# micromodels of software
## declarative modelling and analysis with Alloy

## lecture 4: a case study

Daniel Jackson
MIT Lab for Computer Science
Marktoberdorf, August 2002

# why looseness?

# why looseness?

risk-driven modelling

› give only crucial properties

# why looseness?

risk-driven modelling

- give only crucial properties

implementation freedom

- allow concurrency
- representation independence

# why looseness?

risk-driven modelling
- give only crucial properties

implementation freedom
- allow concurrency
- representation independence

account for environment
- fewer assumptions better

# why looseness?

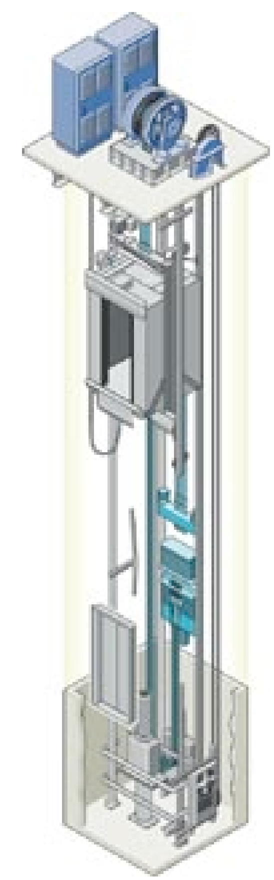
risk-driven modelling
- give only crucial properties

implementation freedom
- allow concurrency
- representation independence

account for environment
- fewer assumptions better

specify a family of systems
- every program is a family? [Parnas]

# example: elevator policy

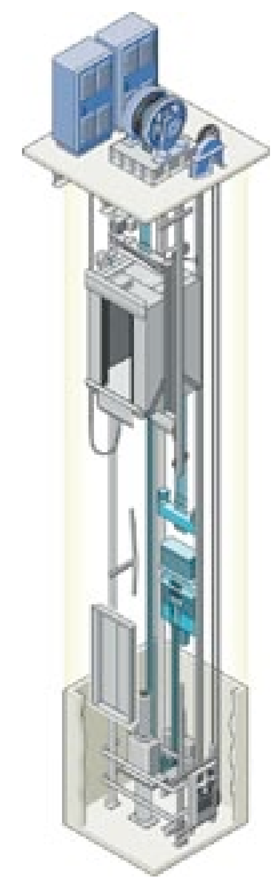


# example: elevator policy

challenge

› specify a policy for scheduling lifts

› keep concerns separated



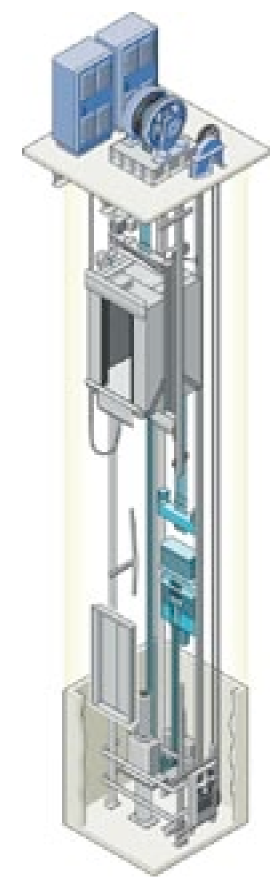
# example: elevator policy

challenge

- specify a policy for scheduling lifts
- keep concerns separated

tight enough

- all requests eventually served
- don't skip request from inside lift

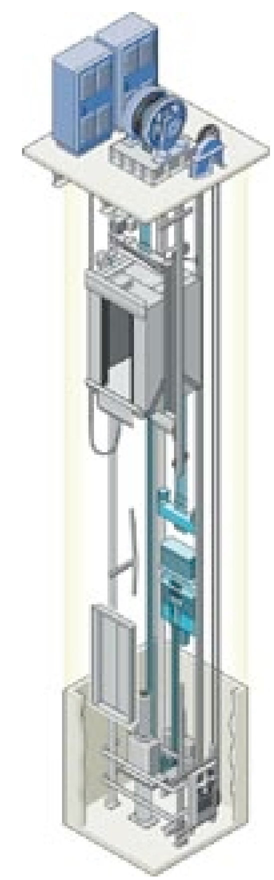


# example: elevator policy

challenge
- › specify a policy for scheduling lifts
- › keep concerns separated

tight enough
- › all requests eventually served
- › don’t skip request from inside lift

loose enough
- › no fixed configuration of floors, lifts, buttons
- › not one algorithm but a family



# complications

# complications

multiple lifts

› don’t send all to service one request

# complications

multiple lifts

- don’t send all to service one request

top and bottom

- lift going in wrong direction may be nearer

# complications

multiple lifts
- don’t send all to service one request

top and bottom
- lift going in wrong direction may be nearer

load balancing
- accommodate strategies based on occupancy, eg
- don’t force nearest lift to serve

# approach: promises

# approach: promises

ways to deny a request

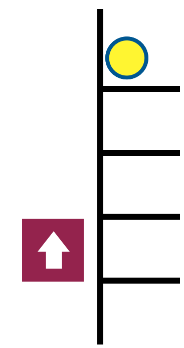
- ‘skipping’: going past floor
- ‘bouncing’: doubling back before floor

# approach: promises

ways to deny a request

› ‘skipping’: going past floor

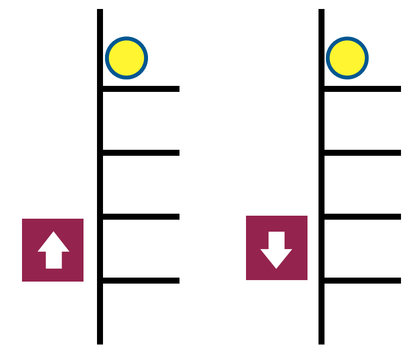
› ‘bouncing’: doubling back before floor



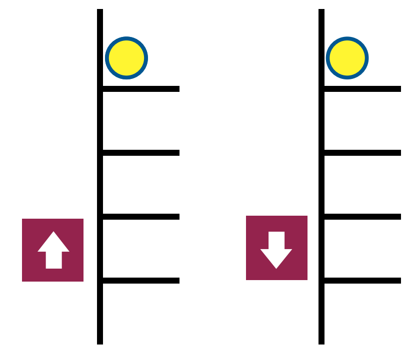
# approach: promises

ways to deny a request

- ‘skipping’: going past floor
- ‘bouncing’: doubling back before floor



# approach: promises



ways to deny a request

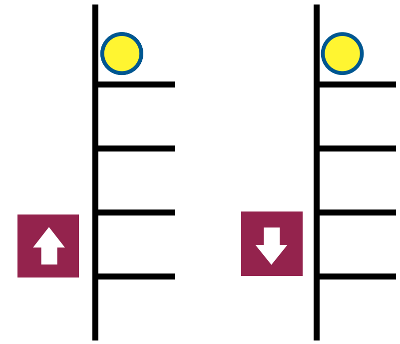
- ‘skipping’: going past floor
- ‘bouncing’: doubling back before floor

policy

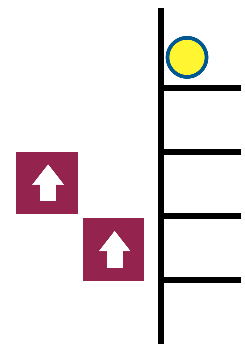
- a lift can’t deny a request from inside
- if a lift denies a floor request
  some lifts promise to serve it later
- a lift can’t deny the last promise

# approach: promises

ways to deny a request
- ‘skipping’: going past floor
- ‘bouncing’: doubling back before floor



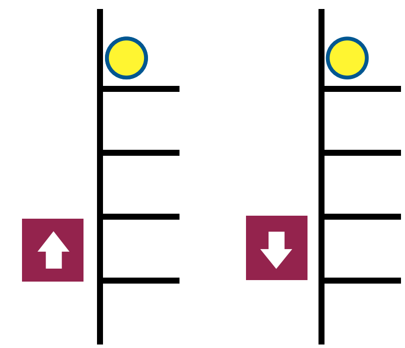
policy
- a lift can’t deny a request from inside
- if a lift denies a floor request
  some lifts promise to serve it later
- a lift can’t deny the last promise



# approach: promises

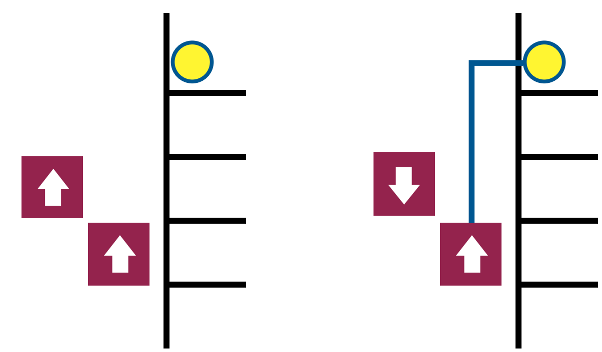
ways to deny a request

- ‘skipping’: going past floor
- ‘bouncing’: doubling back before floor



policy

- a lift can’t deny a request from inside
- if a lift denies a floor request
  some lifts promise to serve it later
- a lift can’t deny the last promise

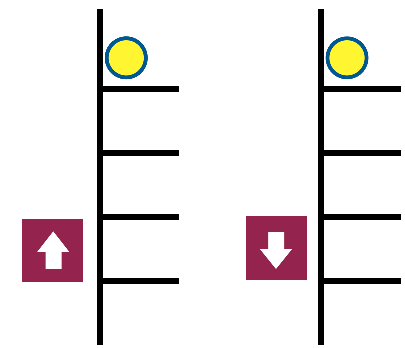


# approach: promises

ways to deny a request

› 'skipping': going past floor

› 'bouncing': doubling back before floor

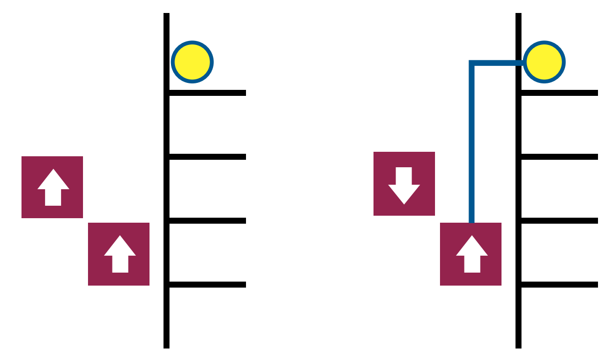


policy

› a lift can't deny a request from inside

› if a lift denies a floor request
  some lifts promise to serve it later

› a lift can't deny the last promise



freedoms

› divide requests amongst lifts

› postpone allocation decision

# basic abstractions

# basic abstractions

floor layout

- orderings above and below
- top and bottom floors

# basic abstractions

floor layout

- orderings above and below
- top and bottom floors

buttons

- inside lift and at floors
- each has an associated floor
- in a given state, some lit

# basic abstractions

floor layout

- orderings above and below
- top and bottom floors

buttons

- inside lift and at floors
- each has an associated floor
- in a given state, some lit

elevator state

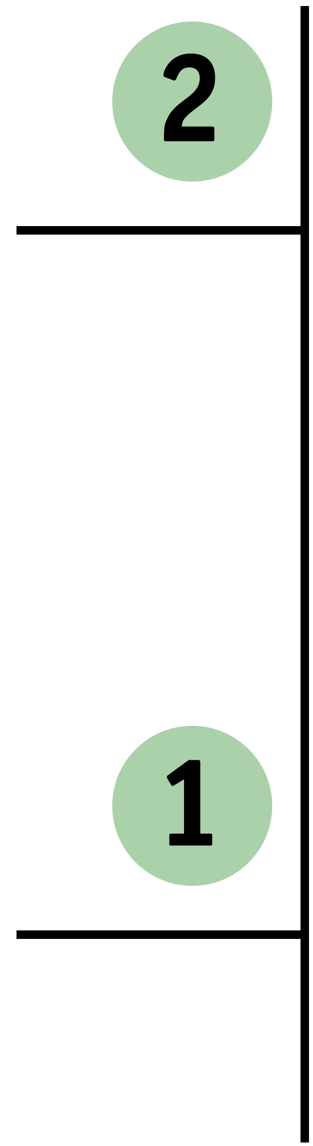
- at or approaching a floor
- rising or falling
- promises to serve some buttons

# basic abstractions

floor layout
- orderings above and below
- top and bottom floors

buttons
- inside lift and at floors
- each has an associated floor
- in a given state, some lit

elevator state
- at or approaching a floor
- rising or falling
- promises to serve some buttons

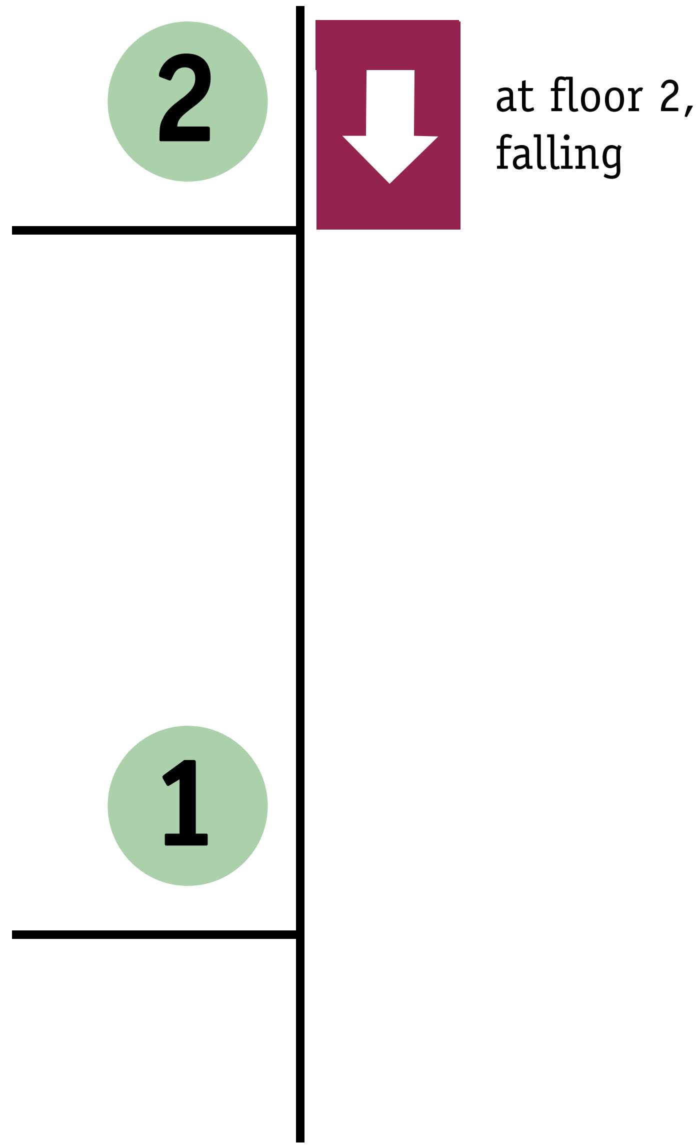


# basic abstractions

floor layout
- orderings above and below
- top and bottom floors

buttons
- inside lift and at floors
- each has an associated floor
- in a given state, some lit

elevator state
- at or approaching a floor
- rising or falling
- promises to serve some buttons



# basic abstractions

floor layout

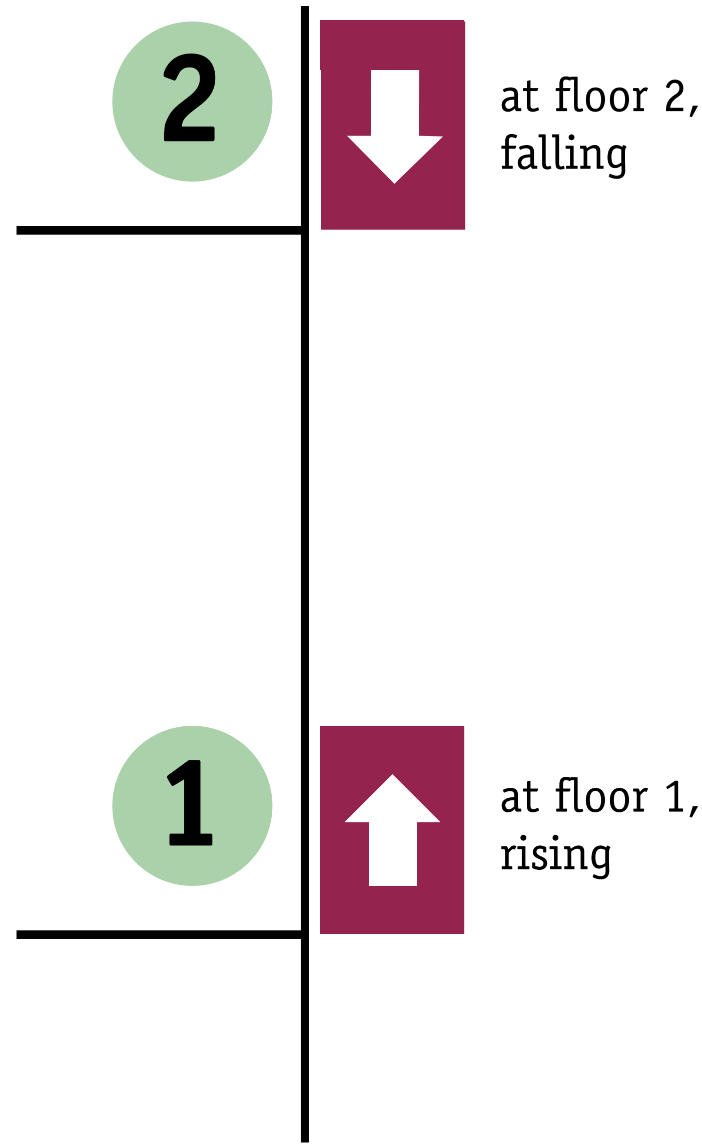
- orderings above and below
- top and bottom floors

buttons

- inside lift and at floors
- each has an associated floor
- in a given state, some lit

elevator state

- at or approaching a floor
- rising or falling
- promises to serve some buttons



# basic abstractions

floor layout

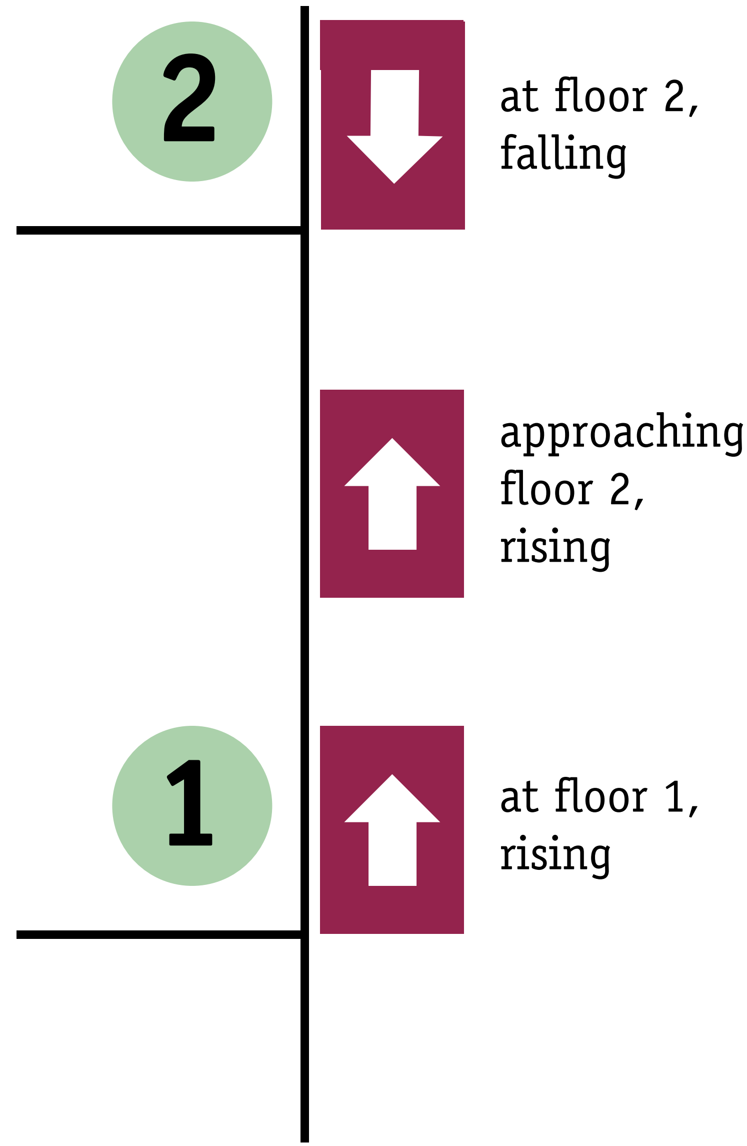
- orderings above and below
- top and bottom floors

buttons

- inside lift and at floors
- each has an associated floor
- in a given state, some lit

elevator state

- at or approaching a floor
- rising or falling
- promises to serve some buttons



# floor layout

# floor layout

```
open std/orders
```

# floor layout

```
open std/orders

sig Floor {
  disj up, down: option FloorButton,
  above, below: option Floor
  }
```

# floor layout

```
open std/orders

sig Floor {
  disj up, down: option FloorButton,
  above, below: option Floor
  }

sig Top extends Floor {} {no up}
```

# floor layout

```
open std/orders

sig Floor {
  disj up, down: option FloorButton,
  above, below: option Floor
  }

sig Top extends Floor {} {no up}

sig Bottom extends Floor {} {no down}
```

# floor layout

```
open std/orders

sig Floor {
  disj up, down: option FloorButton,
  above, below: option Floor
  }

sig Top extends Floor {} {no up}

sig Bottom extends Floor {} {no down}

fact Layout {
  Ord[Floor].next = above
  Ord[Floor].prev = below
  Ord[Floor].last = Top
  Ord[Floor].first = Bottom }
```

# floor layout

```
open std/orders

sig Floor {
  disj up, down: option FloorButton,
  above, below: option Floor
  }

sig Top extends Floor {} {no up}

sig Bottom extends Floor {} {no down}

fact Layout {
  Ord[Floor].next = above
  Ord[Floor].prev = below
  Ord[Floor].last = Top
  Ord[Floor].first = Bottom }
```

use ordering axioms from standard library

# floor layout

don't require buttons on all floors
allow small scope
analysis will place buttons demonically

```
open std/orders

sig Floor {
  disj up, down: option FloorButton,
  above, below: option Floor
  }

sig Top extends Floor {} {no up}

sig Bottom extends Floor {} {no down}

fact Layout {
  Ord[Floor].next = above
  Ord[Floor].prev = below
  Ord[Floor].last = Top
  Ord[Floor].first = Bottom }
```

use ordering axioms from standard library lifts

# lifts

```
sig Lift {
  button: Floor ?->? LiftButton,
  buttons: set LiftButton
  }
```

# lifts

```
sig Lift {
  button: Floor ?->? LiftButton,
  buttons: set LiftButton
  }
```

button panel:
allows different lifts
to cover different sets
of floors

# buttons

# buttons

```
sig Button {floor: Floor}
```

# buttons

```
sig Button {floor: Floor}
disj sig LiftButton extends Button {lift: Lift}
```

# buttons

```
sig Button {floor: Floor}
disj sig LiftButton extends Button {lift: Lift}
disj sig FloorButton extends Button {}
```

# buttons

```
sig Button {floor: Floor}
disj sig LiftButton extends Button {lift: Lift}
disj sig FloorButton extends Button {}
part sig UpButton, DownButton extends FloorButton {}
```

# buttons

```
sig Button {floor: Floor}
disj sig LiftButton extends Button {lift: Lift}
disj sig FloorButton extends Button {}
part sig UpButton, DownButton extends FloorButton {}
fact ButtonDefinitions {
  ~floor = Lift.button + up + down
  lift = {b: Button, p: Lift | some f: Floor | f->b in p.button}
  all p: Lift | p.buttons = p.button [Floor]
  UpButton = Floor.up
  }
```

# buttons

define classes of button;
redundant but convenient

```
sig Button {floor: Floor}
disj sig LiftButton extends Button {lift: Lift}
disj sig FloorButton extends Button {}
part sig UpButton, DownButton extends FloorButton {}
fact ButtonDefinitions {
  ~floor = Lift.button + up + down
  lift = {b: Button, p: Lift | some f: Floor | f->b in p.button}
  all p: Lift | p.buttons = p.button [Floor]
  UpButton = Floor.up
  }
```

# sample layout

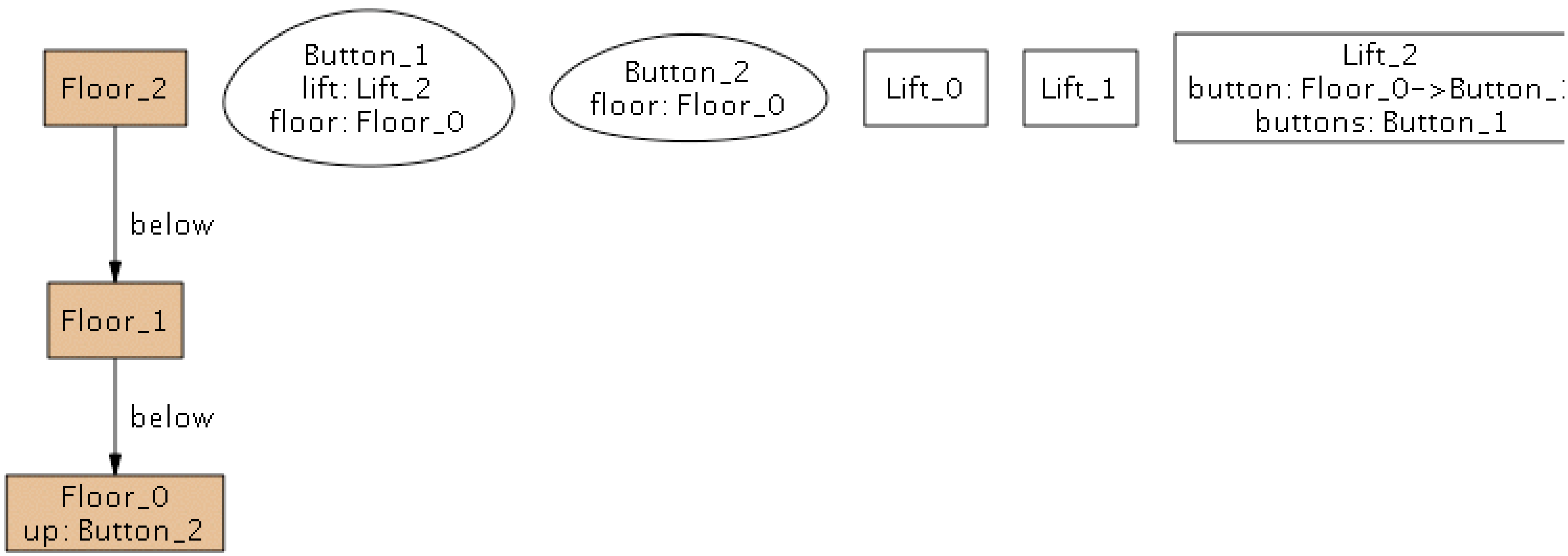
# sample layout

**fun** showLayout () {**some** Lift.buttons}

**run** showLayout

# sample layout

```
fun showLayout () {some Lift.buttons}
run showLayout
```



# system state

# system state

declaring state

› collect together relations that change

```
sig State {
  lit: set Button,
  part rising, falling: set Lift,
  at, approaching: Lift ->? Floor,
  promises: Lift -> FloorButton
  }
```

# system state

outstanding requests

declaring state

- collect together relations that change

```
sig State {
  lit: set Button,
  part rising, falling: set Lift,
  at, approaching: Lift ->? Floor,
  promises: Lift -> FloorButton
  }
```

# system state

declaring state

› collect together relations that change

```
sig State {
 lit: set Button,
 part rising, falling: set Lift,
 at, approaching: Lift ->? Floor,
 promises: Lift -> FloorButton
 }
```

outstanding requests

lift directions

# system state

declaring state

- collect together relations that change

```
sig State {
  lit: set Button,
  part rising, falling: set Lift,
  at, approaching: Lift ->? Floor,
  promises: Lift -> FloorButton
  }
```

outstanding requests

lift directions

lift positions

# system state

declaring state

› collect together relations that change

```
sig State {
  lit: set Button,
  part rising, falling: set Lift,
  at, approaching: Lift ->? Floor,
  promises: Lift -> FloorButton
  }
```

outstanding requests

lift directions

lift positions

promises: many to many

# physical constraints on lift state

# physical constraints on lift state

```
fun LiftPosition (s: State) {
  all p: Lift | with s {
    one (at + approaching)[p]
    no (at & approaching)[p]
    p in rising =>
      no approaching[p] & Bottom,
      no approaching[p] & Top
    p in rising =>
      no at[p] & Top,
      no at[p] & Bottom
    }
  }
```

# physical constraints on lift state

```
fun LiftPosition (s: State) {
  all p: Lift | with s {
    one (at + approaching)[p]
    no (at & approaching)[p]
    p in rising =>
      no approaching[p] & Bottom,
      no approaching[p] & Top
    p in rising =>
      no at[p] & Top,
      no at[p] & Bottom
    }
  }
```

```
sig State {
  at, approaching: Lift ->? Floor,
  part rising, falling: set Lift ...}
```

rising *short for* s.rising

# physical constraints on lift state

```
fun LiftPosition (s: State) {
  all p: Lift | with s {
    one (at + approaching)[p]
    no (at & approaching)[p]
    p in rising =>
      no approaching[p] & Bottom,
      no approaching[p] & Top
    p in rising =>
      no at[p] & Top,
      no at[p] & Bottom
    }
  }
```

lift is at or approaching one floor

```
sig State {
  at, approaching: Lift ->? Floor,
  part rising, falling: set Lift ...}
```

rising *short for* s.rising

# physical constraints on lift state

```
fun LiftPosition (s: State) {
  all p: Lift | with s {
    one (at + approaching)[p]
    no (at & approaching)[p]
    p in rising =>
      no approaching[p] & Bottom,
      no approaching[p] & Top
    p in rising =>
      no at[p] & Top,
      no at[p] & Bottom
    }
  }
```

lift is at or approaching one floor

lift is not at *and* approaching

```
sig State {
  at, approaching: Lift ->? Floor,
  part rising, falling: set Lift ...}
```

rising *short for* s.rising

# physical constraints on lift state

```
fun LiftPosition (s: State) {
  all p: Lift | with s {
    one (at + approaching)[p]
    no (at & approaching)[p]
    p in rising =>
      no approaching[p] & Bottom,
      no approaching[p] & Top
    p in rising =>
      no at[p] & Top,
      no at[p] & Bottom
    }
  }
```

lift is at or approaching one floor

lift is not at *and* approaching

no rising on approach to bottom

```
sig State {
  at, approaching: Lift ->? Floor,
  part rising, falling: set Lift ...}
```

rising *short for* s.rising

# physical constraints on lift state

```
fun LiftPosition (s: State) {
  all p: Lift | with s {
    one (at + approaching)[p]
    no (at & approaching)[p]
    p in rising =>
      no approaching[p] & Bottom,
      no approaching[p] & Top
    p in rising =>
      no at[p] & Top,
      no at[p] & Bottom
  }
}
```

lift is at or approaching one floor

lift is not at *and* approaching

no rising on approach to bottom

no rising at top floor

```
sig State {
  at, approaching: Lift ->? Floor,
  part rising, falling: set Lift ...}
```

rising *short for* s.rising

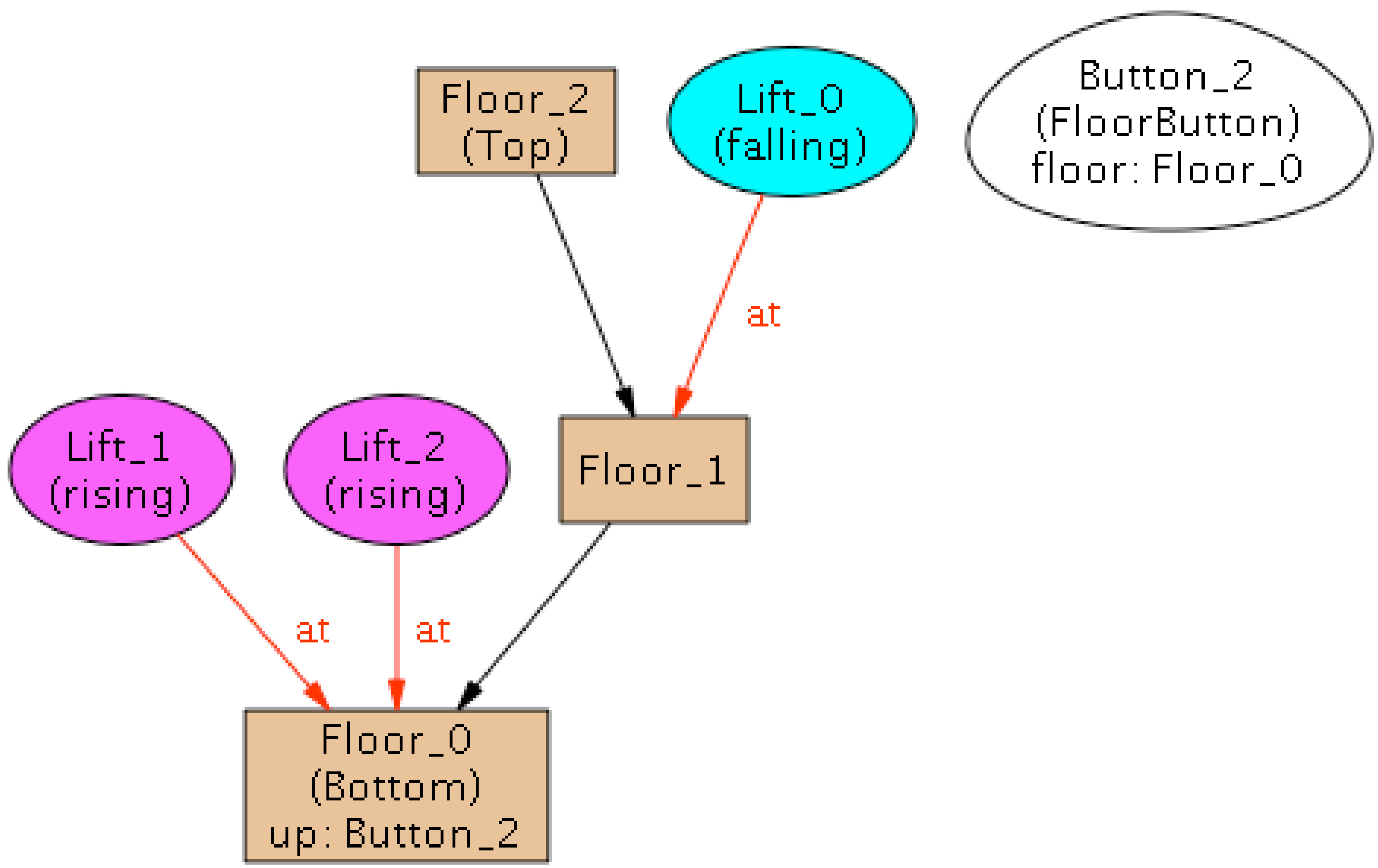
# sample state

# sample state

```
run LiftPosition
```

# sample state

**run** LiftPosition



# physical constraints on lift motion

# physical constraints on lift motion

```
fun nextFloor (s: State, p: Lift): Floor -> Floor {
  result = if p in s.rising then above else below
  }
```

# physical constraints on lift motion

```
fun nextFloor (s: State, p: Lift): Floor -> Floor {
  result = if p in s.rising then above else below
  }

fun LiftMotion (s, s': State) {
  all p: Lift {
    p & s.rising != p & s'.rising => some s'.at[p]
```

# physical constraints on lift motion

```
fun nextFloor (s: State, p: Lift): Floor -> Floor {
  result = if p in s.rising then above else below
  }
fun LiftMotion (s, s': State) {
  all p: Lift {
    p & s.rising != p & s'.rising => some s'.at[p]
    s'.at[p] in s.(at + approaching)[p]
```

# physical constraints on lift motion

```
fun nextFloor (s: State, p: Lift): Floor -> Floor {
  result = if p in s.rising then above else below
  }

fun LiftMotion (s, s': State) {
  all p: Lift {
    p & s.rising != p & s'.rising => some s'.at[p]
    s'.at[p] in s.(at + approaching)[p]
    s'.approaching[p] in
      s.approaching[p] + s.(at + approaching)[p].nextFloor(s,p)
    }
  }
```

# physical constraints on lift motion

```
fun nextFloor (s: State, p: Lift): Floor -> Floor {
  result = if p in s.rising then above else below
  }

fun LiftMotion (s, s': State) {
  all p: Lift {
    p & s.rising != p & s'.rising => some s'.at[p]

    s'.at[p] in s.(at + approaching)[p]

    s'.approaching[p] in
      s.approaching[p] + s.(at + approaching)[p].nextFloor(s,p)
    }
  }
```

no dir change except at floor

# physical constraints on lift motion

```
fun nextFloor (s: State, p: Lift): Floor -> Floor {
  result = if p in s.rising then above else below
  }

fun LiftMotion (s, s': State) {
  all p: Lift {
    p & s.rising != p & s'.rising => some s'.at[p]
    s'.at[p] in s.(at + approaching)[p]

    s'.approaching[p] in
      s.approaching[p] + s.(at + approaching)[p].nextFloor(s,p)
    }
  }
```

no dir change except at floor

floor at after is floor at or approaching before

# physical constraints on lift motion

```
fun nextFloor (s: State, p: Lift): Floor -> Floor {
  result = if p in s.rising then above else below
  }

fun LiftMotion (s, s': State) {
  all p: Lift {
    p & s.rising != p & s'.rising => some s'.at[p]
    s'.at[p] in s.(at + approaching)[p]
    s'.approaching[p] in
      s.approaching[p] + s.(at + approaching)[p].nextFloor(s,p)
    }
  }
```

no dir change except at floor

floor at after is floor at or approaching before

floor approaching after is floor approached before, or next floor

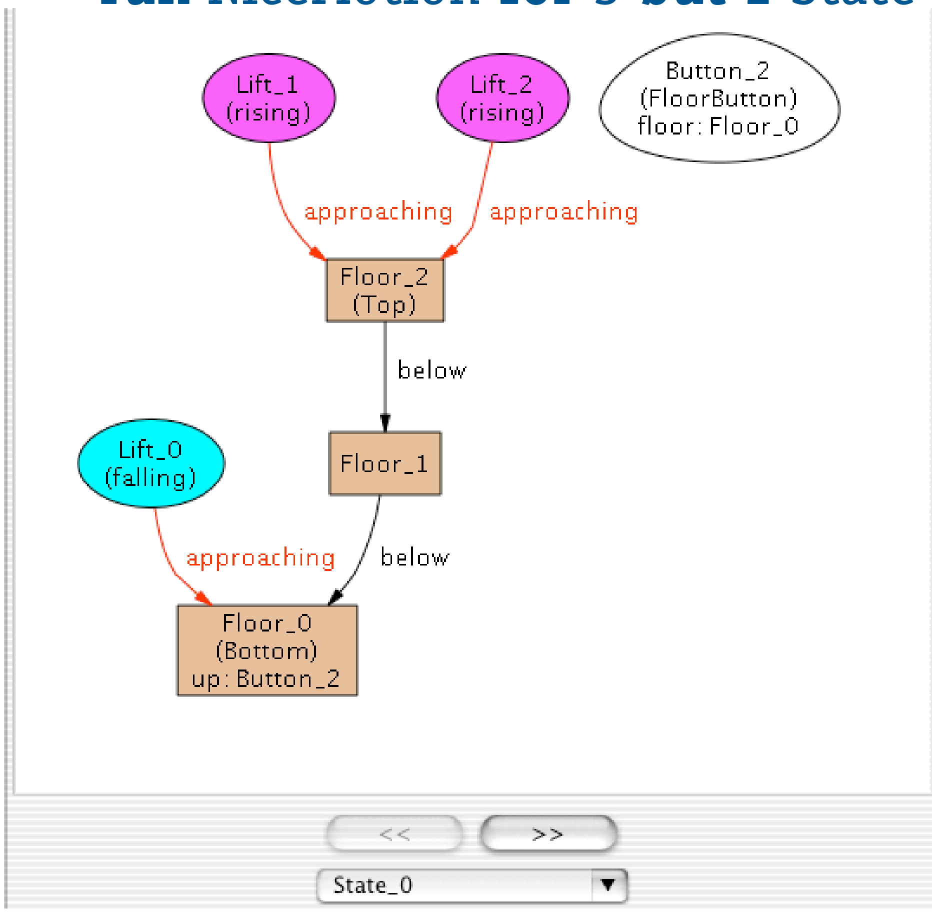
# sample transition

# sample transition

```
fun NiceMotion (s, s': State) {
  LiftMotion (s,s') && LiftPosition (s) && LiftPosition (s')
  s.at != s'.at}

run NiceMotion for 3 but 2 State
```

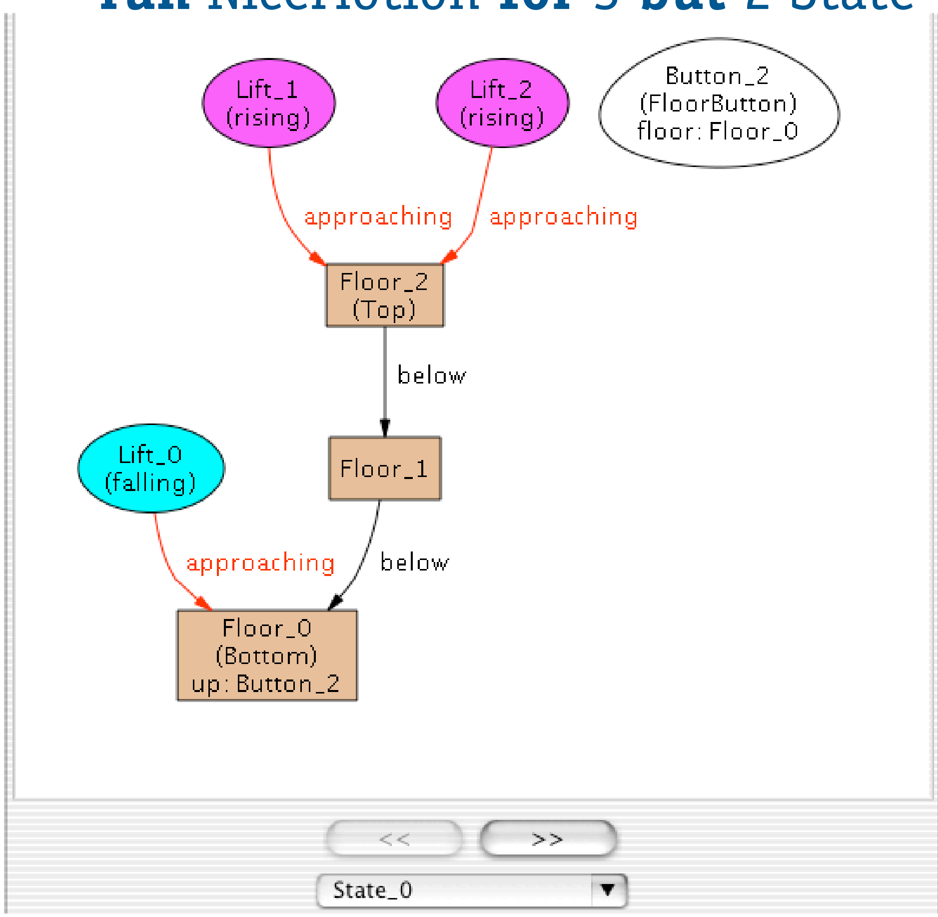
# sample transition

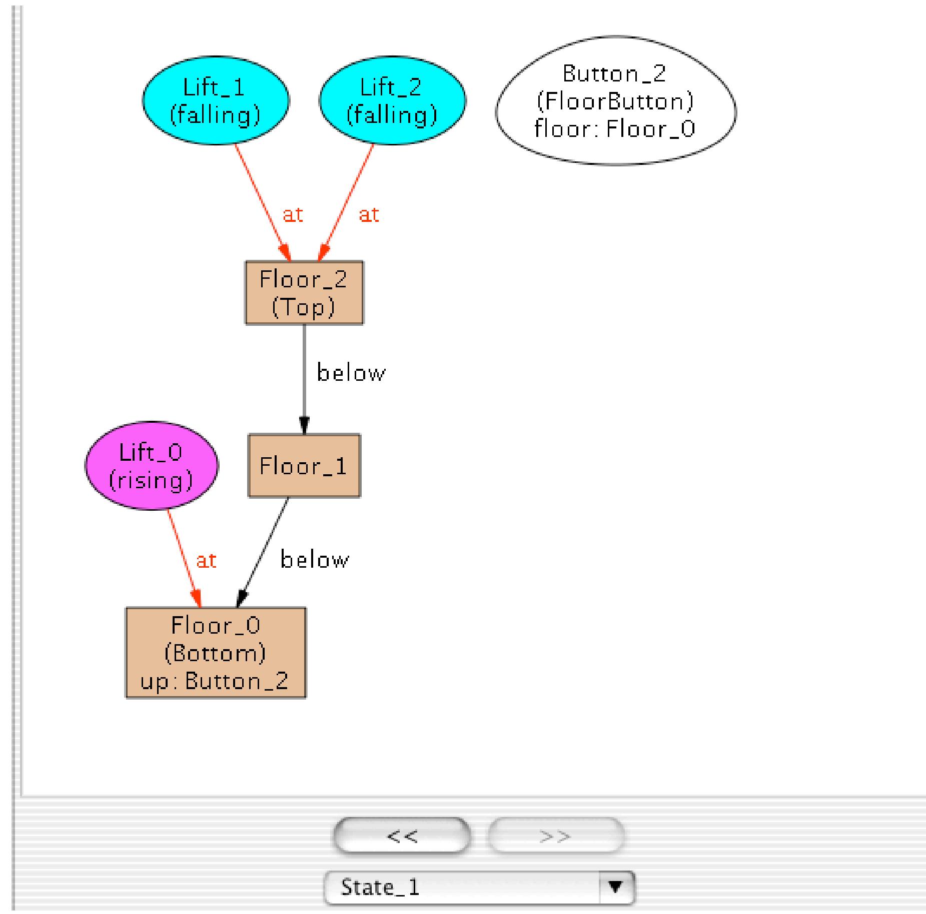
```
fun NiceMotion (s, s': State) {
  LiftMotion (s,s') && LiftPosition (s) && LiftPosition (s')
  s.at != s'.at}

run NiceMotion for 3 but 2 State
```



# sample transition

```
fun NiceMotion (s, s': State) {
  LiftMotion (s,s') && LiftPosition (s) && LiftPosition (s')
  s.at != s'.at}

run NiceMotion for 3 but 2 State
```



# button update

# button update

```
fun ButtonUpdate (s, s': State, press: set Button) {
  s'.lit = s.lit -
    {b: Button | some p: Lift | Serves (s,s',p,b)}
    + press
  no b: press & LiftButton | b.floor in (s+s').at[b.lift]
  no press & s.lit
  s.promises[Lift] - s'.promises[Lift] in s.lit - s'.lit
  }
```

# denying service

# denying service

```
fun Towards (s: State, p: Lift, f: Floor) {
  let next = nextFloor(s,p) |
    f in s.at[p].^next + s.approaching[p].*next
  }
```

# denying service

```
fun Towards (s: State, p: Lift, f: Floor) {
  let next = nextFloor(s,p) |
    f in s.at[p].^next + s.approaching[p].*next
  }
fun Denies (s, s': State, p: Lift, b: Button) {
  let f = b.floor {
    Towards (s,p,f)
    not Towards (s',p,f)
    not Serves (s,s',p,b)
    }
  }
```

# a policy

# a policy

```
fun Policy (s, s': State) {
  no p: Lift, b: p.buttons & s.lit | Denies (s,s',p,b)
  all b: s.lit & FloorButton, p: Lift |  Denies (s,s',p,b) =>
    (some q: Lift | Serves(s,s',q,b))
    or (b in s'.promises[Lift]
      and some b': s.lit | Towards (s',p,b'.floor))
  NoStuckLift (s,s')
  AvoidStops (s,s')
  }
```

# a policy

don't deny lift buttons

```
fun Policy (s, s': State) {
 no p: Lift, b: p.buttons & s.lit | Denies (s,s',p,b)
 all b: s.lit & FloorButton, p: Lift |  Denies (s,s',p,b) =>
   (some q: Lift | Serves(s,s',q,b))
   or (b in s'.promises[Lift]
     and some b': s.lit | Towards (s',p,b'.floor))
 NoStuckLift (s,s')
 AvoidStops (s,s')
 }
```

# a policy

```
fun Policy (s, s': State) {
 no p: Lift, b: p.buttons & s.lit | Denies (s,s',p,b)
 all b: s.lit & FloorButton, p: Lift |  Denies (s,s',p,b) =>
   (some q: Lift | Serves(s,s',q,b))
   or (b in s'.promises[Lift]
     and some b': s.lit | Towards (s',p,b'.floor))
 NoStuckLift (s,s')
 AvoidStops (s,s')
 }
```

don't deny lift buttons

if deny floor button
  some lift serves it
  or some lift promises to

# putting it all together

# putting it all together

```
fun Trans (s, s': State) {
  LiftPosition (s)
  LiftPosition (s')
  LiftMotion (s,s')
  Policy (s,s')
  some press: set Button | ButtonUpdate (s, s', press)
  }
```

# putting it all together

```
fun Trans (s, s': State) {
  LiftPosition (s)
  LiftPosition (s')
  LiftMotion (s,s')
  Policy (s,s')
  some press: set Button | ButtonUpdate (s, s', press)
  }
```

in a transition, some set of buttons is pressed and buttons are updated

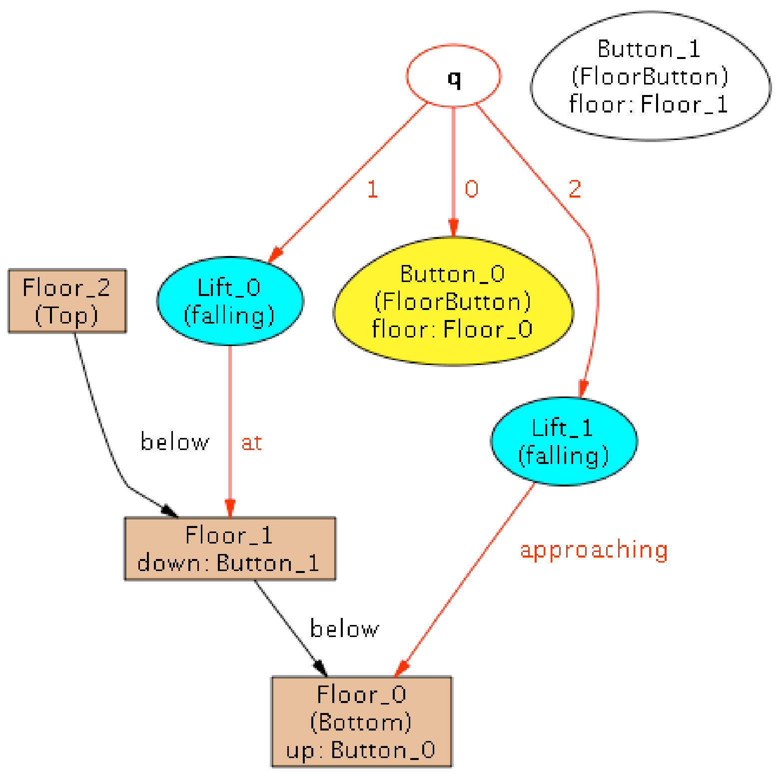
# sample denial

# sample denial

```
fun ShowPolicy (s, s': State) {
  Trans (s, s')
  some b: s.lit & FloorButton, p: Lift | Denies (s,s',p,b)
  no s.promises && some s'.promises}

run ShowPolicy for 3 but 2 State, 2 Lift, 2 Button
```

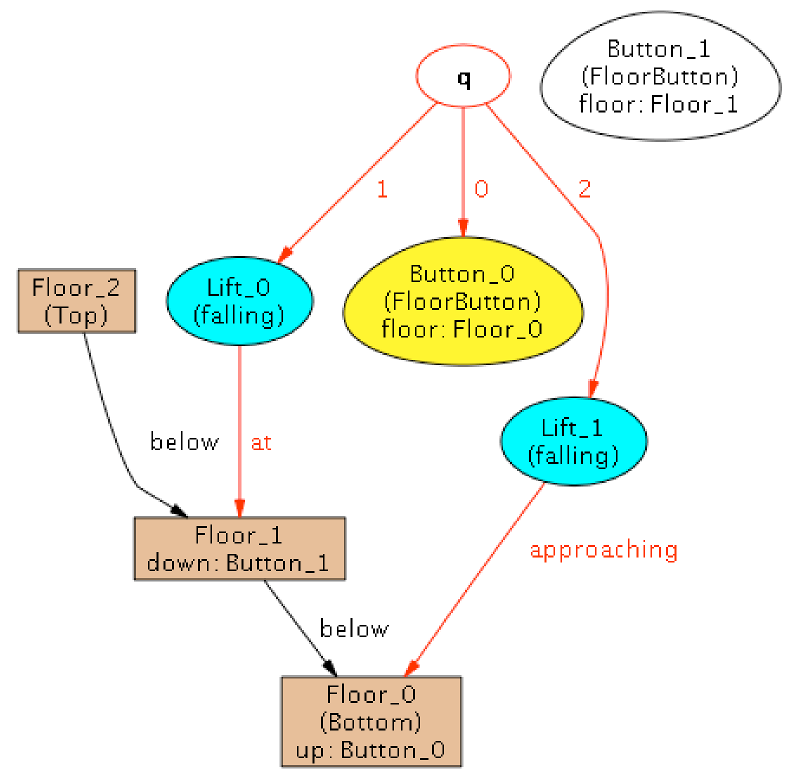
# sample denial

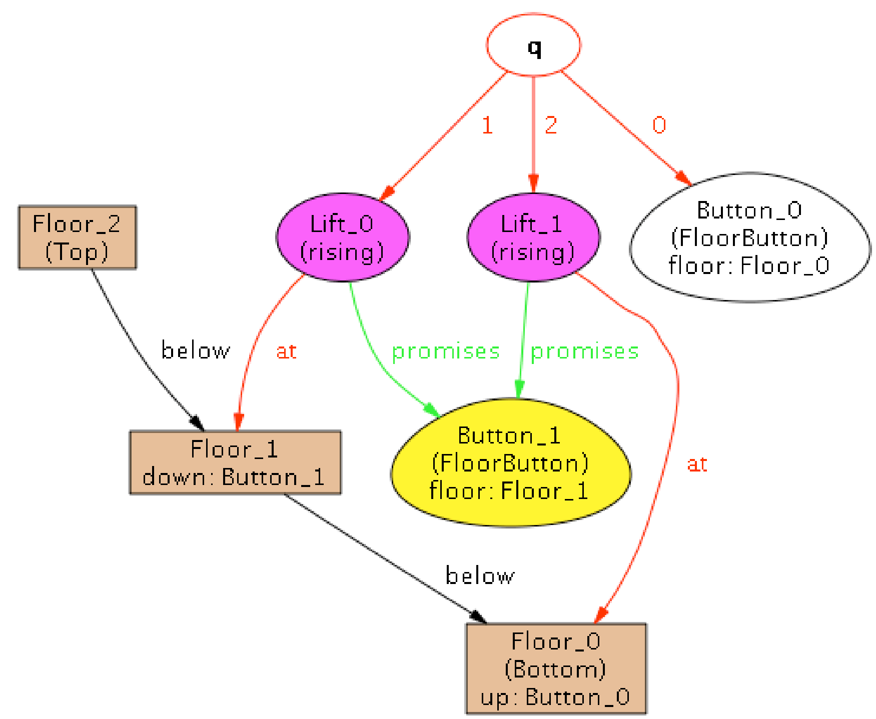
```
fun ShowPolicy (s, s': State) {
  Trans (s, s')
  some b: s.lit & FloorButton, p: Lift | Denies (s,s',p,b)
  no s.promises && some s'.promises}

run ShowPolicy for 3 but 2 State, 2 Lift, 2 Button
```



# sample denial

```
fun ShowPolicy (s, s': State) {
  Trans (s, s')
  some b: s.lit & FloorButton, p: Lift | Denies (s,s',p,b)
  no s.promises && some s'.promises}

run ShowPolicy for 3 but 2 State, 2 Lift, 2 Button
```



# traces

# traces

```
fun Init (s: State) {
  no s.lit.floor &  s.at[Lift]
  no s.promises
  }

fun Trace () {
  Init (Ord[State].first)
  all s: State - Ord[State].last |
    let s' = Ord[State].next[s] | Trans (s,s')
  }
```

# traces

```
fun Init (s: State) {
  no s.lit.floor &  s.at[Lift]
  no s.promises
  }

fun Trace () {
  Init (Ord[State].first)
  all s: State - Ord[State].last |
     let s' = Ord[State].next[s] | Trans (s,s')
  }
```

initial condition holds in first state

# traces

```
fun Init (s: State) {
  no s.lit.floor &  s.at[Lift]
  no s.promises
  }

fun Trace () {
  Init (Ord[State].first)
  all s: State - Ord[State].last |
    let s' = Ord[State].next[s] | Trans (s,s')
  }
```

initial condition holds in first state

transition relation relates each state except the last to the next state

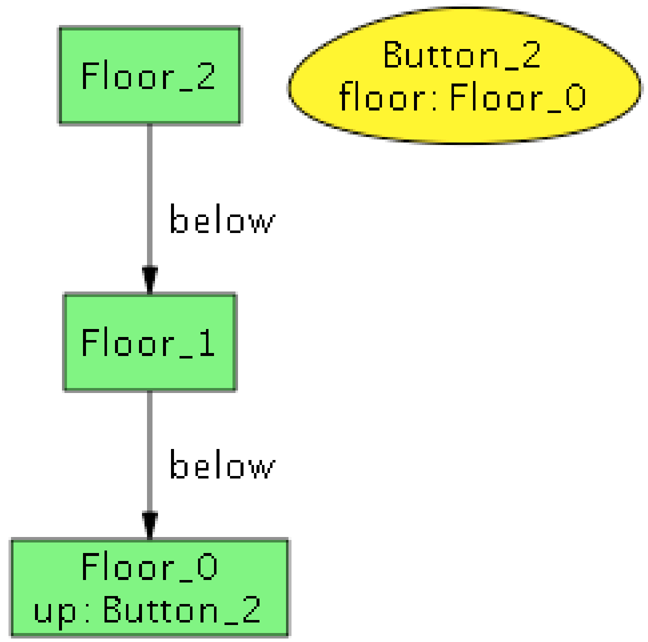
# asserting eventual service

# asserting eventual service

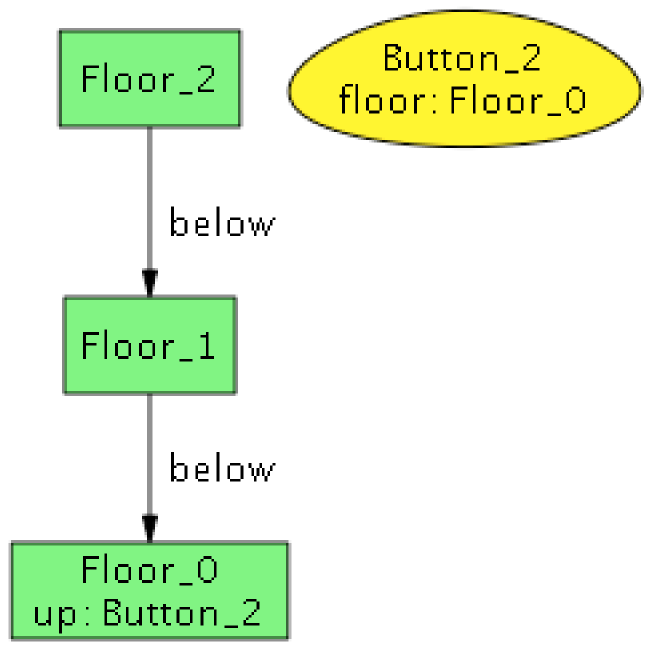
```
assert EventuallyServed {
  Trace () =>
    let start = Ord[State].first {
      all b: start.lit | some s': OrdNexts (start) | b !in s'.lit
      }
  }
```

# counterexample!

# counterexample!

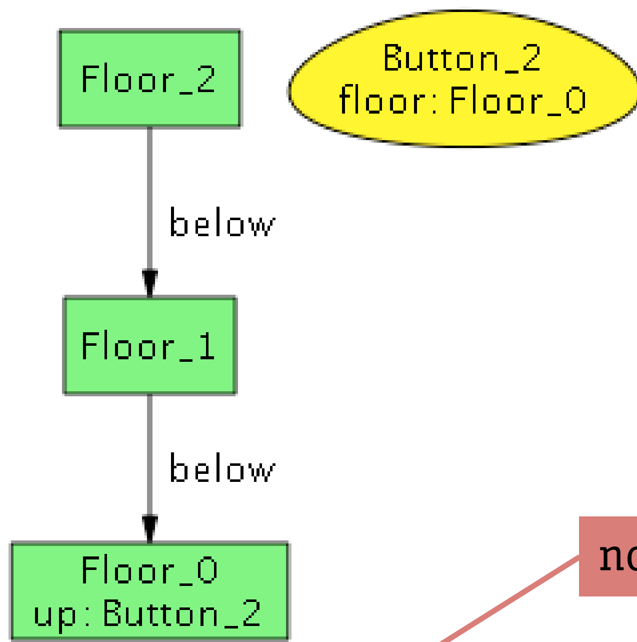


# counterexample!




```
assert EventuallyServed {
  Trace () and some Lift =>
    let start = Ord[State].first {
      all b: start.lit | some s': OrdNexts (start) | b !in s'.lit
      }
  }
```

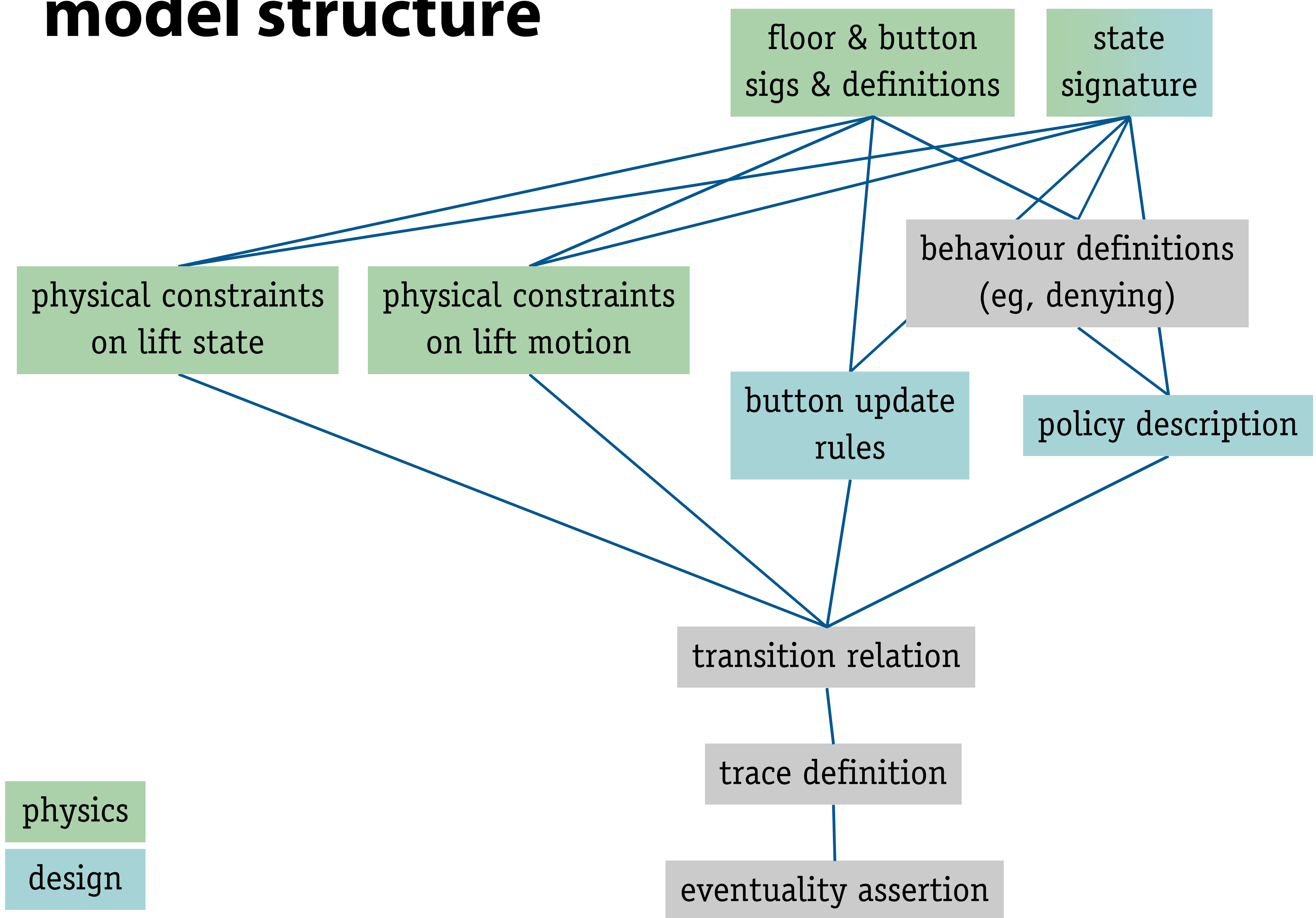
# counterexample!



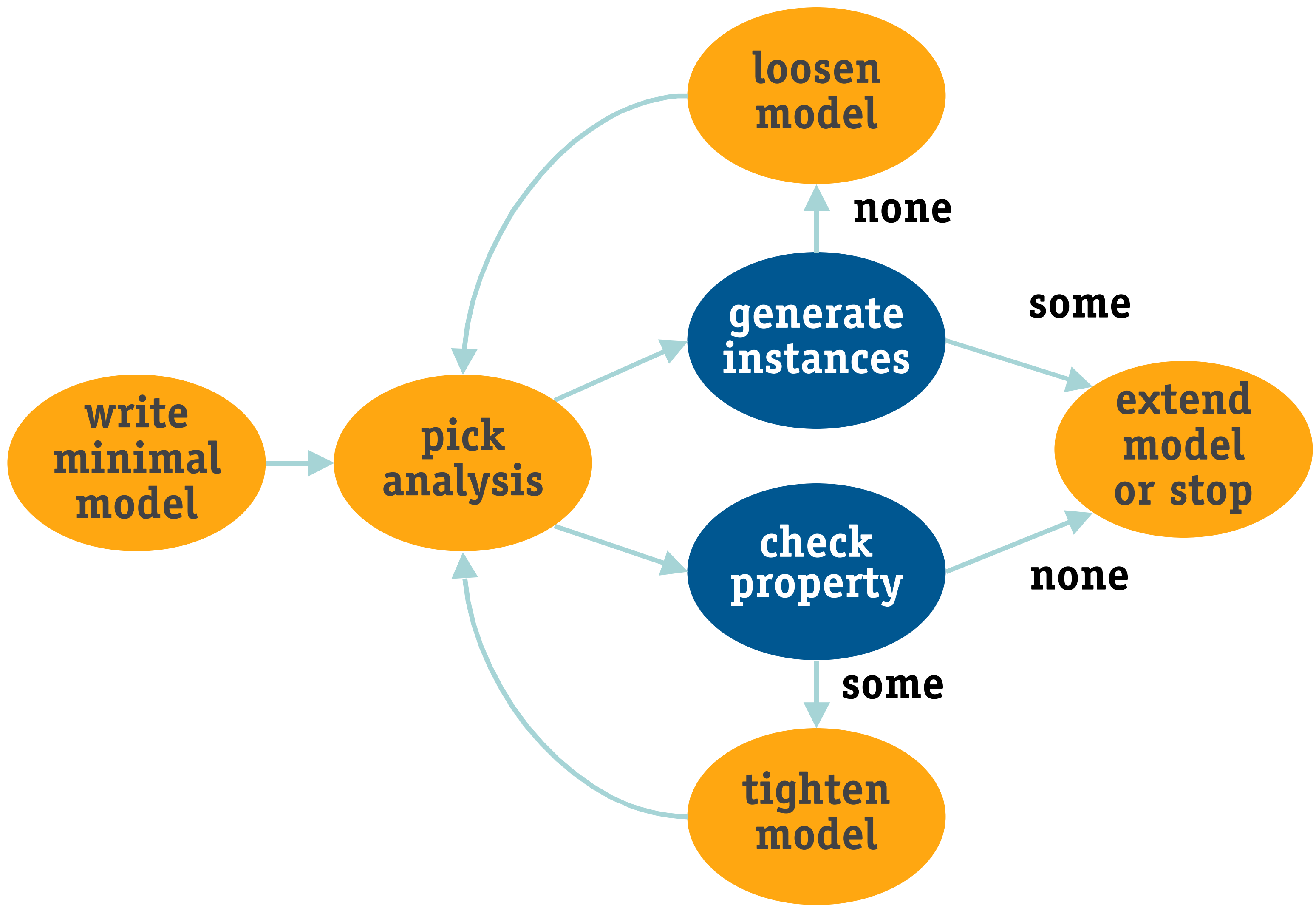

no service without lifts

```
assert EventuallyServed {
  Trace () and some Lift =>
    let start = Ord[State].first {
      all b: start.lit | some s': OrdNexts (start) | b !in s'.lit
      }
  }
```

# model structure



# incremental development



# challenges for you

# challenges for you

key properties of all lift systems

- what are they?
- are they just cultural?

# challenges for you

key properties of all lift systems

- what are they?
- are they just cultural?

replacing promises

- a better way to allow load balancing?