Alloy

Daniel Jackson
MIT Lab for Computer Science
ETAPS, April 10, 2002

joint work with:
Ilya Shlyakhter, Manu Sridharan, Sarfraz Khurshid
Brian Lin, Jesse Pavel, Mana Taghdiri,
Mandana Vaziri, Hoeteck Wee
non supporté

H: 42.5Hz    V:85.4Hz
didn’t you bring a hardcopy backup? fool!

non supporte
H: 42.5Hz  V:85.4Hz
motivations
motivations

‘software model checking’
› system implemented in software?
› infinitely many states?
› handle code directly?
motivations

‘software model checking’
 › system implemented in software?
 › infinitely many states?
 › handle code directly?

my focus
 › attack essence of software design
    -- structures and how they change
 › incremental and partial modelling
 › automatic, interactive analysis
motivations

‘software model checking’
› system implemented in software?
› infinitely many states?
› handle code directly?

my focus
› attack essence of software design
   -- structures and how they change
› incremental and partial modelling
› automatic, interactive analysis

attempt to get benefits of
› SMV: automatic analysis
› Z: expression of structure
motivations

‘software model checking’
› system implemented in software?
› infinitely many states?
› handle code directly?

my focus
› attack essence of software design
  -- structures and how they change
› incremental and partial modelling
› automatic, interactive analysis

attempt to get benefits of
› SMV: automatic analysis
› Z: expression of structure
motivations

‘software model checking’
› system implemented in software?
› infinitely many states?
› handle code directly?

my focus
› attack essence of software design
  -- structures and how they change
› incremental and partial modelling
› automatic, interactive analysis

attempt to get benefits of
› SMV: automatic analysis
› Z: expression of structure
the challenge
the challenge

language must support
  › complex data structures
  › declarative specification
    partiality, separation of concerns
the challenge

language must support
  › complex data structures
  › declarative specification
      partiality, separation of concerns

analysis must be
  › fully automatic
  › interactive performance
  › easy to interpret output
key ideas: foundations
key ideas: foundations

language is first order logic + relations
› all data structures encoded as relations
› hierarchy with higher-arity relations
key ideas: foundations

language is first order logic + relations
› all data structures encoded as relations
› hierarchy with higher-arity relations

analysis is model finding
› make decidable by bounding universe
› ‘small scope hypothesis’
key ideas: foundations

language is first order logic + relations
  › all data structures encoded as relations
  › hierarchy with higher-arity relations

analysis is model finding
  › make decidable by bounding universe
  › ‘small scope hypothesis’

exploit SAT technology
  › analyzer is a compiler
  › symmetry breaking, skolemization, sharing, etc
  › pluggable backend
key ideas: pragmatics
key ideas: pragmatics

syntax
  › ASCII based
  › prefer existing conventions
key ideas: pragmatics

syntax
  › ASCII based
  › prefer existing conventions

semantics
  › relations only: no scalars, sets or tuples
    a represented as \{a\}
    \((a,b)\) represented as \{(a,b)\}
  › gives simpler syntax
  › no complications from partial functions
    undefined, null, maybe, non-denoting terms
key ideas: pragmatics

syntax
› ASCII based
› prefer existing conventions

semantics
› relations only: no scalars, sets or tuples
  a represented as \{a\}
  (a,b) represented as \{(a,b)\}
› gives simpler syntax
› no complications from partial functions
  undefined, null, maybe, non-denoting terms

visualization
› customizable, no built in notion of state, eg.
what’s been done?
what's been done?

sample applications
› Chord peer-to-peer lookup (Wee)
› Intentional Naming (Khurshid)
› Key management (Taghdiri)
› Microsoft COM (Sullivan)
› Classic distributed algorithms (Shlyakhter)
› Firewire leader election (Jackson)
› Red-black tree invariants (Vaziri)
› RM-ODP meta modelling (EPFL)
› Role-based access control (BBN)
what’s been done?

sample applications
› Chord peer-to-peer lookup (Wee)
› Intentional Naming (Khurshid)
› Key management (Taghdiri)
› Microsoft COM (Sullivan)
› Classic distributed algorithms (Shlyakhter)
› Firewire leader election (Jackson)
› Red-black tree invariants (Vaziri)
› RM-ODP meta modelling (EPFL)
› Role-based access control (BBN)

taught in courses at
› CMU, Waterloo, Wisconsin, Rochester, Kansas State, Irvine, Georgia Tech, Queen’s, Michigan State, Imperial, Colorado State, Twente, WPI, MIT
outline of rest of talk
outline of rest of talk

elevator example
  › translating a fragment
  › expressing constraints
  › trace-based analysis
outline of rest of talk

elevator example
  › translating a fragment
  › expressing constraints
  › trace-based analysis

bounding traces
  › how long a trace?
outline of rest of talk

elevator example
  › translating a fragment
  › expressing constraints
  › trace-based analysis

bounding traces
  › how long a trace?

application to code
  › analysis, testing
outline of rest of talk

elevator example
  › translating a fragment
  › expressing constraints
  › trace-based analysis

bounding traces
  › how long a trace?

application to code
  › analysis, testing

related work & conclusions
example: elevator policy
example: elevator policy

challenge

› specify a policy for scheduling elevators
example: elevator policy

challenge
› specify a policy for scheduling elevators

tight enough
› all requests eventually served
› don’t skip request from inside lift
example: elevator policy

challenge
› specify a policy for scheduling elevators

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
approach: promises
approach: promises

deny request
  › ‘skipping’: don’t stop at floor
  › ‘bouncing’: double back before floor
approach: promises

deny request
 › ‘skipping’: don’t stop at floor
 › ‘bouncing’: double back before floor

policy
 › a lift can’t deny a request from inside
 › if a lift denies a floor request
    some lift promises to take it later
approach: promises

deny request
  › ‘skipping’: don’t stop at floor
  › ‘bouncing’: double back before floor

policy
  › a lift can’t deny a request from inside
  › if a lift denies a floor request
      some lift promises to take it later

freedoms
  › divide requests amongst lifts
  › postpone decision until first skip or bounce
  › unlike ‘closest serves’, can balance load
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
language elements
language elements

relations

sig State {at: Lift ->? Floor}
declares relation at with values like {\(s_0, p_0, f_0\), \(s_1, p_0, f_1\)}
language elements

relations

    sig State {at: Lift ->? Floor}
    declares relation at with values like \{(s0,p0,f0),(s1,p0,f1)\}

operators

    + & - .
    union, intersection, difference, join

    s.at
    the lift/floor mapping for state s

    p.(s.at), s.at[p]
    the floor of lift p in state s

    at = \{(s0,p0,f0),(s1,p0,f1)\} , s = \{(s1)\}, p = \{(p0)\}
    s.at = \{(p0,f1)\}, s.at[p] = \{(f1)\}
language elements

relations

sig State {at: Lift ->? Floor}
declares relation at with values like \{(s0,p0,f0),(s1,p0,f1)\}

operators

+ & - .

union, intersection, difference, join

s.at

the lift/floor mapping for state s

p.(s.at), s.at[p]

the floor of lift p in state s

\[\begin{align*}
\text{at} &= \{(s0,p0,f0),(s1,p0,f1)\}, \quad s = \{(s1)\}, \quad p = \{(p0)\} \\
\text{s.at} &= \{(p0,f1)\}, \quad \text{s.at}[p] = \{(f1)\}
\end{align*}\]

formulas

\begin{align*}
in \\
s.at[p] \text{ in } f
\end{align*}

means subset

if p is at a floor in state s, that floor is f
example
example

sig Floor {above, below: option Floor}
-- above, below map each floor to at most one floor
example

sig Floor {above, below: option Floor}
-- above, below map each floor to at most one floor

sig Lift {} -- introduces a set, no relations
example

sig Floor {above, below: option Floor}
-- above, below map each floor to at most one floor

sig Lift {} -- introduces a set, no relations

sig State {at, approaching: Lift ->? Floor}
-- at, approaching map each state to a partial function
example

sig Floor {above, below: option Floor}
-- above, below map each floor to at most one floor

sig Lift {} -- introduces a set, no relations

sig State {at, approaching: Lift ->? Floor}
-- at, approaching map each state to a partial function

fact {all s: State, p: Lift | one s.(at+approaching)[p]}
-- global constraint: in a state, lift is at or approaching one floor
example

sig Floor {above, below: option Floor}
-- above, below map each floor to at most one floor

sig Lift {} -- introduces a set, no relations

sig State {at, approaching: Lift ->? Floor}
-- at, approaching map each state to a partial function

fact {all s: State, p: Lift | one s.(at+approaching)[p]}
-- global constraint: in a state, lift is at or approaching one floor

fun show () {Floor in State.at[Lift]}
-- invocable constraint: each floor has a lift at it in some state
example

sig Floor {above, below: option Floor}
-- above, below map each floor to at most one floor

sig Lift {} -- introduces a set, no relations

sig State {at, approaching: Lift ->? Floor}
-- at, approaching map each state to a partial function

fact {all s: State, p: Lift | one s.(at+approaching)[p]}
-- global constraint: in a state, lift is at or approaching one floor

fun show () {Floor in State.at[Lift]}
-- invocable constraint: each floor has a lift at it in some state

run show for 2 -- find instance with 2 states, lifts, floors
translation
translation

sig Floor {above, below: option Floor}
-- allocate boolean variables Floor[i], above[i,j], below[i,j]
-- interpretation: above[i,j] is true if jth floor is above ith floor
-- ranges of i, j etc determined by scope: for 2 floors, i, j ∈ 0..1
translation

sig Floor {above, below: option Floor}
-- allocate boolean variables Floor[i], above[i,j], below[i,j]
-- interpretation: above[i,j] is true if jth floor is above ith floor
-- ranges of i, j etc determined by scope: for 2 floors, i, j ∈ 0..1

sig Lift {} -- allocate Lift[i]
translation

sig Floor {above, below: option Floor}
-- allocate boolean variables Floor[i], above[i,j], below[i,j]
-- interpretation: above[i,j] is true if jth floor is above ith floor
-- ranges of i, j etc determined by scope: for 2 floors, i, j ∈ 0..1

sig Lift {} -- allocate Lift[i]

sig State {at, approaching: Lift ->? Floor}
-- allocate at[i,j,k], approaching[i,j,k]
translation

sig Floor {above, below: option Floor}
-- allocate boolean variables Floor[i], above[i,j], below[i,j]
-- interpretation: above[i,j] is true if jth floor is above ith floor
-- ranges of i, j etc determined by scope: for 2 floors, i, j € 0..1

sig Lift {} -- allocate Lift[i]

sig State {at, approaching: Lift ->? Floor}
-- allocate at[i,j,k], approaching[i,j,k]

fact {all s: State, p: Lift | one s.(at+approaching)[p]}
fun show () {Floor in State.at[Lift]}
-- create formula ∀k . Floor[k] ⇒ ∃i,j . at[i,j,k] ∧ State[i] ∧ Lift[j]
translation

sig Floor {above, below: option Floor}
-- allocate boolean variables Floor[i], above[i,j], below[i,j]
-- interpretation: above[i,j] is true if jth floor is above ith floor
-- ranges of i, j etc determined by scope: for 2 floors, i, j ∈ 0..1

sig Lift {} -- allocate Lift[i]

sig State {at, approaching: Lift ->? Floor}
-- allocate at[i,j,k], approaching[i,j,k]

fact {all s: State, p: Lift | one s.(at+approaching)[p]}
fun show () {Floor in State.at[Lift]}
-- create formula ∀k. Floor[k] ⇒ ∃i,j. at[i,j,k] ∧ State[i] ∧ Lift[j]

run show for 2 -- solve formula
an instance generated by the analyzer
an instance generated by the analyzer
an instance generated by the analyzer

select projection for type
projection onto Lift
projection onto State
process
user writes model and selects command

```plaintext
*mobile lifes
open std/ord

sig Floor { up, down: option FloorButton, above, below: option Floor} (no up & down)
sig Top extends Floor (){no up}
sig Bottom extends Floor (){no down}
sig Lift {
  button: Floor ?-> LiftButton,
  buttons: set LiftButton
}
sig Button (Floor: Floor)
disj sig LiftButton extends Button (){lift: Lift}
disj sig FloorButton extends Button ()
port sig UpButton, DownButton extends FloorButton ()

fact Layout { 
Or(Floor).next = above
Or(Floor).prev = below
Or(Floor).last = Top
Or(Floor).first = Bottom
}

sig State {
  lift, outstanding: set Button, 
  pair: rising, falling: set Lifts, 
  at, approaching: Lift ?-> LiftButton, 
  premises: Lift -> LiftButton
}
```
process

user writes model and selects command

Alloy Analyzer translates command to boolean formula

```alloy
module lifes
open std/ord

sig Floor {
  up, down: option FloorButton,
  above, below: option Floor
} (no up & down)

sig Top extends Floor () (no up)
sig Bottom extends Floor () (no down)

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

sig Button (Floor: Floor)
disj sig LiftButton extends Button () (lift: Lift)
disj sig FloorButton extends Button ()
port sig upButton, downButton extends FloorButton ()

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

sig State {
  lift, outstanding: set Button,
  pos rising, falling: set Floors,
  at, approaching: Lift ->? Floor,
  premises: lift -> FloorButton
}
```

```plaintext
maxindep 12
pcnf 114 188
1 1 -4 0
17 2 -7 0
18 3 -10 0
15 -16 0
15 -17 0
15 -18 0
20 1 5 0
21 2 -8 0
22 3 -11 0
```
process

user writes model and selects command

Alloy Analyzer translates command to boolean formula

SAT solver finds boolean solution
process

user writes model and selects command

Alloy Analyzer translates command to boolean formula

Alloy Analyzer translates boolean solution to relational

SAT solver finds boolean solution

module lifs
open std/ord

sig Floor {
  up, down: option FloorButton,
  above, below: option Floor
} (no up & down)

sig Top extends Floor () (no up)

sig Button extends Floor () (no down)

sig Lift {
  button: Floor => LiftButton,
  buttons: set LiftButton
}

sig Button (Floor: Floor)

disj sig LiftButton extends Button () (lift: Lift)

disj sig FloorButton extends Button ()

port sig UpButton, DownButton extends FloorButton ()

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

sig State {
  lift, outstanding: set Button,
  pare rising, falling: set Lift,
  at, approaching: Lift => Floor,
  promises: Lift => FloorButton
}

c max indep 12
p cnf 114 188
16 1 4 0
17 2 7 0
18 3 10 0
15 16 0
15 17 0
15 18 0
20 1 5 0
21 2 8 0
22 3 11 0
user writes model and selects command

Alloy Analyzer translates command to boolean formula

process

Alloy Analyzer translates boolean solution to relational

Alloy Analyzer creates custom visualization

SAT solver finds boolean solution

Alloy Analyzer translates model and selects command

Alloy Analyzer translates boolean formula

module lifes
open std/ord

sig Floor {
up, down, option FloorButton, above, below: option Floor
}(no up & down)
sig Top extends Floor ()(no up)
sig Button extends Floor ()(no down)
sig Lift {
button: Floor -> LiftButton, buttons: set LiftButton
}
sig Button (Floor: Floor)
disj sig LiftButton extends Button ()(lift: Lift)
disj sig FloorButton extends Button ()
port sig upButton, DownButton extends FloorButton ()

fact Layout {
Ord(Floor).next = above
Ord(Floor).prev = below
Ord(Floor).last = Top
Ord(Floor).first = Bottom
}
sig State {
lift, outstanding: set Button, permit rising, falling: set Lifts,
at, approaching: Lift -> Floor, promises: Lift -> FloorButton
}

c maxindep 12
p cnf 114 188
16 1 -4 0
17 2 -7 0
18 3 -10 0
15 -16 0
15 -17 0
15 -18 0
20 1 -5 0
21 2 -8 0
22 3 -11 0
constraints
constraints

lift physics & hardware
› can’t be at and approaching a floor
› can’t jump from floor to floor
› can’t change direction between floors
constraints

lift physics & hardware
 › can’t be at and approaching a floor
 › can’t jump from floor to floor
 › can’t change direction between floors

policy
 › can’t skip a request from inside the lift
 › buttons reset when requests serviced
**constraints**

**lift physics & hardware**
- can’t be at and approaching a floor
- can’t jump from floor to floor
- can’t change direction between floors

**policy**
- can’t skip a request from inside the lift
- buttons reset when requests serviced

**analyses**
- generate samples of states, steps, traces
- show policy implies desired properties (eg, no starvation)
static environmental constraints
static environmental constraints

sig Bottom extends Floor {}
static environmental constraints

sig Bottom extends Floor {}

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

static environmental constraints

sig Bottom extends Floor {}

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

fun LiftPosition (s: State) {
    all p: Lift {
        -- lift is not at and approaching same floor
        no s.at[p] & s.approaching[p]
        -- can't be approaching the bottom floor when rising
        p in s.rising => s.approaching[p] != Bottom
    ...
}
}
**static environmental constraints**

```plaintext
sig Bottom extends Floor {}

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

fun LiftPosition (s: State) {
    all p: Lift {
        -- lift is not at and approaching same floor
        no s.at[p] & s.approaching[p]
        -- can't be approaching the bottom floor when rising
        p in s.rising => s.approaching[p] != Bottom
    }
}
```

function: an ‘invocable’ constraint
dynamic environmental constraints
dynamic environmental constraints

fun LiftMotion (s, s': State) {
    all p: Lift {
        -- if at a floor after, was at or approaching that floor before
        s'.at[p] in s.(at + approaching)[p]
        ...
    }
}
dynamic environmental constraints

fun LiftMotion (s, s': State) {
    all p: Lift {
        -- if at a floor after, was at or approaching that floor before
        s'.at[p] in s.(at + approaching)[p]
        ...
    }
}

terse relational operators
s'.at[p] in s.(at + approaching)[p]
all f: Floor | f = s'.at[p] => f = s.at[p] or f = s.approaching[p]
**dynamic environmental constraints**

fun LiftMotion (s, s': State) {
    all p: Lift {
        -- if at a floor after, was at or approaching that floor before
        s'.at[p] in s.(at + approaching)[p]
        ...
    }
}

terse relational operators
s'.at[p] in s.(at + approaching)[p]
all f: Floor | f = s’.at[p] => f = s.at[p] or f = s.approaching[p]
policy: defining denial
policy: defining denial

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

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

fun Towards (s: State, p: Lift, f: Floor) {
    -- p is going towards serving floor f
    let next = nextFloor(s,p) |
        f in s.at[p].^next + s.approaching[p].*next
}
policy: defining denial

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

fun Towards (s: State, p: Lift, f: Floor) {
    -- p is going towards serving floor f
    let next = nextFloor(s,p) |
    f in s.at[p].^next + s.approaching[p].^next
}

fun Denies (s, s': State, p: Lift, b: Button) {
    -- p was going to serve b, but is no longer
    let f = b.floor |
    Towards (s,p,f) and not Towards (s',p,f) and !Serves (s,s',p,b)
}
policy: defining denial

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

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

fun Denies (s, s': State, p: Lift, b: Button) {
  -- p was going to serve b, but is no longer
  let f = b.floor |
  Towards (s,p,f) and not Towards (s',p,f) and !Serves (s,s',p,b)
}
policy
policy

sig State {
    lit: set Button,
    promises: Lift → Button, ...
}

policy

sig State {
    lit: set Button,
    promises: Lift -> Button, ...
}

fun Policy (s, s': State) {
    -- a lift can't deny a promise or a request from inside the lift
    no p: Lift, b: s.promises[p] + p.buttons & s.lit | Denies (s,s',p,b)
    -- if a lift denies a request some lift serves it or promises to
    all b: s.lit & FloorButton – s.promises[Lift], p: Lift | 
       Denies (s,s',p,b) =>
       (some q: Lift | Serves(s,s',q,b)) or b in s'.promises[Lift]
    ...
}
policy

sig State {
    lit: set Button,
    promises: Lift -> Button, ...
}

fun Policy (s, s': State) {
    -- a lift can't deny a promise or a request from inside the lift
    no p: Lift, b: s.promises[p] + p.buttons & s.lit | Denies (s,s',p,b)
    -- if a lift denies a request some lift serves it or promises to
    all b: s.lit & FloorButton – s.promises[Lift], p: Lift |
    Denies (s,s',p,b) =>
        (some q: Lift | Serves(s,s',q,b)) or b in s'.promises[Lift]
    ...
}{
    non-deterministic
}
putting things together
fun Trans (s, s': State) {
    -- the before and after positions and the motion are legal
    LiftPosition (s) and LiftPosition (s') and LiftMotion (s,s')
    -- the policy is satisfied
    Policy (s,s’)
    -- the buttons are reset appropriately
    some press: set Button | ButtonUpdate (s,s',press)
}
animating denial
animating 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 2 but 3 Floor
sample denial
sample denial
sample denial

the denying lift
sample denial

the denying lift
the denied button
sample denial

the denying lift
the denied button
sample denial

the denying lift
the denied button

another lift promises
traces: checking starvation
fun Trace () {
    -- a state is related to its successor by the transition relation
    all s: State – Ord[State].last |
    let s' = Ord[State].next[s] | Trans (s,s')
}
traces: checking starvation

fun Trace () {
    -- a state is related to its successor by the transition relation
    all s: State – Ord[State].last |
    let s’ = Ord[State].next[s] | Trans (s,s’)
}

assert EventuallyServed {
    -- if the states form a trace
    Trace () =>
    -- then a button lit in the start state is eventually reset
    all b: (Ord[State].first).lit | some s’: State | b !in s’.lit
}
traces: checking starvation

fun Trace () {
  -- a state is related to its successor by the transition relation
  all s: State – Ord[State].last |
    let s' = Ord[State].next[s] | Trans (s,s')
}

assert EventuallyServed {
  -- if the states form a trace
  Trace () =>
  -- then a button lit in the start state is eventually reset
    all b: (Ord[State].first).lit | some s': State | b !in s'.lit
}

check EventuallyServed for 3 Lift, 3 Button, 3 Floor, 8 State
counterexample!
counterexample!
counterexample!

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

Lift_1 promises...

Lift_0 (rising)

Button_0
(floor: Floor_0)

Floor_2

approaching

Floor_1

below

Floor_0
up: Button_0(b)

Lift_1 (rising)

Floor_0
up: Button_0(b)

Lift_0 (falling)

Floor_2

below

Floor_1

below
another...

Lift_1 promises

Lift_1 turns
another...

promise passes from Lift_1 to Lift_0!
another...

Lift_1 promises

Lift_1 turns

promise passes from Lift_1 to Lift_0!

Lift_0 drops promise

Lift_0 drops promise
what you’ve seen
what you’ve seen

simple logic, complex system
› relations for all structuring
  buttons to lifts, components to states, states to successors
› declarative style
  separation of concerns by conjunction
› relational operators
  succinct, idioms easy to grasp
students did lift problem as homework after 3 lectures
what you’ve seen

simple logic, complex system
› relations for all structuring
   buttons to lifts, components to states, states to successors
› declarative style
   separation of concerns by conjunction
› relational operators
   succinct, idioms easy to grasp
students did lift problem as homework after 3 lectures

one analysis -- model finding
› for simulation and consequence checking
› (for checking refactoring)
when is a trace long enough?
when is a trace long enough?

for safety properties, check all traces
  › but how long? ie, what is scope of State?
when is a trace long enough?

for safety properties, check all traces
  › but how long? ie, what is scope of State?

idea: bound the diameter
  › if all states reached in path ≤ k
  › enough to consider only traces ≤ k
when is a trace long enough?

for safety properties, check all traces
  › but how long? ie, what is scope of State?

idea: bound the diameter
  › if all states reached in path \( \leq k \)
  › enough to consider only traces \( \leq k \)

strategy
  › ask for loopless trace of length \( k+1 \)
    if none, then \( k \) is a bound
  › tighter bounds possible: eg, no shortcuts
when is a trace long enough?

for safety properties, check all traces
   › but how long? ie, what is scope of State?

idea: bound the diameter
   › if all states reached in path \( \leq k \)
   › enough to consider only traces \( \leq k \)

strategy
   › ask for loopless trace of length \( k+1 \)
     if none, then \( k \) is a bound
   › tighter bounds possible: eg, no shortcuts

like bounded model checking
   › but can express conditions directly
when is a trace long enough?

for safety properties, check all traces
› but how long? ie, what is scope of State?

idea: bound the diameter
› if all states reached in path \( \leq k \)
› enough to consider only traces \( \leq k \)

strategy
› ask for loopless trace of length \( k+1 \)
   if none, then \( k \) is a bound
› tighter bounds possible: eg, no shortcuts

like bounded model checking
› but can express conditions directly

\[
\text{diameter} = 1
\]
\[
\text{max loopless} = 1
\]
when is a trace long enough?

for safety properties, check all traces
- but how long? ie, what is scope of State?

idea: bound the diameter
- if all states reached in path \( \leq k \)
- enough to consider only traces \( \leq k \)

strategy
- ask for loopless trace of length \( k+1 \)
  - if none, then \( k \) is a bound
- tighter bounds possible: eg, no shortcuts

like bounded model checking
- but can express conditions directly

\( \text{diameter} = 1 \)
\( \text{max loopless} = 1 \)
\( \text{diameter} = 1 \)
\( \text{max loopless} = 5 \)
applications to code
applications to code

Alloy Annotation Language
› mutation, nulls, dynamic dispatch
applications to code

Alloy Annotation Language
› mutation, nulls, dynamic dispatch

test suite generation
› ask analyzer for instances of rep invariant
› can test one operation of an abstract type
› symmetry breaking gives good coverage
applications to code

Alloy Annotation Language
  › mutation, nulls, dynamic dispatch

test suite generation
  › ask analyzer for instances of rep invariant
  › can test one operation of an abstract type
  › symmetry breaking gives good coverage

code analysis
  › translate body of method into Alloy constraint
  › assert that body implies specification
  › analyzer gives counterexamples heap traces
applications to code

Alloy Annotation Language
› mutation, nulls, dynamic dispatch

test suite generation
› ask analyzer for instances of rep invariant
› can test one operation of an abstract type
› symmetry breaking gives good coverage

code analysis
› translate body of method into Alloy constraint
› assert that body implies specification
› analyzer gives counterexamples heap traces

example: red-black trees
all x,y: Leaf | #(x.~*children & Black) = #(y.~*children & Black)
related work: UML
related work: UML

Object Constraint Language (IBM)
› not fully declarative
› pre/post built-in
› Smalltalk-like syntax for quantifiers
related work: UML

Object Constraint Language (IBM)
› not fully declarative
› pre/post built-in
› Smalltalk-like syntax for quantifiers

not designed for analysis
› ‘tool just like Alloy’s, but with Joe User in place of Chaff’
related work: UML

Object Constraint Language (IBM)
› not fully declarative
› pre/post built-in
› Smalltalk-like syntax for quantifiers

not designed for analysis
› ‘tool just like Alloy’s, but with Joe User in place of Chaff’

many researchers working on fixing it
› better to start again with something simpler?
› must we really discard traditional logic?
› is this really what industry needs?
related work: UML

Object Constraint Language (IBM)
› not fully declarative
› pre/post built-in
› Smalltalk-like syntax for quantifiers

not designed for analysis
› ‘tool just like Alloy’s, but with Joe User in place of Chaff’

many researchers working on fixing it
› better to start again with something simpler?
› must we really discard traditional logic?
› is this really what industry needs?

see UML metamodel in Alloy on sdg.lcs.mit.edu/alloy
related work: model checking
related work: model checking

only low-level datatypes
  › must encode in records, arrays
  › no transitive closure, etc
related work: model checking

only low-level datatypes
  › must encode in records, arrays
  › no transitive closure, etc

built-in communications
  › not suited for abstract schemes
  › fixed topology of processes
related work: model checking

only low-level datatypes
› must encode in records, arrays
› no transitive closure, etc

built-in communications
› not suited for abstract schemes
› fixed topology of processes

culture of model checking
› emphasizes finding showstopper flaws
› but in software, essence is incremental modelling
› keep counters, discard model or vice versa?
related work: model checking

only low-level datatypes
  › must encode in records, arrays
  › no transitive closure, etc

built-in communications
  › not suited for abstract schemes
  › fixed topology of processes

culture of model checking
  › emphasizes finding showstopper flaws
  › but in software, essence is incremental modelling
  › keep counters, discard model or vice versa?
related work: static analysis
related work: static analysis

type analyses
  › scalable, compositional, economical
  › can’t express complex structural properties
related work: static analysis

type analyses
  › scalable, compositional, economical
  › can’t express complex structural properties

proof-based techniques (eg, PCC)
  › complete: good when adversary seeds bugs (but ESC)
  › can’t check structural properties without lemmas
related work: static analysis

type analyses
› scalable, compositional, economical
› can’t express complex structural properties

proof-based techniques (eg, PCC)
› complete: good when adversary seeds bugs (but ESC)
› can’t check structural properties without lemmas

shape analyses (eg, PEGs, TVLA)
› automatic and complete for whole program
› but for modular analysis, not complete
    eg, assume arguments to procedure aren’t aliased
conclusion
conclusion

summary
› executability $\not\Rightarrow$ loss of abstraction
› analysis is more than verification
› first-order logic can be tractable
conclusion

summary
› executability ⇏ loss of abstraction
› analysis is more than verification
› first-order logic can be tractable

current challenges
› documenting idioms
› tool performance  
› design conformance
conclusion

summary
› executability $\neq$ loss of abstraction
› analysis is more than verification
› first-order logic can be tractable

current challenges
› documenting idioms
› tool performance
› design conformance

http://sdg.lcs.mit.edu/alloy
› tool downloads
› papers