Lecture 4: a case study

and analysis with Alloy
declarative modeling
micromodels of software
Why looseness?
risk-driven modelling

give only crucial properties

why looseness?
why looseness?

- representation independence
- allow concurrency
- implementation freedom
- give only crucial properties
- risk-driven modeling

Why looseness?
why looseness? Fewer assumptions better account for environment representation independence allow concurrency implementation freedom give only crucial properties risk-driven modelling
why looseness?

- every program is a family [Parnas]
- fewer assumptions better
- account for environment
- representation independence
- allow concurrency
- implementation freedom
- give only crucial properties
- risk-driven modelling

specific family of systems
example: elevator policy
example: elevator policy

- Keep concerns separated
- Specify a policy for scheduling lifts

challenge
Example: Elevator Policy

- don't skip request from inside lift
- all requests eventually served
- tight enough
- keep concerns separated
- specify a policy for scheduling lifts

Challenge
Example: Elevator policy

- Tight enough
  - All requests eventually served
  - Keep concerns separated
- Loose enough
  - Don't skip requests from inside lift
  - No fixed configuration of floors, lifts, buttons
  - Not one algorithm but a family
- Challenge
complications
complications

> don't send all to service one request

multiple kites

complications
complications

- Lift going in wrong direction may be nearer top and bottom
- Don't send all to service one request multiple lifts
complications

- don’t force nearest lift to serve
- accommodate strategies based on occupancy, etc.
- 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
approach: promises

ways to deny a request:

- skipping: going past floor
- bouncing: doubling back before floor
approach: promises

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

ways to deny a request
approach: promises

ways to deny a request

- skipping: going past floor
- bouncing: doubling back before floor
Some lifts promise to serve it later. If a lift denies a floor request, a lift can’t deny a request from inside. A lift can’t deny the last promise. A lift can’t deny the request before floor going past floor. Skipping: going past floor. Bouncing: doubling back before floor.
approach: promises

\[ a \text{ lift can't deny the last promise} \]

\[ \text{some lifts promise to serve it later} \]

\[ \text{if a lift denies a floor request} \]

\[ \text{a lift can't deny a request from inside} \]

\[ \text{policy} \]

\[ \text{bouncing: doubling back before floor} \]

\[ \text{skipping: going past floor} \]

\[ \text{ways to deny a request} \]
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

Policy

approach: promises

- skipping: going past floor
- bouncing: doubling back before floor
- ways to deny a request
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, if a lift denies a request from inside
- policy

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

floor layout

orderings above and below

top and bottom floors
Floor layout

floor layout

orderings

above and below

floors

buttons

in a given state, some lit

each has an associated floor

inside lit and at floors

buttons

top and bottom floors

orderings above and below

floor layout

basic abstractions
basic abstractions

floor layout

orderings

above and below floors

buttons inside lift and at floors each has an associated floor

in a given state, some lit

elevator state promises to serve some buttons

rising or falling

at or approaching a floor

each floor promises to serve some buttons

top and bottom floors

orderings above and below

floor layout
Basic abstractions

1. Elevator state
   - In a given state, some lights
   - Each has an associated floor
   - In the lit at floor
   - Buttons

2. Floor layout
   - Or, elevators above and below
   - Top and bottom floors
   - Promises to serve some buttons
   - Rising or falling
   - At or approaching a floor
basic abstractions

Floor layout

Orderings above and below

Top and bottom floors

Each has an associated floor

Interior lit at floors

Promises to serve some buttons

Elevator state

In a given state, some lit

Rising or falling

At or approaching a floor

Falling at floor 2

Falling at floor 2,
floor layout

above and below floors

buttons

elevator state

rising or falling at or approaching a floor

promises to serve some buttons

in a given state, some lit

each has an associated floor

inside lift and at floors

top and bottom floors

orderings above and below

floor layout

basic abstractions
Elevator state

- Promises to serve some buttons
- Rising or falling
- At or approaching a floor

In a given state, some buttons
- Each has an associated floor
- Inside lift and at floors
- Top and bottom floors

Floor Layout

Above and below

Basic abstractions
floor layout
floor layout

{ open std/orders

  sig Floor {
    disj

    up, down: option Floor

    Floor
  }

  option above, below: Floor
}
{no up} { } sig Top extends Floor {}
{
  above, below: option Floor
  disj up, down: option FloorButton,
}
sig Floor
open std/orders

floor layout
{ } no down
{ } no up

**Floor**

**sig** bottom extends Floor
**sig** top extends Floor

{ above, below: option
  down: option
  up: option
}

**sig** Floor

**open** std/orders

Floor Layout
{ open std/orders
 sig Floor {
   disj up, down:
   option Floor
   above, below:
   prev = below
   next = above
   last = Top
   first = Bottom
 }

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

 sig Top extends Floor {} {no up}

 sig Bottom extends Floor {} {no down}

 sig Floor

 open std/orders

 Floor Layout
Floor layout

use ordering axioms from standard library

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

sig Bottom extends Floor {} {no down}

sig Top extends Floor {} {no up}

sig Floor extends

floor

open sig order

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

sig Floor

open sig order
Floor layout

{ floor | floor.last = bottom, floor.first = top, floor.prev = below, floor.next = above } fact layout

{ sig Floor extends { no down } }

{ sig Floor extends { no up } } { sig Floor } { sig Floor } { sig Floor } { sig Floor } { sig Floor }

 std/orders

use ordering axioms from

standard library

analyze will place buttons demontically

don't require buttons on all floors
lifts
{ buttons: set liftButton
button: floor ?-? liftButton,
} sig lift

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

\{ 
  buttons: set LiftButton
\} sig Lifts
buttons

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

buttons
buttons

{ } extends Button

{ disj sig FloorButton extends Button {lift: Lift} }
{} part sig UpButton, DownButton extends FloorButton

{} disj sig FloorButton extends Button

{} disj sig LiftButton extends Button

{} disj sig Button

buttons


```
{ buttons

    UpButton = Floor.up

    all p: Lift, p.buttons = p.button
    { some f: Floor | f->p in p.button

    Lift = Lift.button + up + down

    } facts

    ButtonDefinitions

    \[]

    part sig UpButton, DownButton extends FloorButton

    \[]

    disj sig FloorButton extends Button

    \[]

    disj sig LiftButton extends Button

    \[]

    sig Button

    \[]

```
define classes of button; redundant but convenient

define class of button:

...
sample layout

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

run showLayout
fun showLayout
    { some Lift.buttons

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

} collect together relations that change

declaring state

sys stem state
declaring state

collect together relations that change

declearing state

outstanding requests
system state

declaring state

Collect together relations that change
outstanding requests

Lift directions

{ promises: Lift -> FloorButton
at, approaching: Lift ->? Floor,
part: rising, falling: set Lift,
set: set button, }

.sig State
system state

{ promises: lift -> FloorButton
part directions: lift -> Floor,
part status: taking; See lift,
part status: door;

} State

outstanding requests

Lift directions

Lift positions

outstanding requests

declaring state

> collect together relations that change
system state

- outstanding requests
- lift directions
- lift positions
- promises: many to many

sig State { lit: set Button, part: rising, falling: set Lift, promises: Lift -> FloorButton }
Physical constraints on lift state
physical constraints on lift state

fun LiftPosition (s: State) {
    all p: Lift | with s {   
        one (at + approaching)[p] & Top,
        no at[p] & Top,
        no at[p] & Top,
        <= p in rising =>
        no approaching[p] & Top,
        no approaching[p] & Top,
        no approaching[p] & Top,
        <= p in rising =>
        no at[p] & Bottom,
        no at[p] & Bottom,
        no at[p] & Bottom,
        <= p in rising =>
        one (at + approaching)[p]
    }
}
fun LiftPosition (s: State) {
    all p: Lift | lift -? Floor,
    one (at + approaching: Lift -? Floor) =>
    no at[p] & Top, no at[p] & Bottom,
    no at[p] & Top, no at[p] & Bottom,
    no approaching[p] & Top, no approaching[p] & Bottom,
    no approaching[p] & Top, no approaching[p] & Bottom,
    no at[p] & Top, no at[p] & Bottom,
    no at[p] & Top, no at[p] & Bottom,
    no approaching[p] & Top, no approaching[p] & Bottom,
    no approaching[p] & Top, no approaching[p] & Bottom,
    no at[p] & Top, no at[p] & Bottom,
    no at[p] & Top, no at[p] & Bottom,
    no approaching[p] & Top, no approaching[p] & Bottom,
    no approaching[p] & Top, no approaching[p] & Bottom,
    no at[p] & Top, no at[p] & Bottom,
    no at[p] & Top, no at[p] & Bottom,
    no approaching[p] & Top, no approaching[p] & Bottom,
    no approaching[p] & Top, no approaching[p] & Bottom,
    no at[p] & Top, no at[p] & Bottom,
    no at[p] & Top, no at[p] & Bottom,
    no approaching[p] & Top, no approaching[p] & Bottom,
    no approaching[p] & Top, no approaching[p] & Bottom,
    no at[p] & Top, no at[p] & Bottom,
    no at[p] & Top, no at[p] & Bottom,
    no approaching[p] & Top, no approaching[p] & Bottom,
    no approaching[p] & Top, no approaching[p] & Bottom,
    no at[p] & Top, no at[p] & Bottom,
    no at[p] & Top, no at[p] & Bottom,
    no approaching[p] & Top, no approaching[p] & Bottom,
    no approaching[p] & Top, no approaching[p] & Bottom,
    no at[p] & Top, no at[p] & Bottom,
fun LiftPosition (s: State) {
  all p: Lift =>
  no (at[p] & top, no at[p] & bottom)
  in rising ⇐
  no approaching[p] & top,
  no approaching[p] & bottom,
  no at[p] & top,
  no at[p] & bottom
}
fun LiftPosition (s: State) {
  all p | with s {
    one (at + approaching)[p] 
    no (at & approaching)[p] p
    in rising => no approaching[p] & Bottom, 
    no at[p] & Top, 
    no approaching[p] & Top, 
    no approaching[p] & Bottom, 
    no at[p] & Top, 
    no at[p] & Bottom }
  }

part LiftPosition, falling: set Lift 
    at, approaching: Lift -> Floor 
}
physical constraints on lift state

fun LiftPosition (s: State) {
    all p : Lift with s
    LiftPosition (s: State)
}

part LiftPosition, falling: set Lift...

{ ... }

dig State

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

{  at + approaching | p, no (at & approaching) | p
  in rising =>
    no approaching | p & Bottom,
    no approaching | p & Top
  in rising =>
    no at | p & Top,
    no at | p & Bottom

no rising on approaching to bottom

Lift is not at and approaching

Lift is at or approaching one floor

no at [p] & bottom
no at [p] & Top

Physical constraints on Lift state

```text
part

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

fun LiftPosition (s: State) {
  all p: Lift | lift is at or approaching one floor
  at p =>
    no at & approaching & Top,
    no at & Bottom,
  all p: Lift | lift is at or approaching one floor
  approaching p =>
    no at & Top,
    no at & Bottom,
  all p: Lift | lift is at or approaching one floor
  rising p =>
    no approaching & Top,
    no approaching & Bottom
}
```
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 s.rising = p & s.rising
    then above
    else below
}

fun LiftMotion (s, s': State) {
    all p: Lift
    all p: Lift
    LiftMotion (s', s: State)
    { result = if p in s.rising then above else below
    }
}

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

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

fun LiftMotion (s, s': State) {
result = if p in s.rising
    then above s
    else below s
}

fun nextFloor (s: State, p: Lift): Floor -> Floor

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
  s.nextFloor(p)
}

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

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

no dir change except at floor

result = if p in s.rising then above else below

physical constraints on lift motion

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

fun LiftMotion (s, s': State) {
                s.approaching[p] = s'.approaching[p] &
                s.nextFloor[p] = s'.nextFloor[p] &
  no dir change except at floor
  no dir change except at floor
  no dir change except at floor
  no dir change except at floor
  floor at after is floor at or approaching before
  floor at after is floor at or approaching before
  floor at after is floor at or approaching before
  floor at after is floor at or approaching before
  then LiftMotion (s, s': State) }

result = if p in s.rising then above else below

fun nextFloor (s: State, p: Lift) -> Floor

physical constraints on lift motion
sample transition
sample transition

run NiceMotion for 3 but 2 State

\{ s.at \neq s'.at \\
\land NiceMotion (s, s') \land \overline{NicePosition (s)} \land \overline{NicePosition (s')}
\} (s', s: State)

run NiceMotion (s', s: State)
run NiceMotion for 3 but 2 State

{s'at = s'at

{Motion (s,s') & LiftPosition (s) & LiftPosition (s')

run NiceMotion (s,s'; State)

sample transition
run NiceMotion for 3 but 2 State

s.at = s'.at

LiftMotion (s, s') & LiftPosition (s) & LiftPosition (s')

run NiceMotion (s, s) State

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

button update
denying service
denying service
Towards (s, p, f) {
    let next = nextFloor(s, p, f)
    in s.at[p].next + s.approaching[p].next
}

fun Denies (s, s', p, b) {
    let f = b.floor {
        Towards (s, p, f) not
        Towards (s', p, f)
    }
}

fun Denies (s, s', p, b) {
    let f = b.floor {
        Towards (s, p, f) not
        towards (s', p, f)
    }
    in s.at[p].next + s.approaching[p].next
    | (p, s) = nextFloor(s', p)
}

denying service
a policy
Policy (s, s') in State { a policy

AvoidStops (s, s')

NoStuckLift (s, s')

Towards (s, p, p', floor) and some p in Lift

Serve(s, s', q, q')

FloorButton, p in Lift

Denies (s, s', p, q')

all p in Lift & FloorButton, p in Lift

Denies (s, s', p, q')

no p in Lift, p in Buttons & Lift

Policy (s, s') in State

denies (s, s')

A

B

C

D

E
fun policy (s : State)

  { 
    avoids_stops (s, s')
    avoids_kick (s, s')
    and some p : s.lit towards (s, p'.floor)
    or (p in s.promises[lift])
    (some q : lift serves(s, q, p')
      all b : s.lit & floor_button, p : lift
      denies (s, s', p, q) =>
      no p in s.promises[lift] & some b : s.lit & lift_button (s, s', p)
    )
  }

Don’t deny lift buttons
Policy \( (s, s') \) { no p: Lift, b: p.buttons & s.lit \implies \text{Denies} (s, s', p, b) all b: s.lit & FloorButton, p: Lift \implies \text{Denies} (b \in s'.promises[Lift]) and some b: Lift \implies \text{Towards} (s, b, s'.floor) or (b \in s'.promises[Lift]) and some g: Lift \implies \text{Severed} (s, g, s'.floor) \\
\implies \text{All or Still in Front of Floor, \& Lift \text{ Doesn't} (s, s', p, b) } \geq 0 \implies \text{Stops, \& Elevator \text{ Doesn't} (s, s', p, b) } } \}

fun Policy (s, s': State)
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

```scala
fun Trans (s, s': State) {
  LiftPosition (s)
  LiftPosition (s')
  LiftMotion (s, s')
  Policy (s, s')
  Button | ButtonUpdate (s, s', press)
}
```
sample denial
fun `ShowPolicy` for 3 but 2 State, 2 Lift, 2 Button

{ some 8 promises & some 8 promises |
  no promises & promises (s', p')
  Trans (s, s') |
  some b : s.lit & FloorButton, p : Lift | Denies (s, s', p')
}

fun `ShowPolicy` (s, s' : State) sample denial
**Sample denial**

```plaintext
run ShowPolicy: 3 but 2 State, 2 Llit, 2 Button

no s.promises & s.promises

some b: s.Llit & s.FloorButton

run ShowPolicy (s, s': State)
```

Trans (s, s')
ShowPolicy (s, s') {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

trans (s, s')

Sample denial
fun Trace() {

    let s' = ord[State].next[s] |
    all s : State - ord[State].last |
    Init (ord[State].first) |

    Trans (s, s') |

    no s.promises |
    no s.lift.floor & s.lift[lift] |

    } (Init (s : State))

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

  all s: State - Ord[State].last
      no promises
      no s.lit.floor & s.at[Lift]
      Init (s::State)
traces

fun Trace () { Init (Ord[State].first) { all s: State - Ord[State].last | let s' = Ord[State].next[s] | Transition (s,s') } } { no promises

fun Init (s: State) { traces

asserting eventual service
asserting eventual service
counterexample!
counterexample!
assert EventuallyServed {
    let start = ord[state].first
    in
    all b: start.lit | b \in s'.lit
    in
    some s: OrdNexts (start) | b !
    in
    s'.lit
}

counterexample!
assert EventuallyServed { Trace () and some (lit) => no service without lifts

23

counterexample!
model structure

- Trace definition
- Transition relation
- Eventuality assertion

- Policy description
- Button update
- Behaviour definitions (e.g., denying)

- Signatures
- State
- Floor & button

- Physical constraints on lift state
- Physical constraints on lift motion

- Design
- Physics
incremental development

write minimal model

pick analysis

generate instances

loosen model

extend model or stop

check property

tighten model

some

none

some

none
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

are they just cultural?

what are they?

a better way to allow load balancing?

replacing promises

>