Bounded Model Checking

handling infinite non-deterministic behavior

Classical Logic Operations

and $(a \wedge b)$

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

or $(a \vee b)$

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

limp $(a \leftarrow b)$

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

rimp $(a \rightarrow b)$

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

nand $(a \mathbin{\overline{\wedge}} b)$

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

nor $(a \mathbin{\overline{\vee}} b)$

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

nlimp $(a \not\leftarrow b)$

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

nrimp $(a \not\rightarrow b)$

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

xor $(a \oplus b)$

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

equiv $(a \leftrightarrow b)$

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

ite $(ite(a, b, c))$

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

neg $(\overline{a})$

a	f
0	1
1	0

f is the value of the expression in parens

What is Bounded Model Checking?

Given a system that is modeled as a Finite State Machine (FSM), and specifications of what that system is supposed to do, model checking is the task of making sure that all specifications are satisfied by the model. This applies to hardware as well as software. The specifications state functional, security, and safety requirements (avoidance of bad states that can be exploited by an attacker or that result in a crash). The problem of model checking is solved used some form of logic. In most cases some form of temporal logic is best suited for this role because operators in temporal logic are designed to handle state queries about events indefinitely far into the future.

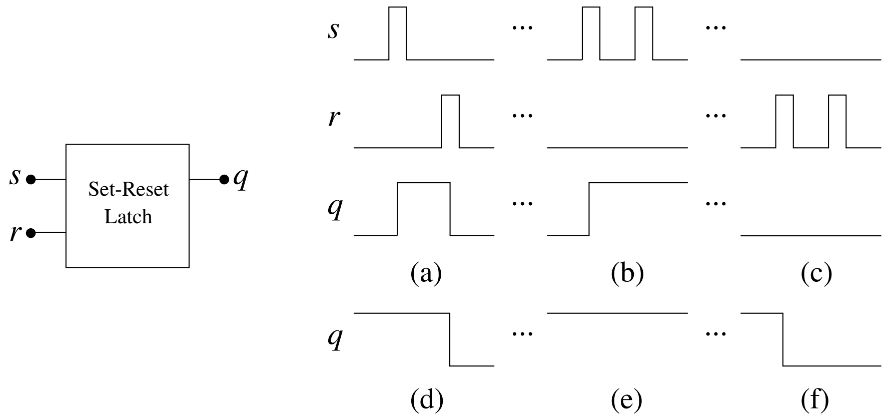
Over the years Boolean logic has risen as a great tool for solving hard but significant real-world problems including many instances of NP-complete problems: this is due to the efficiency of SAT and SMT solvers. But Boolean formulas cannot be allowed to grow arbitrarily to match what temporal logic can do. However, a FSM can be "unrolled" for a finite number of states and properties checked in that limited space. That is known as Bounded Model Checking (BMC). BMC, with the success of SAT and SMT solver progression over the years, has become and important and effective part of model checking; increasing confidence in system robustness.

Functional Verification - Temporal Logic Operators

Op. name	$(\mathbf{S}, \mathbf{s_i}) \models$	**if and only if**
	p (a variable)	$s_i(p) = 1.$
not	$\overline{\psi_1}$	$(S, s_i) \not\models \psi_1$
and	$\psi_1 \wedge \psi_2$	$(S, s_i) \models \psi_1$ and $(S, s_i) \models \psi_2$
or	$\psi_1 \vee \psi_2$	$(S, s_i) \models \psi_1$ or $(S, s_i) \models \psi_2$
henceforth	$\Box\, \psi_1$	$(S, s_j) \models \psi_1$ for all states $s_j, j \geq i.$
eventually	$\diamond\, \psi_1$	$(S, s_j) \models \psi_1$ for some state $s_j, j \geq i.$
next	$\circ\, \psi_1$	$(S, s_{i+1}) \models \psi_1.$
until	$\psi_1 \mathcal{U} \psi_2$	For some $j \geq i$, $(S, s_i), (S, s_{i+1}), \ldots, (S, s_{j-1}) \models \psi_1,$ and $(S, s_j) \models \psi_2.$

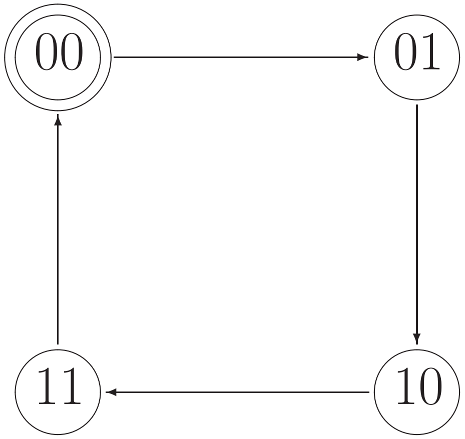
- $S = \{s_0, s_1, s_2, ...\}$ is a sequence of legal states and transitions.
- If ψ evaluates to 1 for state $s_i \in S$ we say $(S, s_i) \models \psi$.
- We say $S \models \psi$ if and only if $(S, s_0) \models \psi$.
- Two LTTL formulas ψ_1 and ψ_2 are *equivalent* if, for all sequences S, $S \models \psi_1$ if and only if $S \models \psi_2$.

Functional Verification - Set/Reset Latch



Expressions	Comments
$\Box\,\overline{(s \wedge r)}$	No two inputs have value 1 simultaneously.
$\Box\,((s \wedge \overline{q}) \rightarrow ((s\,\mathcal{U}\,q) \vee \Box\, s))$	Input s cannot change if s is 1 and q is 0.
$\Box\,((r \wedge q) \rightarrow ((r\,\mathcal{U}\,\overline{q}) \vee \Box\, r))$	Input r cannot change if r is 1 and q is 1.
$\Box\,(s \rightarrow \Diamond\, q)$	If s is 1, q will eventually be 1.
$\Box\,(r \rightarrow \Diamond\, \overline{q})$	If r is 1, q will eventually be 0.
$\Box\,((\overline{q} \rightarrow ((\overline{q}\,\mathcal{U}\,s) \vee \Box\,\overline{q})))$	Output q rises to 1 only if s becomes 1.
$\Box\,((q \rightarrow ((q\,\mathcal{U}\,r) \vee \Box\, q)))$	Output q drops to 0 only if r becomes 1.

Bounded Model Checking - Simple Counter Example



Variables: at time i,

v_1^i = value of bit 1,

v_2^i = value of bit 2

Property to prove:

Does the 2-bit counter above reach state 11 in exactly 3 time steps?

BMC, Counter: The Formula

Force the property to hold:

$$\neg(v_1^0 \wedge v_2^0) \wedge \neg(v_1^1 \wedge v_2^1) \wedge \neg(v_1^2 \wedge v_2^2) \wedge (v_1^3 \wedge v_2^3)$$

BMC, Counter: The Formula

Force the property to hold:

$$\neg(v_1^0 \wedge v_2^0) \wedge \neg(v_1^1 \wedge v_2^1) \wedge \neg(v_1^2 \wedge v_2^2) \wedge (v_1^3 \wedge v_2^3)$$

Express the starting state:

$$\neg v_1^0 \wedge \neg v_2^0$$

BMC, Counter: The Formula

Force the property to hold:

$$\neg(v_1^0 \wedge v_2^0) \wedge \neg(v_1^1 \wedge v_2^1) \wedge \neg(v_1^2 \wedge v_2^2) \wedge (v_1^3 \wedge v_2^3)$$

Express the starting state:

$$\neg v_1^0 \wedge \neg v_2^0$$

Force legal transitions (repeat transition relation):

$$\begin{gathered}(v_2^1 \leftrightarrow \neg v_2^0) \;\wedge\; (v_1^1 \leftrightarrow v_1^0 \oplus v_2^0) \;\wedge\; (v_2^2 \leftrightarrow \neg v_2^1) \wedge \\ (v_1^2 \leftrightarrow v_1^1 \oplus v_2^1) \;\wedge\; (v_2^3 \leftrightarrow \neg v_2^2) \;\wedge\; (v_1^3 \leftrightarrow v_1^2 \oplus v_2^2)\end{gathered}$$

BMC, Counter: The Formula

Force the property to hold:

$$\neg(v_1^0 \wedge v_2^0) \wedge \neg(v_1^1 \wedge v_2^1) \wedge \neg(v_1^2 \wedge v_2^2) \wedge (v_1^3 \wedge v_2^3)$$

Express the starting state:

$$\neg v_1^0 \wedge \neg v_2^0$$

Force legal transitions (repeat transition relation):

$$\begin{gathered}(v_2^1 \leftrightarrow \neg v_2^0) \;\wedge\; (v_1^1 \leftrightarrow v_1^0 \oplus v_2^0) \;\wedge\; (v_2^2 \leftrightarrow \neg v_2^1) \wedge \\ (v_1^2 \leftrightarrow v_1^1 \oplus v_2^1) \;\wedge\; (v_2^3 \leftrightarrow \neg v_2^2) \;\wedge\; (v_1^3 \leftrightarrow v_1^2 \oplus v_2^2)\end{gathered}$$

Satisfied *only* by:

$$v_1^0 = 0, v_2^0 = 0, v_1^1 = 0, v_2^1 = 1, v_1^2 = 1, v_2^2 = 0, v_1^3 = 1, v_2^3 = 1$$

Hence the property holds!

BMC, Counter: The Formula

Force the property to hold:

$$\neg(v_1^0 \wedge v_2^0) \wedge \neg(v_1^1 \wedge v_2^1) \wedge \neg(v_1^2 \wedge v_2^2) \wedge (v_1^3 \wedge v_2^3)$$

Express the starting state:

$$\neg v_1^0 \wedge \neg v_2^0$$

Force legal transitions (repeat transition relation):

$$(v_2^1 \leftrightarrow \neg v_2^0) \;\wedge\; (v_1^1 \leftrightarrow v_1^0 \oplus v_2^0) \;\wedge\; (v_2^2 \leftrightarrow \neg v_2^1) \;\wedge$$
$$(v_1^2 \leftrightarrow v_1^1 \oplus v_2^1) \;\wedge\; (v_2^3 \leftrightarrow \neg v_2^2) \;\wedge\; (v_1^3 \leftrightarrow v_1^2 \oplus v_2^2)$$

Satisfied *only* by:

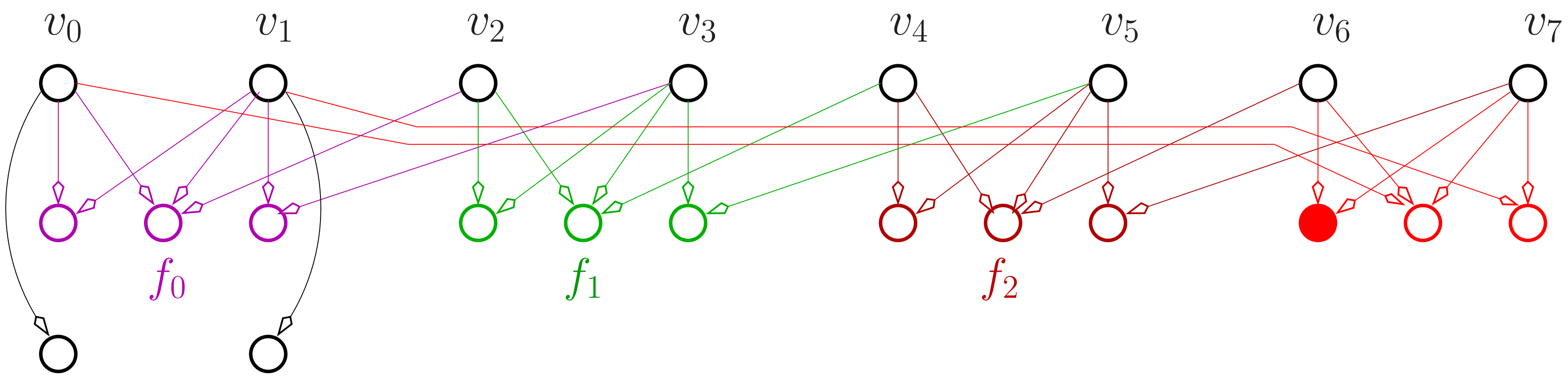
$$v_1^0 = 0, v_2^0 = 0, v_1^1 = 0, v_2^1 = 1, v_1^2 = 1, v_2^2 = 0, v_1^3 = 1, v_2^3 = 1$$

Hence the property holds!

We could have asked does the counter ever enter state 11?
No need to unroll in that case

BMC, Counter: The Formula

Three repetitions of a function

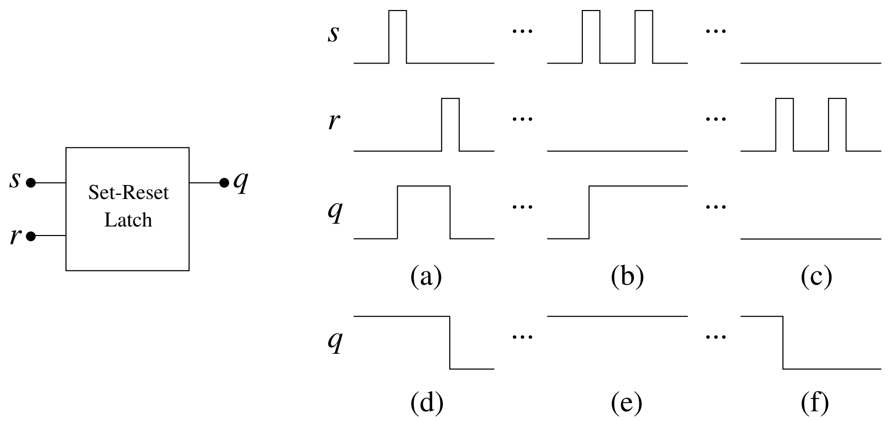


For $i \in \{0, 1, 2\}$:

$$f_i = \overline{(v_{2i} \wedge v_{2i+1})} \wedge (v_{2i+1} \equiv v_{2i+3}) \wedge (v_{2i+2} \equiv v_{2i} \oplus v_{2i+1})$$

$$f_3 = (v_6 \wedge v_7) \wedge (v_7 \equiv v_1) \wedge (v_0 \equiv v_6 \oplus v_7)$$

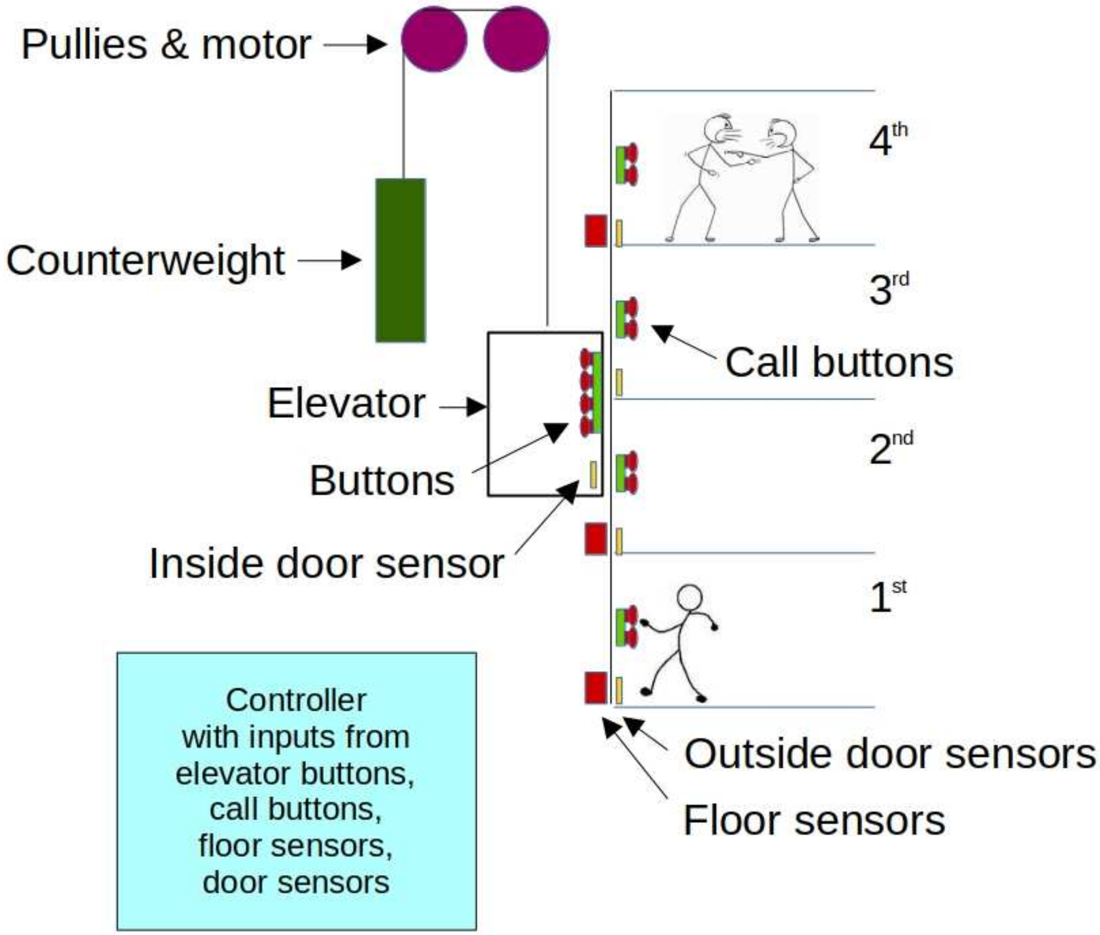
Temporal operators and Boolean expressions - Set/Reset Latch



Temporal	Boolean
$\Box\,\overline{(s \wedge r)}$	$\forall_i\ (\neg s_i \vee \neg r_i).$
$\Box\,((s \wedge \overline{q}) \rightarrow ((s\ \mathcal{U}\ q) \vee \Box\, s))$	$\forall_i\ (\neg s_i \vee q_i \vee q_{i+1} \vee s_{i+1}).$
$\Box\,((r \wedge q) \rightarrow ((r\ \mathcal{U}\ \overline{q}) \vee \Box\, r))$	$\forall i\ (r_i \vee \neg q_i \vee \neg q_{i+1} \vee r_{i+1}).$
$\Box\,(s \rightarrow \Diamond\, q)$	$\forall_i\ \neg(s_i) \vee \exists_{j>i}\ q_j.$
$\Box\,(r \rightarrow \Diamond\, \overline{q})$	$\forall_i\ \neg(r_i) \vee \exists_{j>i}\ \neg(q_j).$
$\Box\,((\overline{q} \rightarrow ((\overline{q}\ \mathcal{U}\ s) \vee \Box\, \overline{q})))$	$\forall_i\ (q_i \vee s_{i+1} \vee \neg q_{i+1}).$
$\Box\,((q \rightarrow ((q\ \mathcal{U}\ r) \vee \Box\, q)))$	$\forall_i\ (\neg q_i \vee \neg r_{i+1} \vee q_{i+1}).$

Notes: $\forall_i$ means conjoin many clauses ending at some bound for i.
$\forall_i \ldots \exists_{j>i}$ means many clauses like $(a_i \vee b_{i+1}) \wedge (a_i \vee b_{i+2}) \wedge (a_i \vee b_{i+3}) \ldots$

Example - Elevator Controller



Pullies & motor

Counterweight

Elevator

Buttons

Inside door sensor

4th

3rd

Call buttons

2nd

1st

Outside door sensors

Floor sensors

Controller
with inputs from
elevator buttons,
call buttons,
floor sensors,
door sensors

Above figure is a schematic of an elevator servicing a 4 storey building
There is a controller that takes input from the following sensors:

Call button: Up/Down, each floor, send signal when pressed, not when released

Elev. button: 4 buttons, rider chooses floor, signal sent only when when pressed

Floor sensor: Each floor, send signal when elevator exactly aligns with floor

Out. door sensor: Each floor, send continuous signal when floor door is open

In. door sensor: Send continuous signal when elevator door is open

Example - Elevator Controller

Controller is responsible for proper functioning of the elevator:

1. A floor door is never open if the elevator is not present at that floor.
2. The elevator door is open only when stopped at a floor and when the outside door of that floor is also open.
3. The elevator door is always closed when in motion. The elevator does not begin to move until all doors are closed.
4. A floor requested by a waiting person will be served sometime - that is, a person cannot cancel a call.
5. When the top (penthouse) floor is requested (either from within the elevator or by a waiting party), that floor is served immediately - that is, the elevator does not stop on the way there.
6. Otherwise, if the elevator is descending and in use, it serves floors that have been called in the order they are reached by the elevator until there are no more waiting calls on any floor beneath the elevator's current position.
7. If the elevator is ascending and in use, it serves floors that have been called in the order they are reached by the elevator until there are no more waiting calls on any floor above the elevator's current position.
8. The elevator always returns to the 1^{st} floor when not in use.
9. The elevator never reverses direction while it is in motion.
10. The elevator door and floor doors always open and close together.

Not included for simplicity:

1. Capacity of elevator is not considered.
2. Inertia is not taken into account - elevator stops/starts instantly.

Example - Elevator Controller

Controller outputs (signals to devices):

1. Closes elevator door and floor door simultaneously. Signals doors.
2. Starts elevator to accend and signals when it should stop. Signals motor.
3. Starts elevator to descend and signals when it should stop. Signals motor.
4. Stops the elevator at a floor. Signals motor.
5. Prevents elevator from continuing below 1st floor and above 4th floor. Signals motor.
6. Resets lights on call buttons and elevator buttons when elevator lands on floors. Signals buttons.

Controller state (epoch i):

1. cb_j^i: call button on floor j - 0, if not pressed, 1 if up pressed, 2 if down pressed.
2. cbl_j^i: call button light on floor j - 0 if both off, 1 if up on only, 2 if down light on only, 3 if both on.
3. eb_j^i: elevator floor j button - 0 if not pressed, 1 if pressed.
4. ebl_j^i: elevator floor j button light - 0 if not lit, 1 if lit.
5. ids^i: elevator door sensor - 0 if door is open, 1 if closed.
6. ods_j^i: floor j door sensor - 0 if floor j door is open, 1 if it is closed.
7. mtr^i: motor direction - 0 if stopped, 1 if moving up, 2 if moving down.
8. fs_j^i: floor j sensor - 1 if elevator aligns with floor j, 0 if not.

Other state (epoch i):

1. pe^i: person in elevator - 0 if none, 1 if at least one.

Example - Elevator Controller

Define type `State` and write initial `State` in Cryptol:

```
  type State = { cb : [4][12],  // call buttons - 1/floor, 4 total
                cbl : [4][12],  // call button lights - 1/floor, 4 total
                 eb : [4][12],  // elevator buttons - 1/floor, 4 total
                ebl : [4][12],  // elevator button lights - 1/button, 4 total
                ids : [12],     // elevator door sensor
                ods : [4][12],  // outside door sensor - 1/floor, 4 total
                mtr : [12],     // motor direction, in motion
                 fs : [4][12],  // floor sensor - 1/floor, 4 total
                 pe : [12] }    // person(s) in elevator

  initState : State
  initState = { cb = [0,0,0,0], // no call buttons pressed
               cbl = [0,0,0,0], // all call button lights are off
                eb = [0,0,0,0], // no elevator buttons pressed
               ebl = [0,0,0,0], // elevator button lights are off
               ids = 0,         // elevator door is open
               ods = [0,1,1,1], // outside floor door is open on 1st floor only
               mtr = 0,         // elevator is stationary
                fs = [1,0,0,0], // elevator is on 1st floor
                pe = 0 }        // no one is in the elevator
```

In Cryptol - how to query a state using `initState` as example:

```
Main> :l elevator.cry
Main> :s base=10
Main> initState.fs@0
1
Main> initState.ods
[0, 1, 1, 1]
```

Example - Elevator Controller

Example BAD `State`:

```
badState : State
badState = { cb = [0,0,0,0], // no call buttons pressed
            cbl = [0,0,0,0], // all call button lights are off
             eb = [0,0,1,0], // 3rd floor elevator button is pressed
            ebl = [0,0,0,0], // elevator button lights are off
            ids = 1,         // elevator door is closed
            ods = [1,1,1,1], // outside floor doors are all closed
            mtr = 0,         // elevator is stationary
             fs = [1,0,0,0], // elevator is on 1st floor
             pe = 0 }        // no one is in the elevator
```

Legal `State` transitions must be defined, avoiding any bad `State`.

Once rules are defined for safe `State` transitions, they may be used as in `foxChickenCornRedux.cry` of Lesson 2.3 to establish a simulation of elevator behavior over some finite number of steps.

The rules define the behavior of the Contoller given sensor inputs.

Then queries such as "does the elevator reverse direction while in motion" may be made