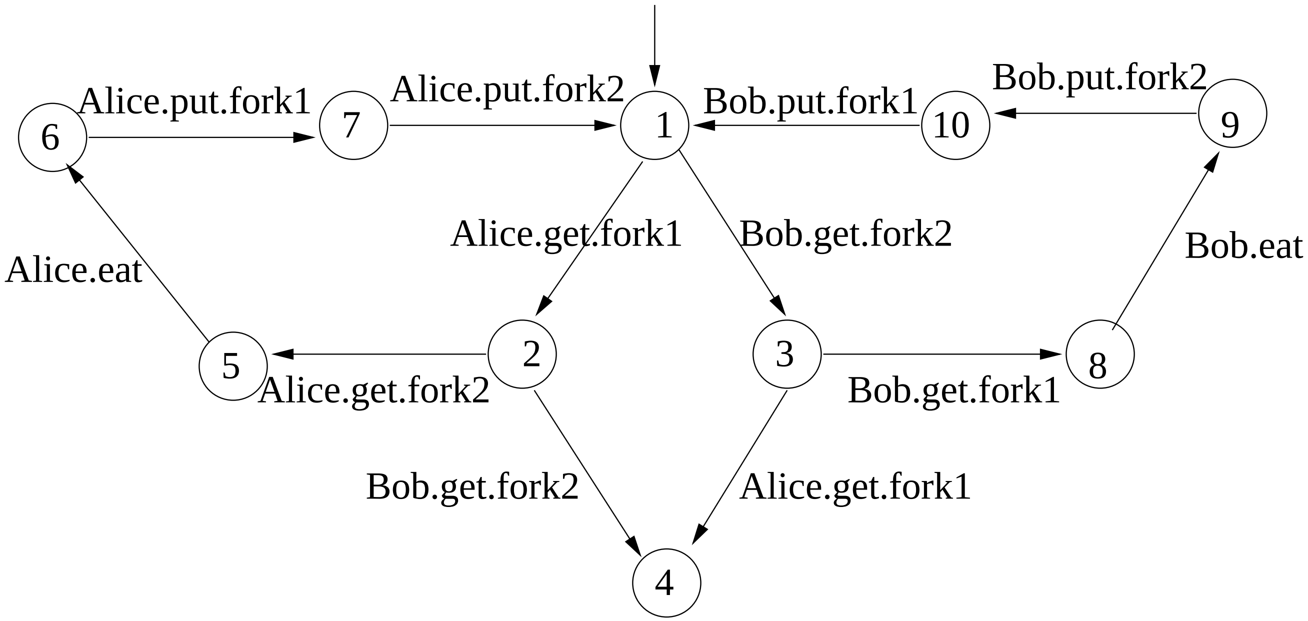
- deadlock-freeness, i.e., the system never deadlocks.
  - $\#assert\ College()\ deadlockfree;$
- invariant, e.g., the value of an array index must never be negative, the amount in a saving account must always be non-negative.
  - $\#assert\ Bank()\ \ |= \ \ []$ *cond* where [] reads 'always' and *cond* could be $Value >= Debit$.

# How to verify safety?

Verification of safety $\approx$ reachability analysis

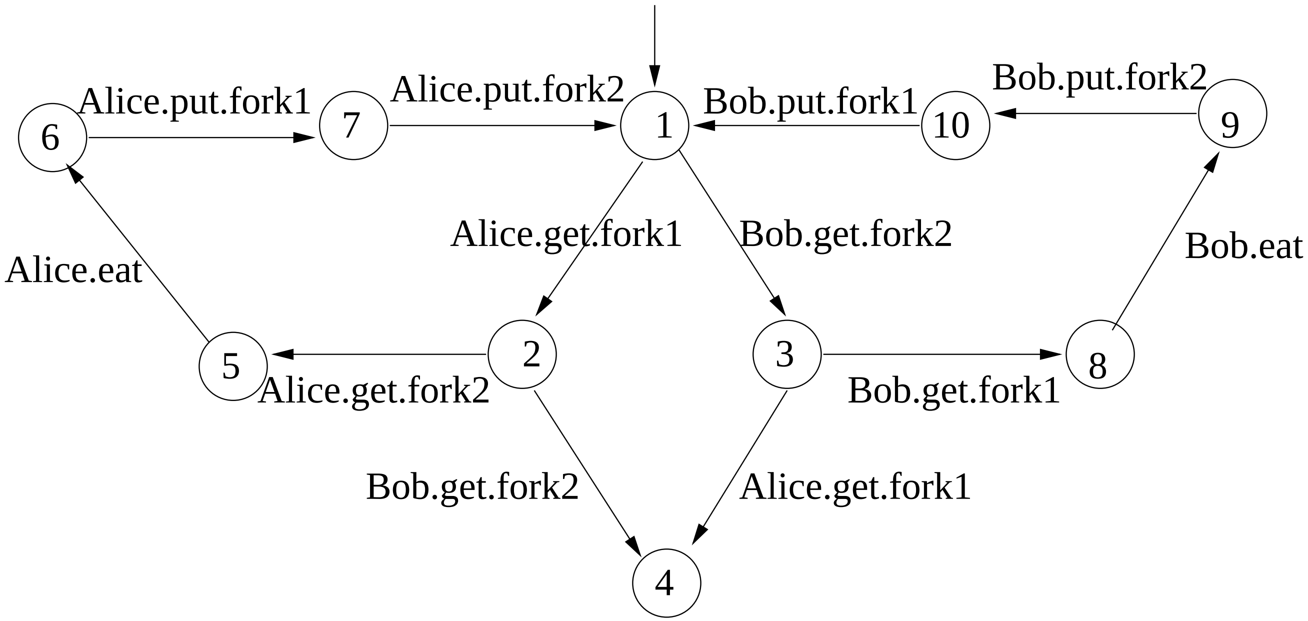
- A counterexample to a safety property is a *finite* execution which leads to a bad state.
- Searching through all reachable states for a bad one,
  - e.g., one which has no outgoing transition.
  - e.g., one that violates the invariant.
- Depth First Search (DFS) vs Breadth First Search (BFS)

# Verifying Safety: Example




Depth First Search: $1 \longrightarrow 2 \longrightarrow 5 \longrightarrow 6 \longrightarrow 7 \longrightarrow 1 \longrightarrow backtrack \longrightarrow 4 \longrightarrow FOUND!$

# Verifying Safety: Example (cont'ed)



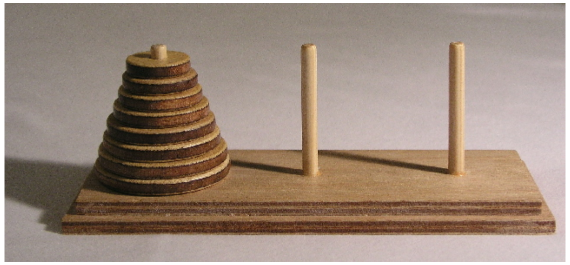

Breadth First Search: $1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 4 \rightarrow FOUND!$

## Safety Verification: Applications

Many properties can be formulated as a safety property and solved using reachability analysis.

- mutual exclusion: []!(more than one processes are accessing the criticl section)
- security: [](only the authorized user can access the information)
- program analysis: arrays are always bounded, pointers are always non-null, etc.

## Safety Verification: Applications (cont'ed)



$\#assert\ Hanoi()\ \ |=\ \ []!$(the disks are stacked in order on right rod)

$\#assert\ Cube()\ \ |=\ \ []!$(all stickers on each face are of the same color)

# What is Liveness?

Liveness $\approx$ *something good eventually happens*

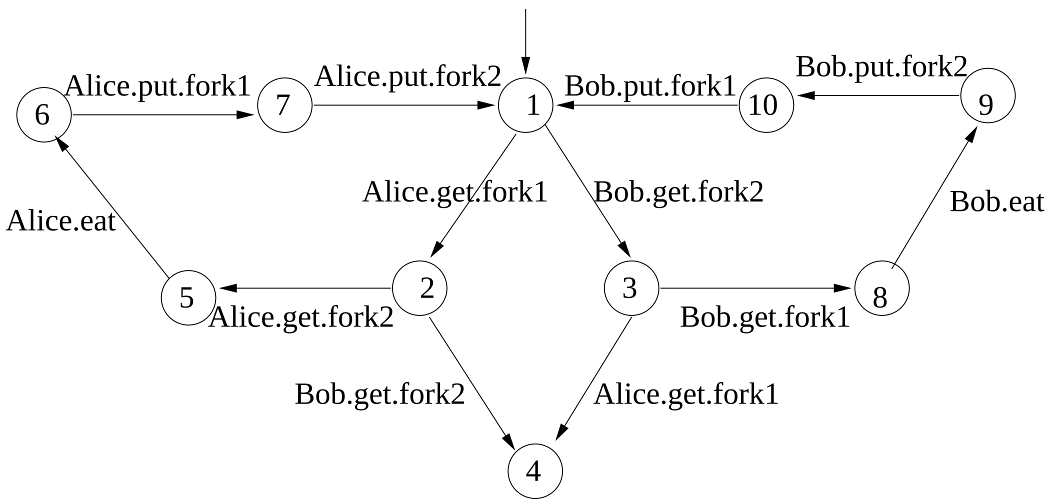
- a program is eventually terminating?
- a file writer is eventually closed?
- both Alice and Bob always eventually get to eat?

# How to verify liveness?

Verification of liveness $\approx$ loop searching

- A counterexample to a liveness property is an infinite system execution during which the 'good' thing never happens.
  - e.g., an infinite loop fails the property that the program is eventually terminating.
- Searching through the Labeled Transition System for a bad loop.
- Nested Depth First Search vs SCC-based Search

# Liveness Verification: Example



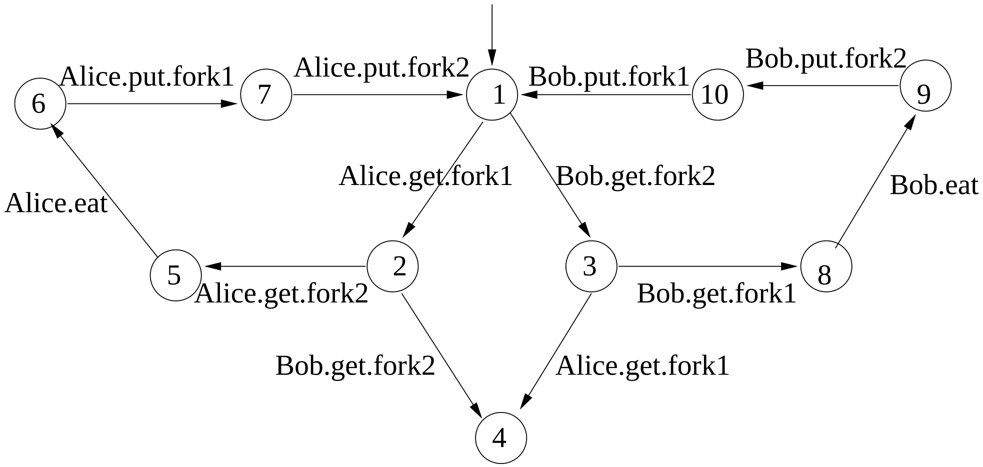

$$\#assert\ College()\ \ |=\ \ [\,]<>\ Alice.eat$$

- $\times$ $\langle Alice.get.fork1, Bob.get.fork2\rangle$
- $\times$ $\langle Bob.get.fork2 \rightarrow Bob.get.fork1 \rightarrow Bob.eat \rightarrow Bob.put.fork2 \rightarrow Bob.put.fork1\rangle^{\infty}$

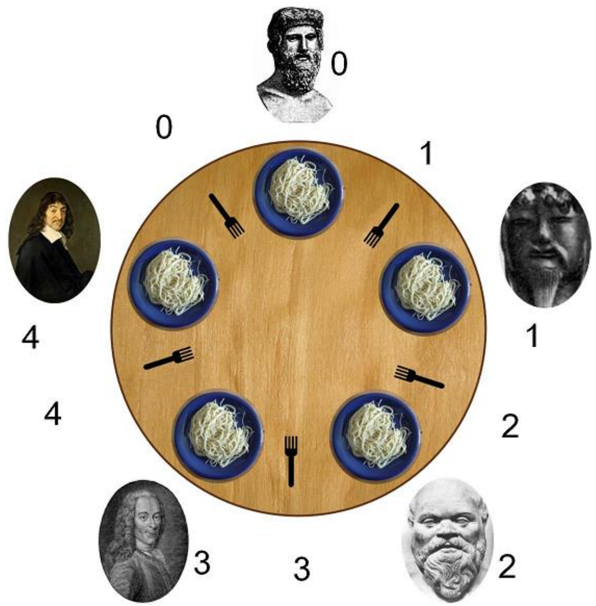
# What is Fairness?

Fairness ≈ *something is often possible, then it must eventually be performed*

- Fairness is important for verification of liveness.
- The default fairness assumption: *the system must eventually do something if possible.*



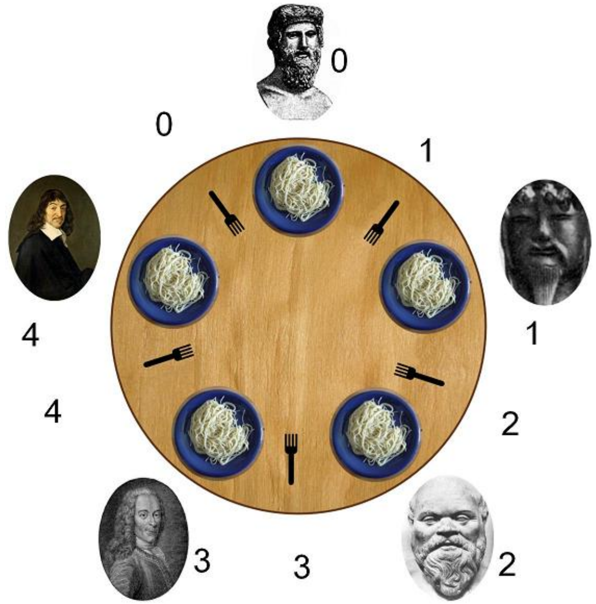
# Generalized Dining Philosophers




Event *get.i.j* (*put.i.j*) is the event of *i*-phil gets (puts down) the *j*-fork.

$$
\begin{aligned}
\mathit{Phil}(i) \;&= \mathit{get}.i.(i+1)\%N \rightarrow \mathit{get}.i.i \rightarrow \mathit{eat}.i \\
&\quad \rightarrow \mathit{put}.i.(i+1)\%N \rightarrow \mathit{put}.i.i \rightarrow \mathit{Phil}(i) \\
\mathit{Fork}(x) \;&= \mathit{get}.x.x \rightarrow \mathit{put}.x.x \rightarrow \mathit{Fork}(x) \;\Box \\
&\quad \mathit{get}.(x-1)\%N.x \rightarrow \mathit{put}.(x-1)\%N.x \rightarrow \mathit{Fork}(x) \\
\mathit{College}() &= \,||\; x : \{0..N-1\} \bullet (\mathit{Phil}(x) \;||\; \mathit{Fork}(x));
\end{aligned}
$$

# Generalized Dining Philosophers (cont'ed)




$\#assert\ College() \ \ |= \ \ [] <> eat.0$

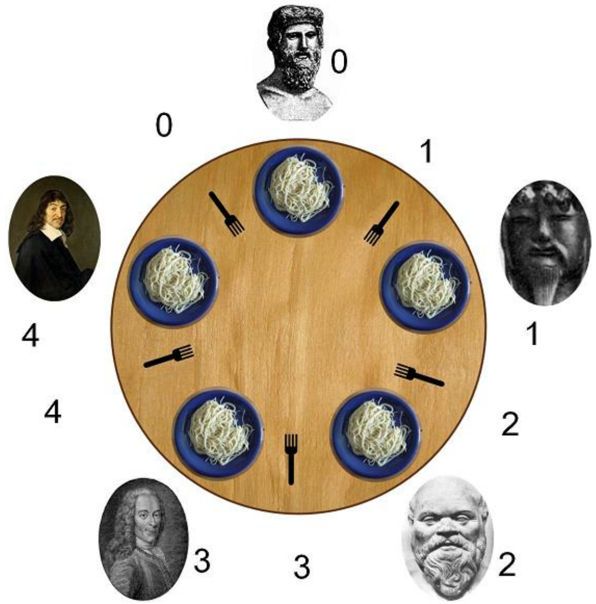
$\langle get.0.1, get.1.2, get.2.3, get.3.4, get.4.0\rangle$ – deadlock!
$\langle get.2.3, get.2.2, eat.2, put.2.3, put.2.2\rangle^\infty$ – lack of weak fairness
$\langle get.1.2, get.1.1, eat.1, put.1.2, put.1.1\rangle^\infty$ – lack of strong fairness

# How to Verify Liveness under Fairness?

Verification of liveness under fairness $\approx$ *fair* loop searching

- A counterexample to a liveness property under fairness is an infinite *fair* system execution during which the 'good' thing never happens.
  - e.g., under weak fairness, a loop is fair if and only if there does NOT exist a transition which is always possible but never performed.
- Searching through the Labeled Transition System for a *fair* loop which is bad.
- Nested Depth First Search vs SCC-based Search

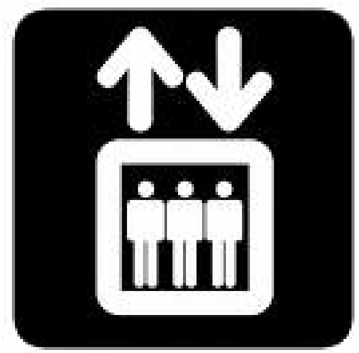
# Liveness Verification under Fairness: Example




Assume weak fairness, $\#assert\ College()\ \ |=\ \ []<>eat.0$

| | |
|---|---|
| $\langle get.0.1, get.1.2, get.2.3, get.3.4, get.4.0\rangle$ | – deadlock! |
| $\langle get.1.2, get.1.1, eat.1, put.1.2, put.1.1\rangle^{\infty}$ | – lack of strong fairness |
| $\langle get.2.3, get.2.2, eat.2, put.2.3, put.2.2\rangle^{\infty}$ | – is NOT a counter example! |

# Case Study: Multi-lift System



# Extending CSP

- The original CSP has no shared variables, arrays, etc!
- CSP can be extended with programming language features for data aspects and data operations.
- The operational semantics must be tuned, e.g.,

$$\frac{(V, P) \xrightarrow{x} (V', P')}{(V, P \Box Q) \xrightarrow{x} (V', P')}\,[\,ch1\,] \qquad \frac{(V, Q) \xrightarrow{x} (V', Q')}{(V, P \Box Q) \xrightarrow{x} (V', Q')}\,[\,ch2\,]$$

# Multi-lift System: the Data Variables

Variables/arrays are necessary to capture the status of the lift.

| | |
|---|---|
| **#define** *NoOfFloor* 3; | – number of floors |
| **#define** *NoOfLift* 2; | – number of lifts |
| **var** $extUpReq[NoOfFloor]$; | – external requests for going up |
| **var** $extDownReq[NoOfFloor]$; | – external requests for going down |
| **var** $intRequests[NoOfFloor * NoOfLift]$; | – internal requests |
| **var** $doorOpen[NoOfLift]$; | – door status |

# Data Operations

A system may have data operations which updates the variables. When the door of the $i$th-lift is open at *level*-floor, the following is invoked to clear the requests.

$intRequests[level + i * NoOfFloor] = 0;$ – clear internal requests
**if** $(dir > 0)\{$
    $extUpReq[level] = 0;$ – clear external requests
$\}$
**else** $\{$
    $extDownReq[level] = 0;$
$\}$

# Data Operations (cont'd)

When the $i$th-lift is residing at *level*-floor is deciding whether to continue traveling on the same direction or to change direction,

$$
\begin{array}{l}
index = level + dir;\ result[i] = 0; \\
\textbf{while}\ (index >= 0\ \&\&\ index < NoOfFloor)\ \{ \\
\quad\quad \textbf{if}\ (extUpReq[index]! = 0\ \&\& extDownReq[index]! = 0\ \&\& \\
\quad\quad\quad\quad intRequests[index + i * NoOfFloor]! = 0)\{ \\
\quad\quad\quad\quad result[i] = 1; \\
\quad\quad \} \\
\quad\quad \textbf{else}\ \{ \\
\quad\quad\quad\quad index = index + dir; \\
\quad\quad \} \\
\}
\end{array}
$$

# Modeling the Lift

$Lift(i, level, dir) =$

**if** $((dir > 0 \ \&\& \ extUpReq[level] == 1) \ || \ (dir < 0 \ \&\& \ extDownReq[level] == 1) \ ||$

$\quad intRequests[level + i * NoOfFloor] == dir) \ \{$

$\quad opendoor.i\{doorOpen[i] = level;$ *data operation shown before*$\} \rightarrow$

$\quad closedoor.i\{doorOpen[i] = -1\} \rightarrow Lift(i, level, dir)$

$\}$ **else** $\{$

$\quad checkIfToMove.i.level\{$*data operation shown before*$\} \rightarrow$

$\quad$ **if** $(result[i] == 1)\{moving.i.dir \rightarrow$

$\quad\quad$ **if** $(level + dir == 0 \ || \ level + dir == NoOfFloors - 1)\{$

$\quad\quad\quad Lift(i, level + dir, -1 * dir)$

$\quad\quad \}$

$\quad\quad$ **else** $\{Lift(i, level + dir, dir)\}$

$\quad \}$ **else** $\{$

$\quad\quad$ **if** $((level == 0 \ \&\& \ dir == 1) \ || \ (level == NoOfFloors - 1 \ \&\& \ dir == -1))\{$

$\quad\quad\quad Lift(i, level, dir)$

$\quad\quad \}$

$\quad\quad$ **else** $\{changedir.i.level \rightarrow Lift(i, level, -1 * dir)\}\}\};$

# Modeling the Users

$$
\begin{array}{l}
aUser() = [] \ pos : \{0..NoOfFloor - 1\} @ (ExternalPush(pos); \ Waiting(pos)); \\
ExternalPush(pos) = \ \textbf{case} \ \{ \\
\quad pos == 0 : \ pushup.pos\{extUpReq[pos] = 1\} \rightarrow Skip \\
\quad pos == NoOfFloor - 1 \ : \ pushdown.pos\{extDownReq[pos] = 1\} \rightarrow Skip \\
\quad \textbf{default} \ : \ pushup.pos\{extUpReq[pos] = 1\} \rightarrow Skip \ [] \\
\qquad\qquad pushdown.pos\{extDownReq[pos] = 1\} \rightarrow Skip \\
\quad \}; \\
Waiting(pos) = [] \ i : \{0..NoOfLift - 1\} @ ([doorOpen[i] == pos] enter.i \rightarrow \\
\quad [] x : \{0..NoOfFloor - 1\} @ (push.x\{intRequests[x + i * NoOfFloor] = 1\} \rightarrow \\
\quad [doorOpen[i] == x] exit.i.x \rightarrow User())); \\
Users() = ||| \ x : \{0..2\} @ aUser();
\end{array}
$$

# Modeling and Questioning the System

$LiftSystem() = Users() \; ||| \; (||| \; x : \{0..NoOfLift - 1\} @ Lift(x, 0, 1));$
$\#\textbf{assert} \; LiftSystem() \; \textbf{deadlockfree};$
$\#\textbf{define} \; pr1 \; extUpReq[0] > 0;$
$\#\textbf{define} \; pr2 \; extUpReq[0] == 0;$
$\#\textbf{assert} \; LiftSystem() \; \models \; \Box(pr1 \Rightarrow \Diamond pr2) \; \&\& \; \Box\Diamond moving.0$
...

Tool Demonstration