Lecture 4: a case study with Alloy and analysis of software models
Every program is a family? [Parnas]

Specify a family of systems?

fewer assumptions better

account for environment

representation independence

allow concurrency

implementation freedom

give only crucial properties

risk-driven modelling

Why looseness?
Challenge

Example: Elevator policy

Loose enough

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

Tight enough

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

Keep concerns separated

- specify a policy for scheduling lifts
**Complications**

- Don't force nearest lift to serve.
- Load balancing.
- Lift going in wrong direction may be nearer.
- Top and bottom.
- Don't send all to service one request.
- Multiple lifts.
approach: promises

postpone allocation decision
divide requests amongst lifts
freedoms

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

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

approach: promises
basic abstractions

floor layout

above and below floors

buttons inside lift and at floors each has an associated floor

in a given state, some light

promise to serve some buttons

rising or falling

at or approaching a floor

floor layout
Floor layout

sig Bottom extends Floor { no down }

sig Top extends Floor {} { no up }

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

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

use ordering axioms from standard library

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

{ button: set LiftButton
  buttons:
  lift
}
### Fact ButtonDefinitions

```plaintext
{ 
  up = p.button = all f: Floor | f->p.button
  lift = some b: Button, p: Lift | b->p.button + p.buttons = p.button
  ~floor = Lift.button + up + down

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

### Define classes of button:

- `FloorButton` extends `Button` with a `floor` attribute.
- `UpButton` and `DownButton` are subclasses of `FloorButton`.

#### Redundant but convenient

```
{ 
  Button {floor: Floor}
  LiftButton {lift: Lift}
  FloorButton {floor: Floor}
}
```
fun showLayout
{
  some Lift.buttons
}

run showLayout
Outstanding requests

directing state

Collect together relations that change system state

Promises: many to many

Lift directions

Lift positions
Physical constraints on lift state

fun LiftPosition (s: State)
{
    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
    }
}
Sample state

Run LiftPosition
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] ins.approaching[p] + s.(at + approaching)[p].nextFloor(s,p) }
}
run NiceMotion for 3 but 2 State

{ s.at = s'at
  LittMotion (s,s') & LittPosition (s) & LittPosition (s')
} run NiceMotion (s,s': State)

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

button update
{ }

{ }

not Services (q, p, s, s')
not Towards (q, p, s, s')
not Towards (q, p, s, s')
let f = p.floor
}

fun Denies (s, s': State, p: Lift, b: Button) {

fun

towards (s: State, p: Lift, f: Floor) {

let

towards = nextFloor (s, p, f)
let

towards = nextFloor (s, p, f)
}

deny service
a policy

Don't deny lift buttons if deny floor button or some lift promises to serve it and some of its promises (s'.pitch'lloof) or (a lift promises lift) (some y lift serves s'.y, y) so all of s'll be blocked. s lift (degens (s'.s, y, p, d))}

AvoidStops (s', s)
NoStuckLift (s', s)

If deny floor button or some lift promises to serve it, don't deny lift buttons.
fun Trans (s, s': State) {LiftPosition (s)LiftPosition (s')LiftMotion (s, s') Policy (s, s')}

putting it all together

in a transition, some set of buttons is pressed and buttons are updated
run ShowPolicy for 3 but 2 State, 2 Lift, 2 Button

\{ no \cdot promises \& some s'\cdot promises
some b: s'.lit \& FloorButton, p: Lift | Denies (s,s',p,b)
Trans (s,s')
\}

run ShowPolicy (s', s: State)

Sample denial
fun Trace () {
    let s' = Ord[State].next[s] | Trans (s, s')

    { 
        let s = Ord[State].first | Traces (s, s')
        all a State - Ord[State].last | Initial condition holds in first state
        transition relation relates each state except the last to the next state
    } ()
}

fun Trace () {
    no promises
    no s.lit.floor & s.at[Lift]

    } (s: State) 

traces
asserting eventual service

```haskell
assert EventuallyServed { Trace () =>
  let start = Ord[State].first
    in
  all b: start.lit | some s: OrdNexts (start) | b !
    in
  s'.lit }
```
assert EventuallyServed { Trace () and some (lit => no service without lifts)
Incremental development
challenges for you

* a better way to allow load balancing?
* replacing promises

* are they just cultural?
* what are they?

key properties of all lift systems