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,

\[
Alice = Alice.\text{get.fork1} \rightarrow Alice.\text{get.fork2} \rightarrow Alice.\text{eat} \\
\rightarrow Alice.\text{put.fork1} \rightarrow Alice.\text{put.fork2} \rightarrow Alice
\]

\[
Bob = Bob.\text{get.fork2} \rightarrow Bob.\text{get.fork1} \rightarrow Bob.\text{eat} \\
\rightarrow Bob.\text{put.fork2} \rightarrow Bob.\text{put.fork1} \rightarrow Bob
\]

\[
\text{Fork1} = Alice.\text{get.fork1} \rightarrow Alice.\text{put.fork1} \rightarrow \text{Fork1} \Box \\
Bob.\text{get.fork1} \rightarrow Bob.\text{put.fork1} \rightarrow \text{Fork1}
\]

\[
\text{Fork2} = Alice.\text{get.fork2} \rightarrow Alice.\text{put.fork2} \rightarrow \text{Fork2} \Box \\
Bob.\text{get.fork2} \rightarrow Bob.\text{put.fork2} \rightarrow \text{Fork2}
\]

\[
\text{College} = Alice \parallel Bob \parallel \text{Fork1} \parallel \text{Fork2}
\]
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: $1 \rightarrow 2 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 1 \rightarrow backtrack \rightarrow 4 \rightarrow FOUND$!
Verifying Safety: Example (cont’ed)

Breadth First Search: 1 → 2 → 3 → 5 → 4 → FOUND!
Safety Verification: Applications

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

- mutual exclusion: \(!(! \text{more than one processes are accessing the criticl section})\)
- security: \(!(! \text{only the authorized user can access the information})\)
- 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

× ⟨Alice.get.fork1, Bob.get.fork2⟩

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

$$Phil(i) = get.i.(i + 1)\%N \rightarrow get.i.i \rightarrow eat.i$$
$$\rightarrow put.i.(i + 1)\%N \rightarrow put.i.i \rightarrow Phil(i)$$

$$Fork(x) = get.x.x \rightarrow put.x.x \rightarrow Fork(x) \Box$$
$$get.(x - 1)\%N.x \rightarrow put.(x - 1)\%N.x \rightarrow Fork(x)$$

$$College() = \parallel x: \{0..N - 1\} \bullet (Phil(x) \parallel Fork(x))$$
Generalized Dining Philosophers (cont’ed)

#assert College() |— [] <> eat.0

⟨get.0.1, get.1.2, get.2.3, get.3.4, get.4.0⟩  
⟨get.2.3, get.2.2, eat.2, put.2.3, put.2.2⟩^∞  
⟨get.1.2, get.1.1, eat.1, put.1.2, put.1.1⟩^∞  

– deadlock!  
– lack of weak fairness  
– lack of strong fairness
How to Verify Liveness under Fairness?

Verification of liveness under fairness ≈ 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() \models [] <> eat.0

\langle get.0.1, get.1.2, get.2.3, get.3.4, get.4.0 \rangle
\langle get.1.2, get.1.1, eat.1, put.1.2, put.1.1 \rangle^\infty
\langle get.2.3, get.2.2, eat.2, put.2.3, put.2.2 \rangle^\infty

- deadlock!
- lack of strong fairness
- 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.,

\[
\begin{align*}
(V, P) & \xrightarrow{x} (V', P') \quad [\text{ch1}] \\
(V, P \parallel Q) & \xrightarrow{x} (V', P') \\
(V, Q) & \xrightarrow{x} (V', Q') \quad [\text{ch2}] \\
(V, P \parallel Q) & \xrightarrow{x} (V', Q')
\end{align*}
\]
Multi-lift System: the Data Variables

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

```c
#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.

\[
\begin{align*}
\text{intRequests}[level + i \times \text{NoOfFloor}] &= 0; & \text{– clear internal requests} \\
\text{if (dir > 0)} & \{ \text{extUpReq}[level] = 0; \} & \text{– clear external requests} \\
\text{else} & \{ \text{extDownReq}[level] = 0; \} \\
\end{align*}
\]
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,

$$\text{index} = \text{level} + \text{dir}; \text{result}[i] = 0;$$

**while** $(\text{index} \geq 0 \&\& \text{index} < \text{NoOfFloor})$ {

**if** $(\text{extUpReq}[\text{index}] != 0 \&\& \text{extDownReq}[\text{index}] != 0 \&\&$

$$\text{intRequests[\text{index} + i * \text{NoOfFloor}]} != 0)$

$$\text{result}[i] = 1;$$

} **else** {

$$\text{index} = \text{index} + \text{dir};$$

}

}
Modeling the Lift

$Lift(i, level, dir) =$

if ($(dir > 0 &\& extUpReq[level] == 1) \mid (dir < 0 &\& extDownReq[level] == 1) \mid$
intRequests[level + i * NoOfFloor] == dir) {
  opendoor.i{doorOpen[i] = level; *data operation shown before*} →
closedoor.i{doorOpen[i] = −1} → Lift(i, level, dir)
} else {
  checkIfToMove.i.level{*data operation shown before*} →
  if (result[i] == 1){moving.i.dir →
    if (level + dir == 0 || level + dir == NoOfFloors − 1) {
      Lift(i, level + dir, −1 * dir)
    }
  else {Lift(i, level + dir, dir)}
} else {
  if ($(level == 0 &\& dir == 1) \mid (level == NoOfFloors − 1 &\& dir == −1))$
    Lift(i, level, dir)
} else {changedir.i.level → Lift(i, level, −1 * dir)}
};
Modeling the Users

\[
aUser() = \emptyset\; pos : \{0..NoOfFloor - 1\} @ (ExternalPush(pos); \; Waiting(pos));
\]

\[
ExternalPush(pos) = \text{case}\ \begin{cases} 
    pos == 0 & : \; pushup.pos\{\text{extUpReq[pos] = 1}\} \rightarrow Skip \\
    pos == NoOfFloor - 1 & : \; pushdown.pos\{\text{extDownReq[pos] = 1}\} \rightarrow Skip \\
    \text{default} & : \; pushup.pos\{\text{extUpReq[pos] = 1}\} \rightarrow Skip \; \emptyset \\
    \; pushdown.pos\{\text{extDownReq[pos] = 1}\} \rightarrow Skip \\
\end{cases}
\]

\[
Waiting(pos) = \emptyset\; i : \{0..\text{NoOfLift} - 1\} @ ([\text{doorOpen}[i] == pos] \; \text{enter}.i \rightarrow \\
\emptyset\; x : \{0..\text{NoOfFloor} - 1\} @ (\text{push}.x\{\text{intRequests}[x + i * \text{NoOfFloor}] = 1\} \rightarrow \\
[\text{doorOpen}[i] == x] \; \text{exit}.i.x \rightarrow User());
\]

\[
Users() = \|||\; x : \{0..2\} @ aUser();
\]
Modeling and Questioning the System

\[
LiftSystem() = Users() \ || \ (|| x : \{0..NoOfLift - 1\} \@Lift(x, 0, 1));
\]

`assert LiftSystem() deadlockfree;`

`define pr1 extUpReq[0] > 0;`

`define pr2 extUpReq[0] == 0;`

`assert LiftSystem() |= □(pr1 ⇒ ◇pr2) & & □□moving.0`

...