alloy

Daniel Jackson
MIT Lab for Computer Science
6898: Advanced Topics in Software Design
February 11, 2002

course admin

new version of bill's notes online

schedule

- today: Alloy language
- weds: modelling idioms
  first problem set out; due 2 weeks later
- mon: holiday
- weds: peer review
- mon: JML, OCL, Z

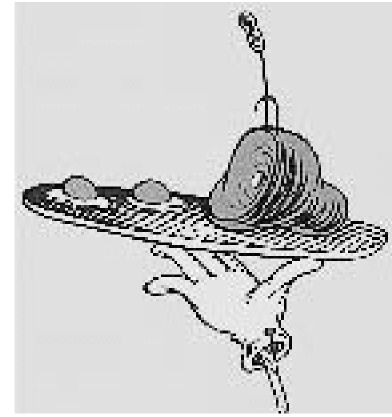
peer review

- 4 students present models for discussion

tasks

- scribe for today
- organizer for peer review
- presenters on JML, OCL, Z

software blueprints



what?

- clear abstract design
- captures just essence

why?

- fewer showstopper flaws
- major refactoring less likely
- easier coding, better performance

how?

- identify risky aspects
- develop model incrementally
- simulate & analyze as you go

alloy: a new approach

alloy
- language & analyzer designed hand-in-hand
- fully automatic checking of theorems
- simulation without test cases

language
- a flexible notation for describing structure
- static structures, or dynamic behaviours
- a logic, so declarative: incremental & implicit

analyzer
- mouse-click automation
- generates counterexamples to theorems
- based on new SAT technology

roots & influences

Z notation (Oxford, 1980-1992)

- elegant & powerful, but no automation

SMV model checker (CMU, 1989)

- 10^{100} states, but low-level & for hardware

Nitpick (Jackson & Damon, 1995)

- Z subset (no quantifiers), explicit search

Alloy (Jackson, Shlyakhter, Sridharan, 1997-2002)

- full logic with quantifiers & any-arity relations
- flexible structuring mechanisms
- SAT backend, new solvers every month

experience with alloy

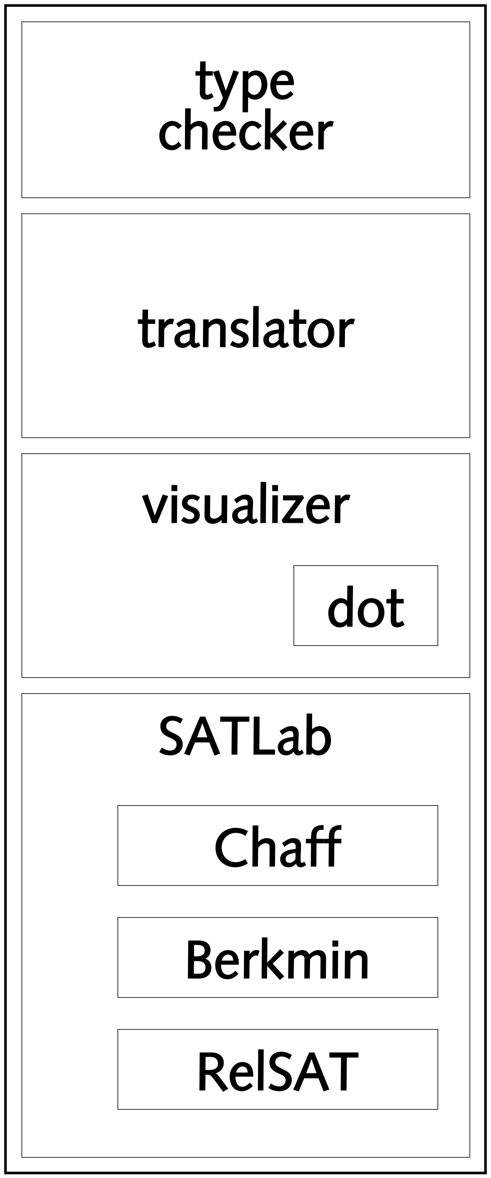
applications

› Chord peer-to-peer lookup (Wee)
› Access control (Wee)
› Intentional Naming (Khurshid)
› Microsoft COM (Sullivan)
› Classic distributed algorithms (Shlyakhter)
› Firewire leader election (Jackson)
› Red-black tree invariants (Vaziri)

taught in courses at

› CMU, Waterloo, Wisconsin, Rochester, Kansas State, Irvine, Georgia Tech, Queen's, Michigan State, Imperial, Colorado State, Twente

elements of alloy project



language design
- flexible, clean syntax, all F.O.

scheme for translation to SAT
- skolemization, grounding out
- exploiting symmetry & sharing

customizable visualization

framework for plug-in solvers
- currently Chaff & BerkMin
- decouples Alloy from SAT

alloy type system

types

- a universe of atoms, partitioned into basic types
- relational type is sequence of basic types
- sets are unary relations; scalars are singleton sets

examples

- basic types ROUTER, IP, LINK
- relations

Up: ⟨ROUTER⟩	the set of routers that's up
ip: ⟨ROUTER, IP⟩	maps router to its IP addr
from, to: ⟨LINK,ROUTER⟩	maps link to routers
table: ⟨ROUTER, IP, LINK⟩	maps router to table

relational operators

join

$p.q = \{(p_1, \ldots p_{n-1}, q_2, \ldots q_m) \mid (p_1, \ldots p_n) \in p \wedge (q_1, \ldots q_m) \in q \wedge p_n = q_1\}$

for binary relations, p.q is composition

for set s and relation r, s.r is relational image

q[p] is syntactic variant of p.q

product

$p\text{->}q = \{(p_1, \ldots p_n, q_1, \ldots q_m) \mid (p_1, \ldots p_n) \in p \wedge (q_1, \ldots q_m) \in q\}$

for sets s and t, s->t is cross product

set operators

p+q, p-q, p&q   union, difference, intersection

p in q = ‘every tuple in p is also in q’

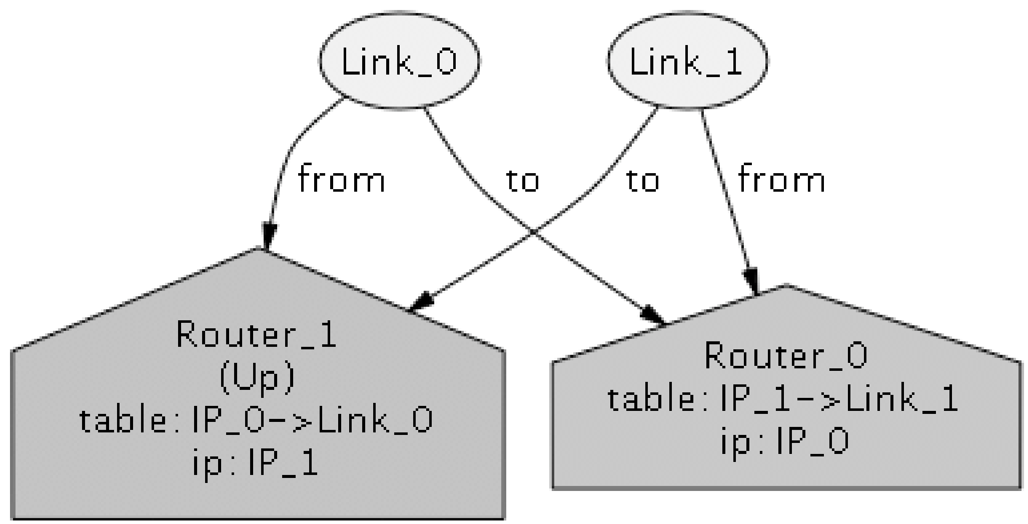
for scalar e and set s, e in s is set membership

for relations p and q, p in q is set subset

alloy declarations

module routing	
-- declare sets & relations	IP: ⟨IP⟩
sig IP {}	
	Link: ⟨LINK⟩
sig Link {from, to: Router}	from, to: ⟨LINK,ROUTER⟩
sig Router {	Router: ⟨ROUTER⟩
ip: IP,	ip: ⟨ROUTER, IP⟩
table: IP ->? Link,	table: ⟨ROUTER, IP, LINK⟩
nexts: set Router	nexts: ⟨ROUTER,ROUTER⟩
}	
	Up: ⟨ROUTER⟩
sig Up extends Router {}	

a sample network



interlude: identity etc

constants

- iden[t]   identity: maps each atom of type of t to itself
- univ [t]   universal: contains every tuple of type t
- none [t]   zero: contains no tuple of type t

examples

- sig Router {
    ip: IP,
    table: IP ->? Link,
    nexts: set Router
    }
- fact NoSelfLinks {all r: Router | r !in r.nexts}
- fact NoSelfLinks' {no Router$nexts & iden [Router]}

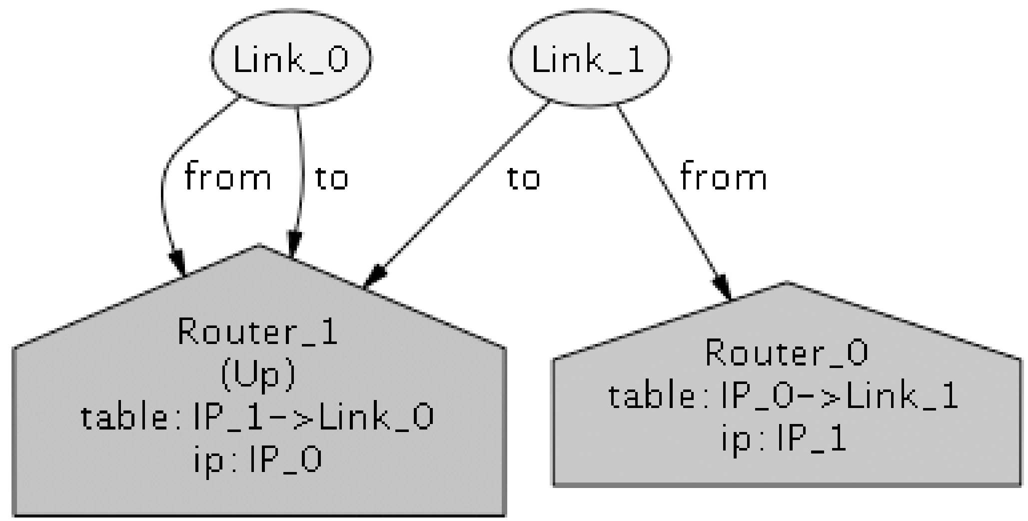
alloy constraints

```
fact Basics {
  all r: Router {
    // router table refers only to router's links
    r.table[IP].from = r
    // nexts are routers reachable in one step
    r.nexts = r.table[IP].to
    // router doesn't forward to itself
    no r.table[r.ip] }
  // ip addresses are unique
  no disj r1, r2: Router | r1.ip = r2.ip }

fun Consistent () {
  // table forwards on plausible link
  all r: Router, i: IP | r.table[i].to in i.~ip.*~nexts }
```

simulation commands

```
-- show me a network that satisfies the Consistent constraint
run Consistent for 2

-- show me one that doesn’t
fun Inconsistent () {not Consistent ()}
run Inconsistent for 2
```

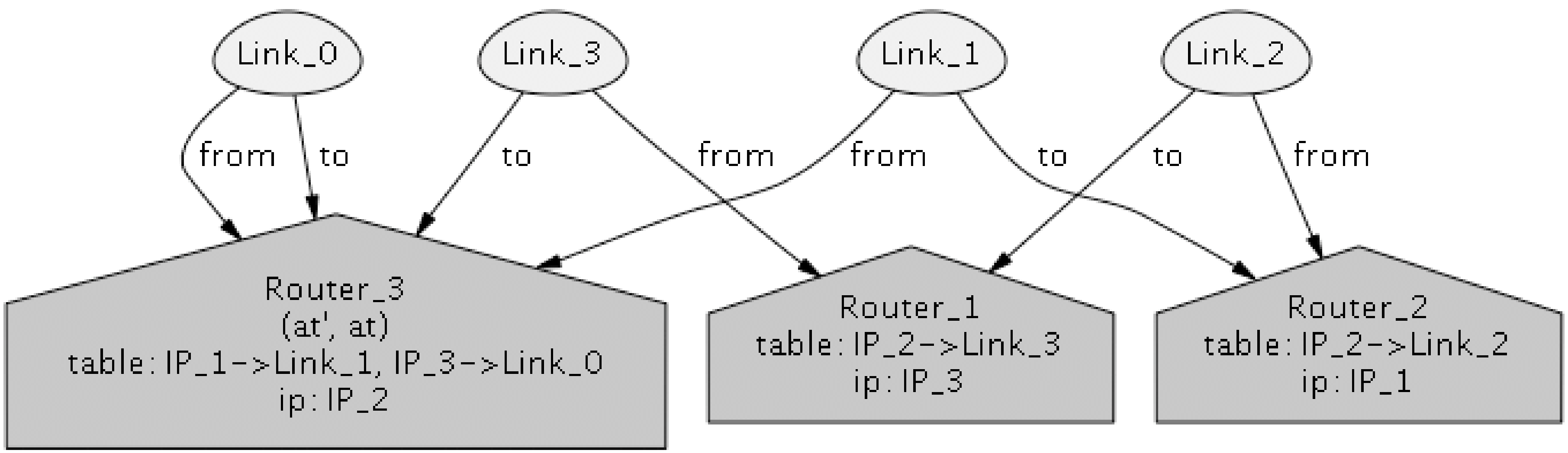
an inconsistent state



assertions & commands

```
-- define forwarding operation
-- packet with destination d goes from at to at’
fun Forward (d: IP, at, at': Router) {
   at' = at.table[d].to
   }

-- assert that packet doesn’t get stuck in a loop
assert Progress {
   all d: IP, at, at': Router |
      Consistent() && Forward (d, at, at') => at != at’
   }

-- issue command to check assertion
check Progress for 4
```

lack of progress



introducing mutation

```
-- links now depend on state
sig Link {from, to: State ->! Router}

-- one table per state
sig Router {ip: IP, table: State -> IP ->? Link}

-- state is just an atom
-- put router connectivity here
sig State {nexts: Router -> Router}
```

state in constraints

```
fact {
   all r: Router, s: State {
      (r.table[s][IP].from)[s] = r
      s.nexts[r] = (r.table[s] [IP].to)[s]
      no r.table[s][r.ip]
      }
   no disj r1, r2: Router | r1.ip = r2.ip
   }

fun Consistent (s: State) {
   all r: Router, i: IP |
      (r.table[s][i].to)[s] in i.~ip.*~(s.nexts)
   }
```

propagation

in one step, each router can …

- incorporate a neighbour's entries
- drop entries

```
fun Propagate (s, s': State) {
   all r: Router |
      r.table[s'] in r.table[s] + r.~(s.nexts).table[s]
   }
```

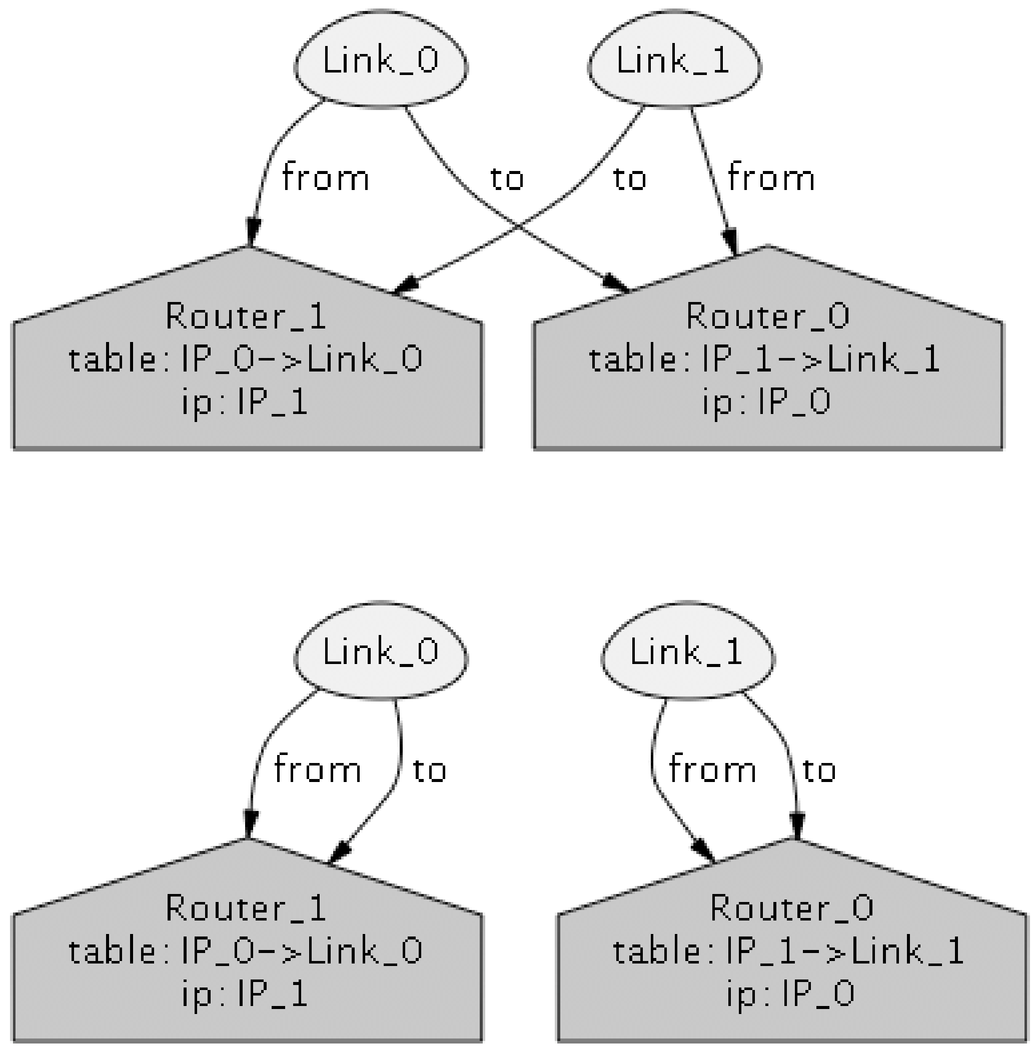
declarative spec

- more possibilities, better checking
- easier than writing operationally

does propagation work?

```
assert PropagationOK {
   all s, s': State |
     Consistent (s) && Propagate (s,s') => Consistent (s')
   }

check PropagationOK for 2
```

no!

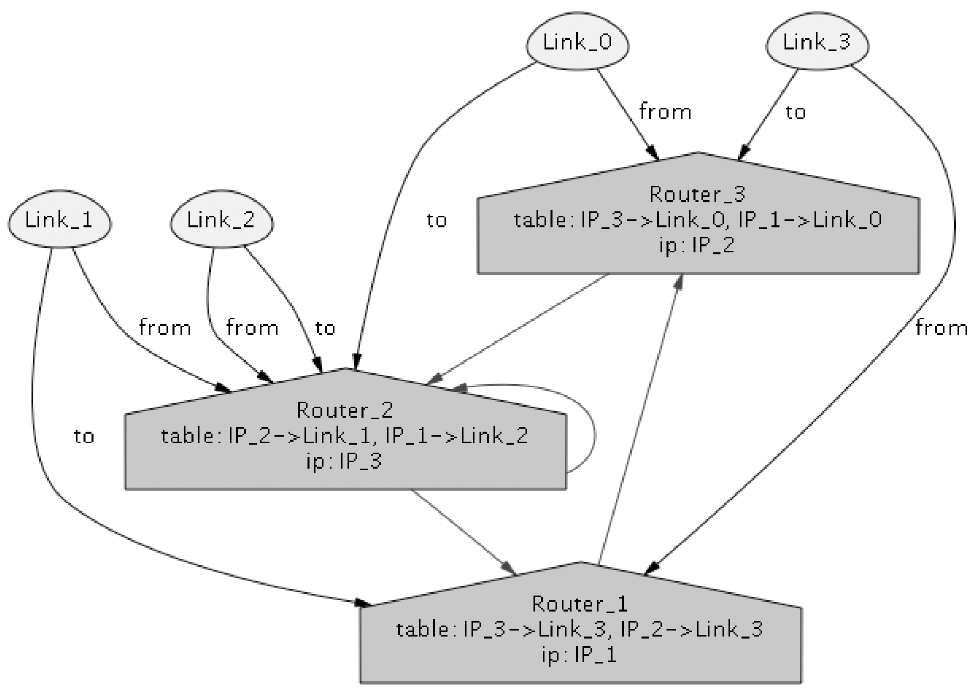


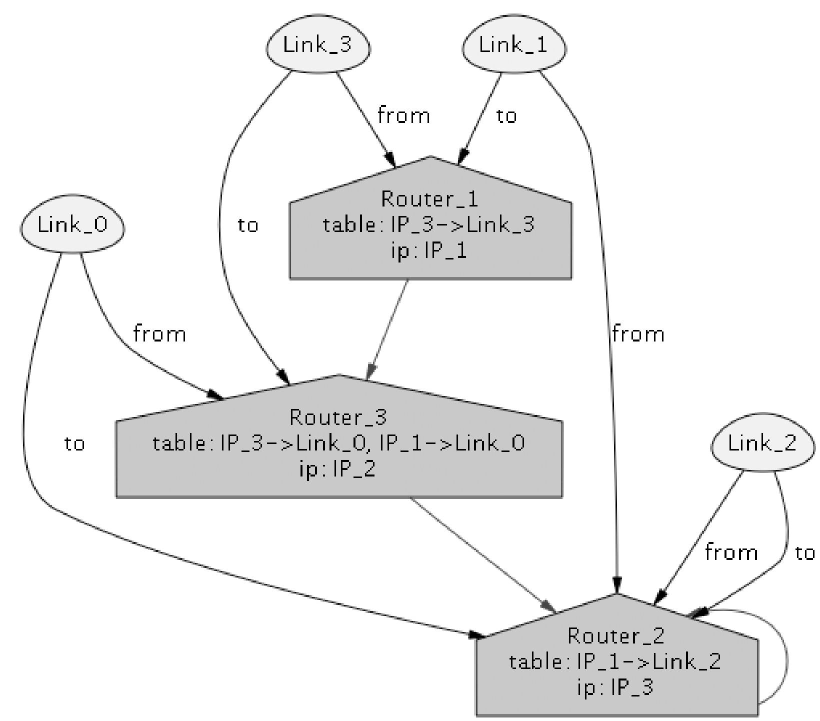
Link_0
Link_1
from
to
to
from
Router_1
table: IP_0->Link_0
ip: IP_1
Router_0
table: IP_1->Link_1
ip: IP_0
Link_0
Link_1
from
to
from
to
Router_1
table: IP_0->Link_0
ip: IP_1
Router_0
table: IP_1->Link_1
ip: IP_0

try again…

```
fun NoTopologyChange (s,s': State) {
   all x: Link {
     x.from[s] = x.from[s']
     x.to[s] = x.to[s']
     }
   }

assert PropagationOK' {
   all s, s': State |
     Consistent (s) && NoTopologyChange (s,s')
         && Propagate (s,s') => Consistent (s')
   }

check PropagationOK' for 4 but 2 State
```

still broken!





language recap (1)

sig X {f: Y} declares

- a set X
- a type TX associated with X
- a relation f with type ⟨TX,TY⟩
- a constraint (all x: X | x.f in Y && one x.f)

fact {…}
introduces a global constraint

fun F (…) {…}
declares a constraint to be instantiated

assert A {…}
declares a theorem intended to follow from the facts

language recap (2)

run F for 3 instructs analyzer to

- find example of F
- using 3 atoms for each type

check A for 5 but 2 X instructs analyzer to

- find counterexample of A
- using 5 atoms for each type, but 2 for type TX

other features (3)

arbitrary expressions in decls

- sig PhoneBook {friends: set Friend, number: friends -> Num}

signature extensions

- sig Man extends Person {wife: option Woman}

polymorphism

- fun Acyclic[t] (r: t->t) {no ^r & iden[t]}

modules

- open models/trees

integers

- #r.table[IP] < r.fanout

models, validity & scopes

semantic elements

- assignment: function from free variables to values
- meaning functions

  E : Expression -> Ass -> Relation
  F : Formula -> Ass -> Bool

examples

- expression: Alice.~likes
- assignment:

  Alice = {(alice)}
  Person = {(alice),(bob),(carol)}
  likes = {(bob, alice),(carol, alice)}
- value: {(bob),(carol)}

- formula: Alice in p.likes
- assignment:
  p = {(bob)}
  Alice = {(alice)}
  Person = {(alice),(bob),(carol)}
  likes = {(bob, alice),(carol, alice)}
- value: true

- formula: all p: Person | Alice in p.likes
- assignment:
  Alice = {(alice)}
  Person = {(alice),(bob),(carol)}
  likes = {(bob, alice),(carol, alice)}
- value: false

validity, satisfiability, etc

meaning of a formula

- Ass (f) = {set of all well-typed assignments for formula f}
- Models (f) = {a: Ass(f) | F[f]a = true}
- Valid (f) = all a: Ass (f) | a in Models(f)
- Satisfiable (f) = some a: Ass (f) | a in Models(f)
- ! Valid (f) = Satisfiable (!f)

checking assertion

- SYSTEM => PROPERTY
- intended to be valid, so try to show that negation is sat
- model of negation of theorem is a counterexample

scope

a scope is a function

- from basic types to natural numbers

assignment a is within scope s iff

- for basic type t, $\#a(t) \le s(t)$

'small scope hypothesis'

- many errors can be found in small scopes
- ie,

  for the theorems f that arise in practice
  if f has a counterexample, it has one in a small scope

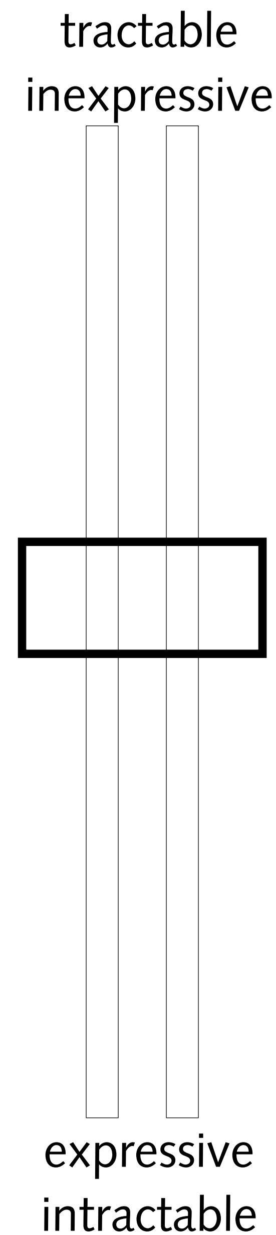
what you've seen

simple notation

- expressive but first-order
- properties in same notation
- static & dynamic constraints
- flexible: no fixed idiom

fully automatic analysis

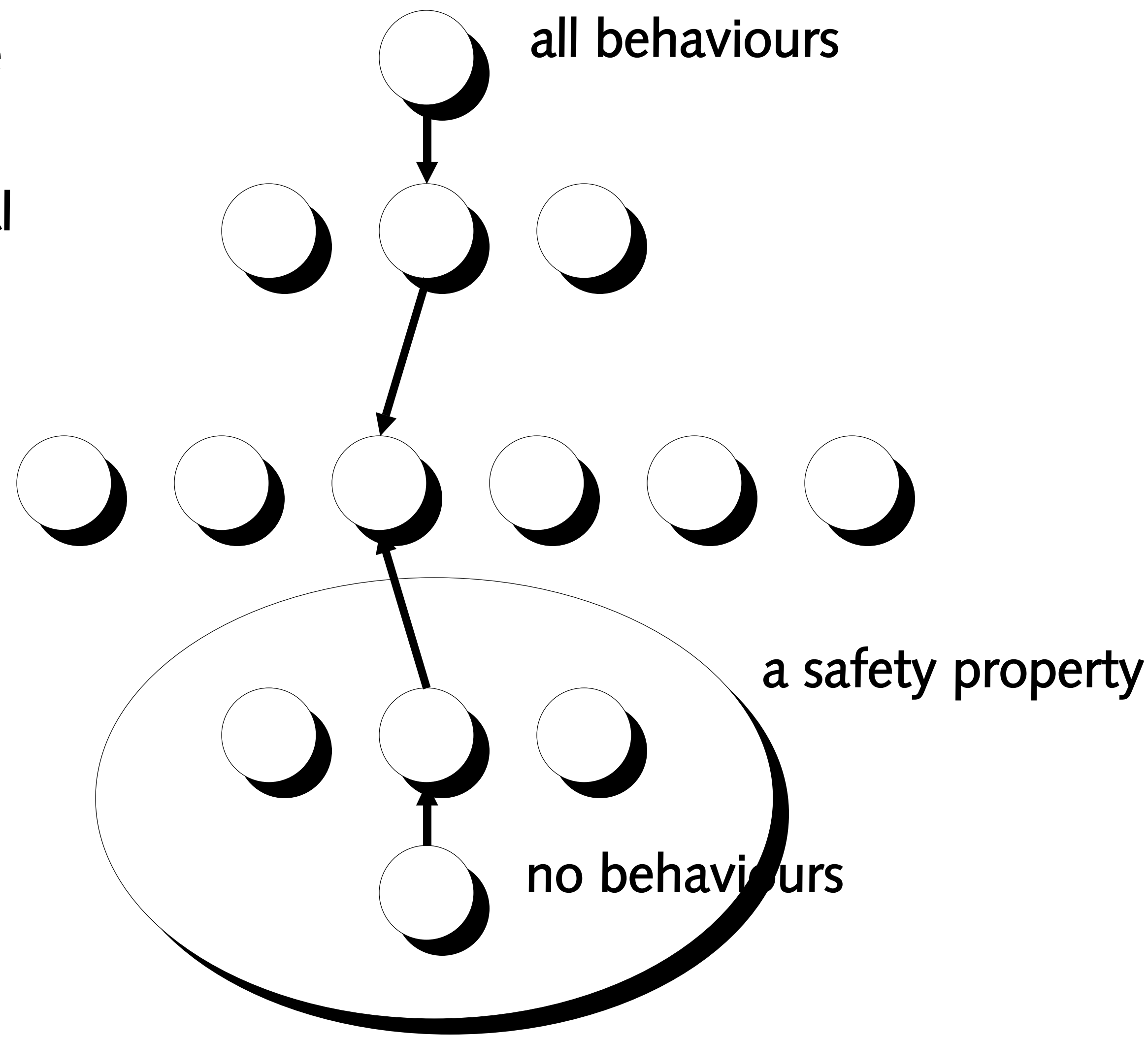
- simulation, even of implicit operations
- checking over large spaces
- concrete output



incrementality

↓ declarative

↑ operational



next time

idioms

- mutation
- frame conditions
- object-oriented structure
- operations and traces

reading

- questions are on web page
- answers to me by mail before class