ALLOY IN 90 MINUTES

Daniel Jackson · RE’05 · Paris · Sept 1, 2005

CSAIL
MIT COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE LABORATORY

topics

10 mins	**intro**	what it is, how it got here
15 mins	**demo**	address book: simulation & checking
5 mins	**key ideas**	elements of alloy approach
20 mins	**basis**	logic & language
10 mins	**patterns**	shows flexibility
20 mins	**example**	hotel locking: environmental assumptions
10 mins	**evaluation**	pluses & minuses

what you won’t learn
- how analysis works
- application to code checking and test case generation
- how language design is justified

introduction

premises

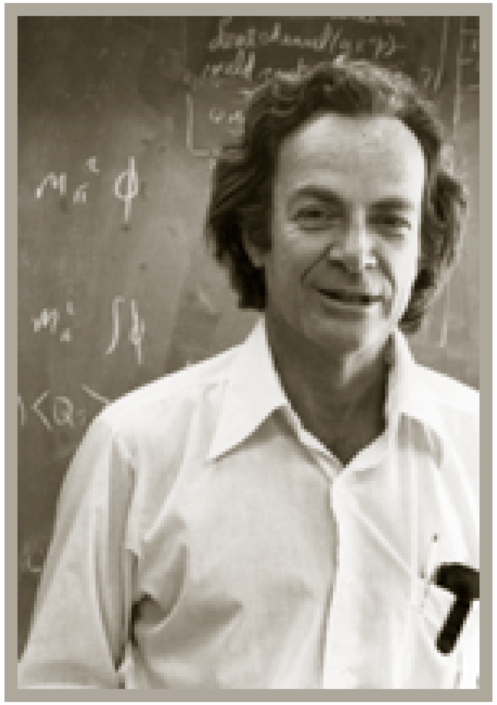
software development needs

- simple, expressive and precise notations
- deep and automatic analyses

… especially in early stages

The first principle is that you must not fool yourself, and you are the easiest person to fool

-- Richard P. Feynman



desiderata

wanted

- syntax: flexible and easy to use
  - eg, declarations & navigations from OMT, Syntropy, etc
- semantics: simple and uniform
  - eg, relational logic from Z
- analysis: fully automatic and interactive
  - eg, symbolic model checking from SMV

transatlantic alloy

Oxford, home of Z

Pittsburgh, home of SMV

the alloy project, 1994-2005

Nitpick [1995]

- a relational subset of Z (Tarski's RC: binary relations, no ∀∃)
- analysis: enumeration of relations + symmetry

Alloy 1.0 [1999]

- language: object modelling (set-valued 'navigation' exprs, ∀∃)
- analysis: WalkSAT, then Davis-Putnam

Alloy 2.0 [2001]

- language: relational logic (arbitrary arity, ∀∃)
- analysis: Chaff, Berkmin

Alloy 3.0 [2004]

- added castless subtypes & overloading

address book: a demo

what we didn’t do

incrementality

- didn’t write a long model and then analyze it

low burden

- no test cases, lemmas or tactics

concrete feedback

- no false alarms, easy to diagnose

key ideas

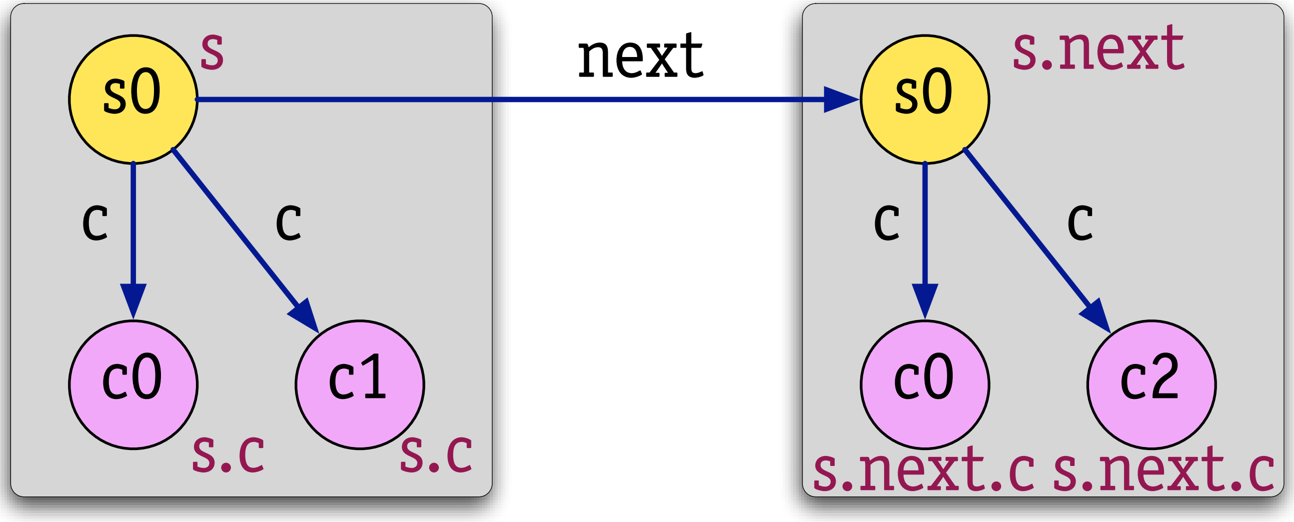
#1: everything's a relation

Alloy uses relations for

- all datatypes -- even sets, scalars and tuples
- structures in space *and* time

key operator is **dot join**

- for taking components of a structure
- for indexing into a collection
- for resolving indirection

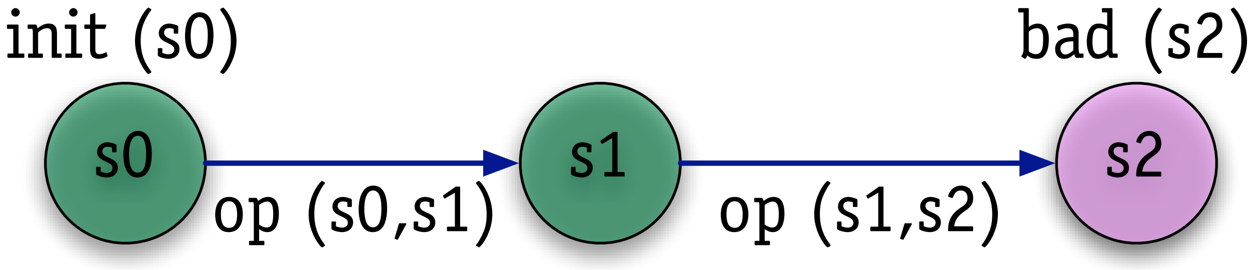


#2: pure logic

no special syntax or semantics for state machines

use constraints for describing

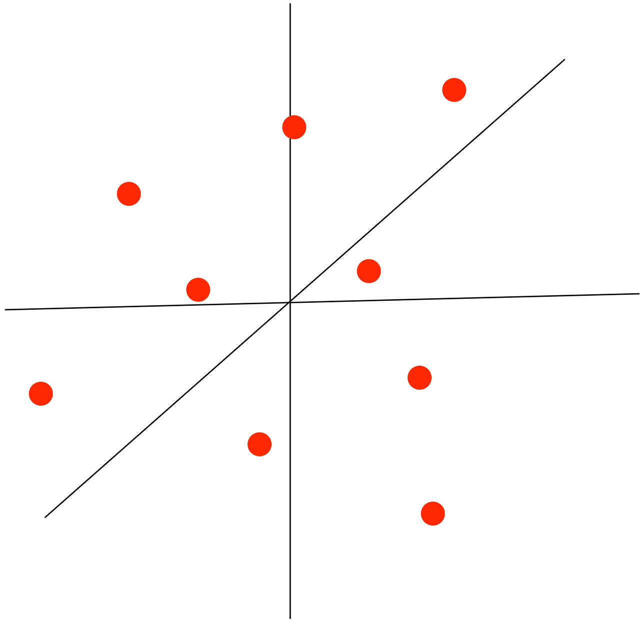
- subtypes & classification
- declarations & multiplicity
- invariants, operations & traces
- assertions, including temporal
- equivalence under refactoring



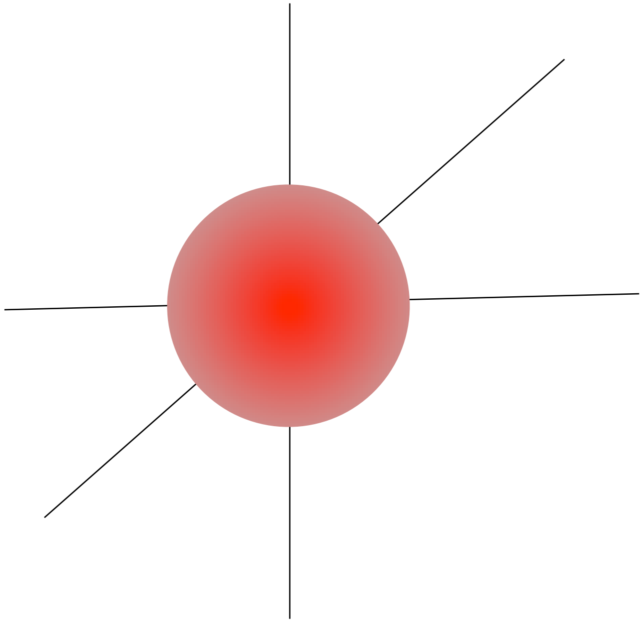
#3: counterexamples & scope

observations about analyzing designs

- most assertions are wrong
- most flaws have small counterexamples



testing:
a few cases of arbitrary size

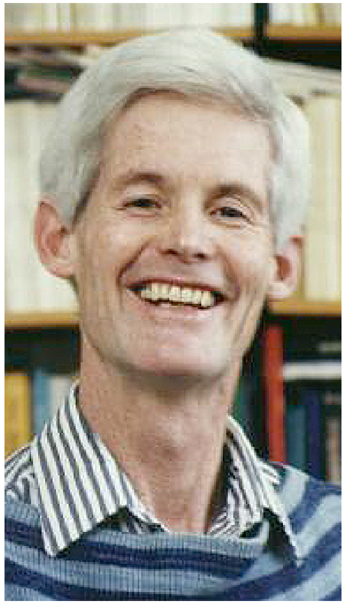


scope-complete:
all cases within small scope

#4: analysis by SAT

SAT, the quintessential hard problem (Cook, 1971)
- SAT is hard, so reduce SAT to your problem

SAT, the universal constraint solver (Kautz, Selman et al 1990's)
- SAT is easy, so reduce your problem to SAT
- solvers: Chaff (Malik), Berkmin (Goldberg & Novikov), others

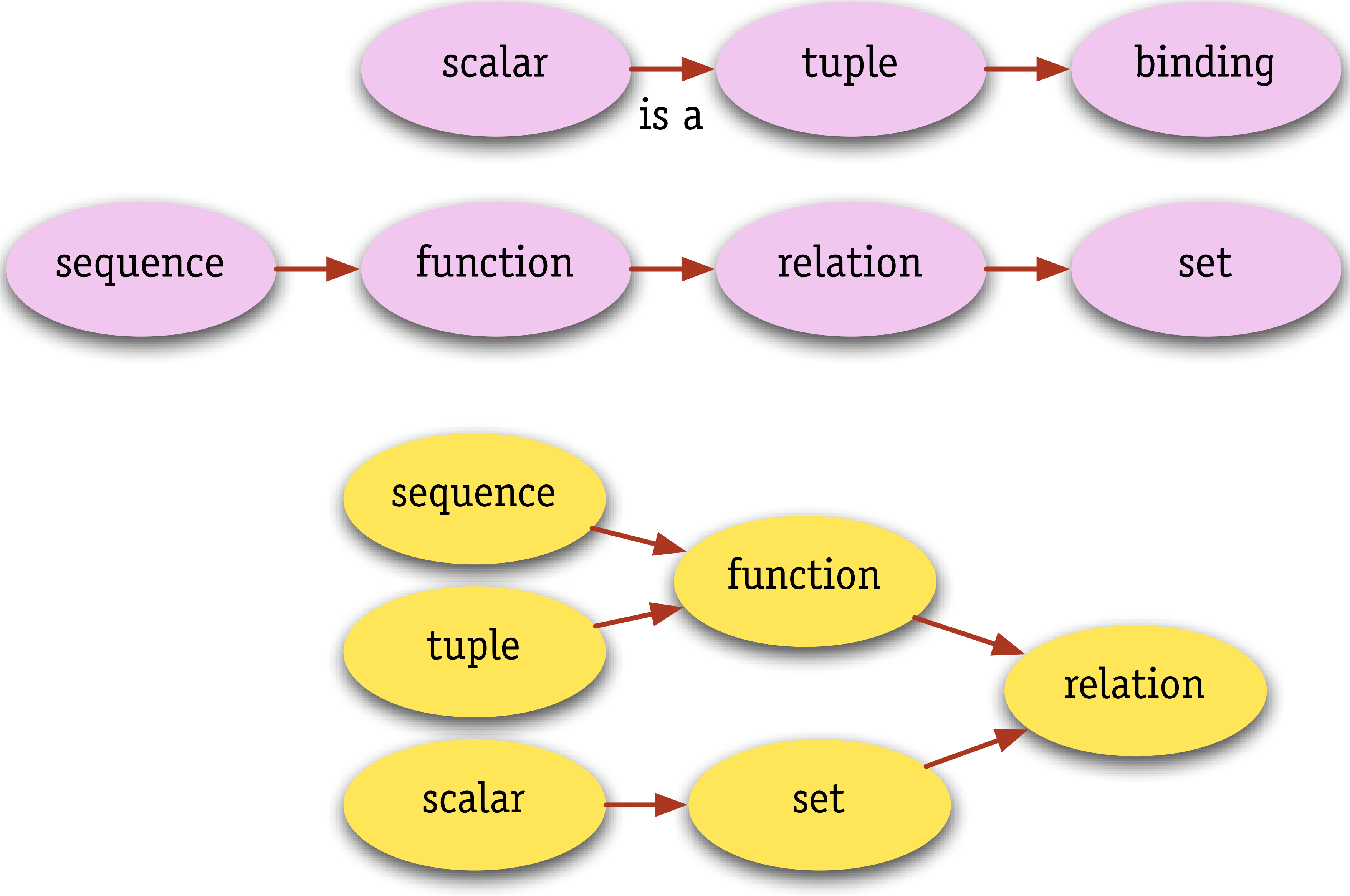


Stephen Cook

Yakov Novikov

logic

relations from Z to A



Z

Alloy

composites as relations

how to represent composite structures?

standard approach
- composites: with nested objects of various kinds
- change of state: with local mutations

Alloy approach
- composites: with atoms and global relations
- change of state: relations include time or state atoms

set operators

union	p + q	$\{t \mid t \in p \lor t \in q\}$
difference	p - q	$\{t \mid t \in p \land t \notin q\}$
intersection	p & q	$\{t \mid t \in p \land t \in q\}$
subset	p in q	$\{(p_1, \ldots p_n) \in p\} \subseteq \{(q_1, \ldots q_n) \in q\}$
equality	p = q	$\{(p_1, \ldots p_n) \in p\} = \{(q_1, \ldots q_n) \in q\}$

```
pred add (b, b': Book, n: Name, a: Addr) {
     b'.addr = b.addr + n->a
     }
```

arrow product

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

idioms

- when s and t are sets
  - s -> t is their cartesian product
  - r: s -> t says r maps atoms in s to atoms in t
- when x and y are scalars
  - x -> y is a tuple

```
sig Book { addr: Name -> Addr }
pred add (b, b': Book, n: Name, a: Addr) {
    b'.addr = b.addr + n->a
    }
```

dot join

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

```
sig Book {
    names: set Name,
    addr: Name -> Addr
    }
pred add (b, b': Book, n: Name, a: Addr) {
    n not in b.names
    b'.addr = b.addr + n->a
    }
```

what does addr.Addr.n denote?

join idioms

when `p` and `q` are binary relations

- `p.q` is standard relational composition

when `r` is a binary relation and `s` is a set

- `s.r` is relational image of `s` under `r` ('navigation')
- `r.s` is relational image of `s` under `~r` ('backwards navigation')

when `f` is a function and `x` is a scalar

- `x.f` is application of `f` to `x`

other handy operators

transitive closure	^p	smallest $q \mid q.q \subseteq q \wedge p \subseteq q$
override	p ++ q	q + (p - dom q <: p)

… *and 5 more*

```
pred add (b, b': Book, n: Name, a: Addr) {
    b'.addr = b.addr + n->a
    }
pred add (b, b': Book, n: Name, a: Addr) {
    b'.addr = b.addr ++ n->a
    }
```

a sample instance

```
sig Name, Addr {}
sig Book { addr: Name -> Addr }
pred add (b, b': Book, n: Name, a: Addr) {
  b'.addr = b.addr + n->a
  }

  Name = N0 + N1
  Addr = A0 + A1
  Book = B0 + B1
  b = B0, b’ = B1, n = N1, a = A1
  addr =
      B0 -> N0 -> A0,
      B1 -> N0 -> A0,
      B1 -> N1 -> A1
```

quantifiers & cardinalities

quantifiers

all, **some**, **no**, **one**, **lone**

quantified formulas

all x: e | F $\bigwedge_{v \in x} F[\{(v)\}/x]$

cardinality expressions

no e	#e = 0
some e	#e > 0
lone e	#e =< 1
one e	#e = 1

```
sig Book { addr: Name -> Addr }
pred show () { some addr }
```

declarations & multiplicity

multiplicity keywords: **some**, **one**, **lone**, **set**

set declarations

s: ***m*** e	s **in** e **and** ***m*** e
s: e	s: **one** e

relation declarations

r: e ***m*** -> ***n*** e′	r **in** e -> e′
	all x: e | ***n*** x.r
	all x: e′ | ***m*** r.x

sig Book { names: **set** Name, addr: Name -> Addr }
sig Book { addr: Name -> **lone** Addr }
sig Book { addr: (Name -> **lone** Addr) -> Time }

puns

to support familiar declaration syntax

› Alloy declaration	`r: A -> B`
has traditional reading	$r \in 2^{(A \times B)}$
has Alloy reading	$r \subseteq A \times B$

to support ‘navigation expressions’

› Alloy expression	`x.f.g`
has traditional reading	g(f(x)) unless f(x) undefined or a set
has Alloy reading	image (image({(x)}, f), g)

language

elements of an alloy model

signatures and fields

- introduces sets and relations
- ‘extends’ hierarchy for classification and subtypes

constraints paragraphs

- facts: assumed to hold
- predicates: reusable constraints
- functions: reusable expressions
- assertions: conjectures to check

commands

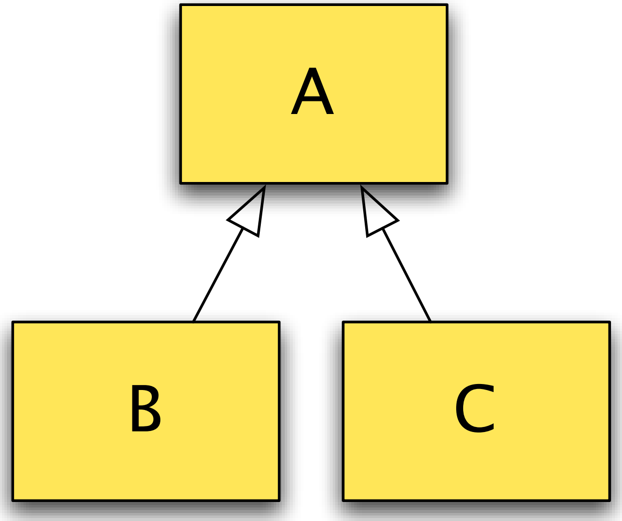
- run: generate instances of a predicate
- check: generate counterexamples to an assertion

signatures

```
sig A {}
sig B extends A {}
sig C extends A {}
```

means

```
B in A
C in A
no B & C
```



fields

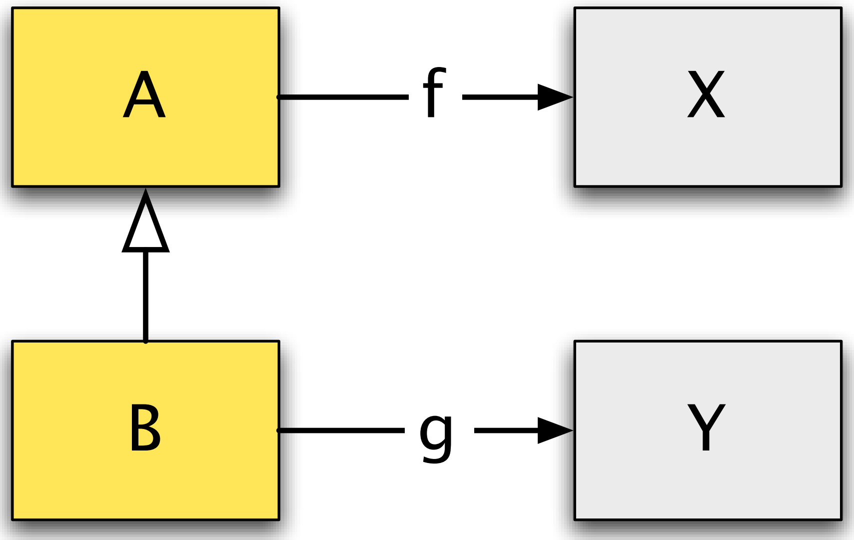
```
sig A {f: set X}
sig B extends A {g: set Y}
```

means

```
B in A
f: A -> X
g: B -> Y
```

some well-defined expressions
(for a: A, b: B)

```
a.f
b.g
b.f
a.g
```



fact, pred, run

fact F {…}
pred P () {…}
run P

means

fact: assume constraint F holds
pred: define constraint P
run: find an instance that satisfies P *and* F

assert, check

fact F {...}
assert A {...}
check A

means

fact: assume constraint F holds
assert: believe that A follows from F
check: find an instance that satisfies F *and not* A

example, revisited

```
module examples/addressBook/addLocal

abstract sig Target {}
sig Addr extends Target {}
sig Name extends Target {}
sig Book {addr: Name -> Target}

fact Acyclic {all b: Book | no n: Name | n in n.^(b.addr)}
fun lookup (b: Book, n: Name): set Addr {n.^(b.addr) & Addr}
pred add (b, b': Book, n: Name, t: Target) {b'.addr = b.addr + n->t}
run add for 3 but 2 Book

assert addLocal {
   all b,b': Book, n,n': Name, a: Addr |
      add (b,b',n,a) and n != n' => lookup (b,n') = lookup (b',n') }
check addLocal for 3 but 2 Book
```

patterns

sample patterns

Trace	states are ordered into traces by a relation
Local State	state modelled within object signatures
Event	events are modelled as explicit objects
Reiter Frame	frame conditions in Ray Reiter's style

pattern: trace

```
open util/ordering[State]

pred init (s: State) {…}
pred op1 (s, s′: State) {…}
…
pred opN (s, s′: State) {…}

fact traces {
   init (first ())
   all s: State - last() | let s' = next (s) | op1 (s, s′) or … or opN (s, s′)
   }

pred Safe (s: State) {…}
assert alwaysP {all s: State | P(s)}
```

pattern: local state

```
sig Time {...}
sig X {}
sig Object {
   static: X,
   dynamic: X -> Time
   }

pred op (t, t′: Time, o: Object, x: X) {
   o.dynamic.t′ = x
   all o′: Object - o | o′.dynamic.t′ = o′.dynamic.t
or
   dynamic.t′ = dynamic.t ++ o->x
   }
```

pattern: event

```
sig Time {}
sig O {dynamic: X -> Time}
sig Event {pre, post: Time, o: O, x: X}
    {dynamic.post = dynamic.pre ++ o -> x}

fact {
    all t: Time - last() | let t’ = next(t) |
        some e: Event | e.pre = t and e.post = t’
    }
```

pattern: event classification

```
sig Time {}
sig O {f: X -> Time, g: Y -> Time}
sig Event {pre, post: Time, o: O, x: X}
  {f.post = f.pre ++ o -> x}

sig SubEvent extends Event {y: Y}
  {y.post = y.pre ++ o -> y}
```

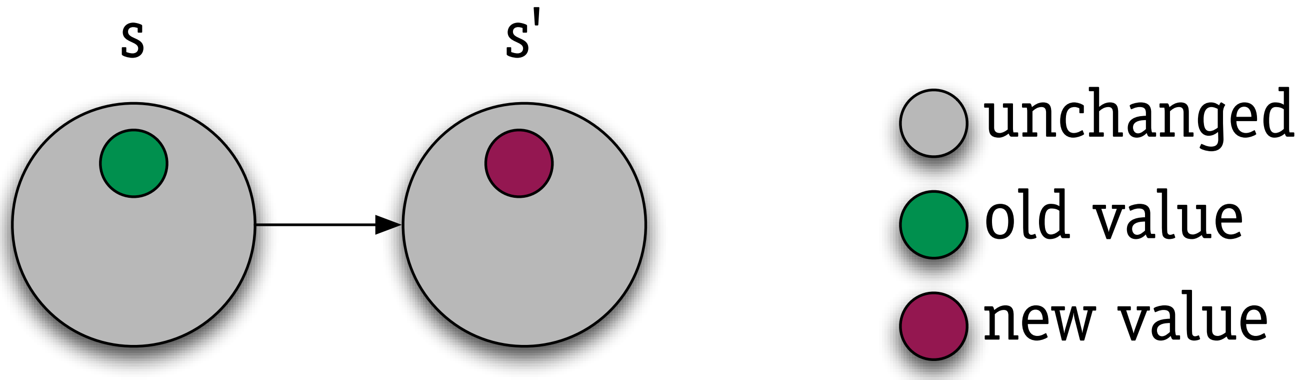
reiter’s frame conditions

in declarative models

- unmentioned ≠ unchanged

Ray Reiter’s scheme

- add ‘explanation closure axioms’
  - if field f changed, then event e happened



See: Alex Borgida, John Mylopoulos and Raymond Reiter.
On the Frame Problem in Procedure Specifications.
IEEE Transactions on Software Engineering, 21:10 (October 1995), pp. 785-798.

pattern: reiter frame

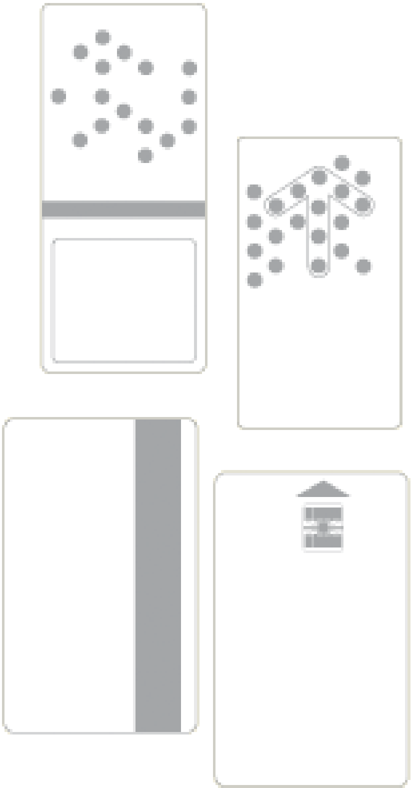
```
sig Time {}
sig O {f: X -> Time, g: Y -> Time}
sig EventA {pre, post: Time, ...}
sig EventB {pre, post: Time, ...}

fact {
  all t: Time - last() | let t’ = next(t) |
     some e: Event {
        e.pre = t and e.post = t’
        f.t = f.t’ or e in EventA
        g.t = g.t’ or e in EventB
        }
  }
```

recodable hotel locks

hotel locking

recodable locks (since 1980)
- new guest gets a different key
- lock is ‘recoded’ to new key
- last guest can no longer enter

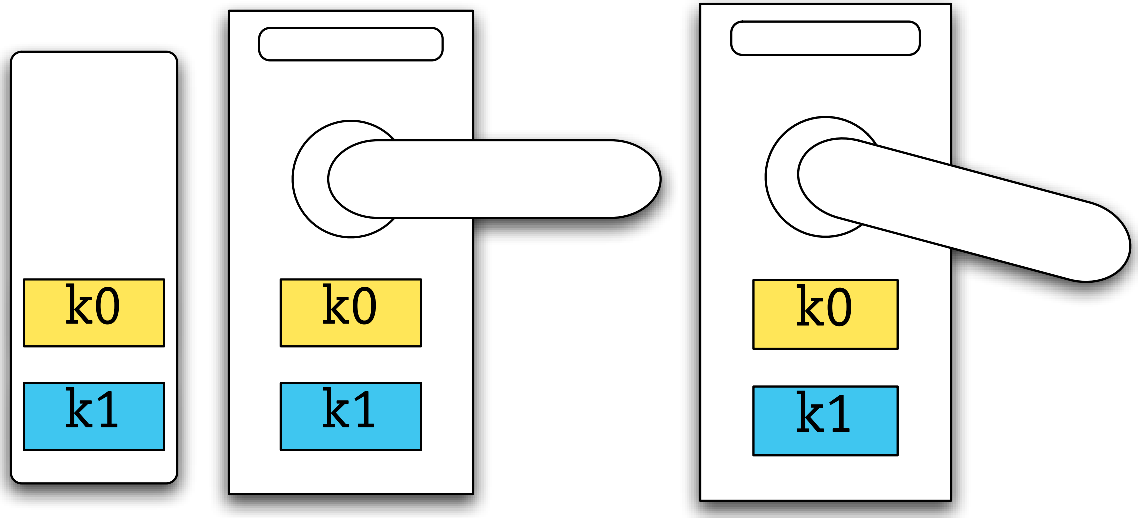
how does it work?
- locks are standalone, not wired



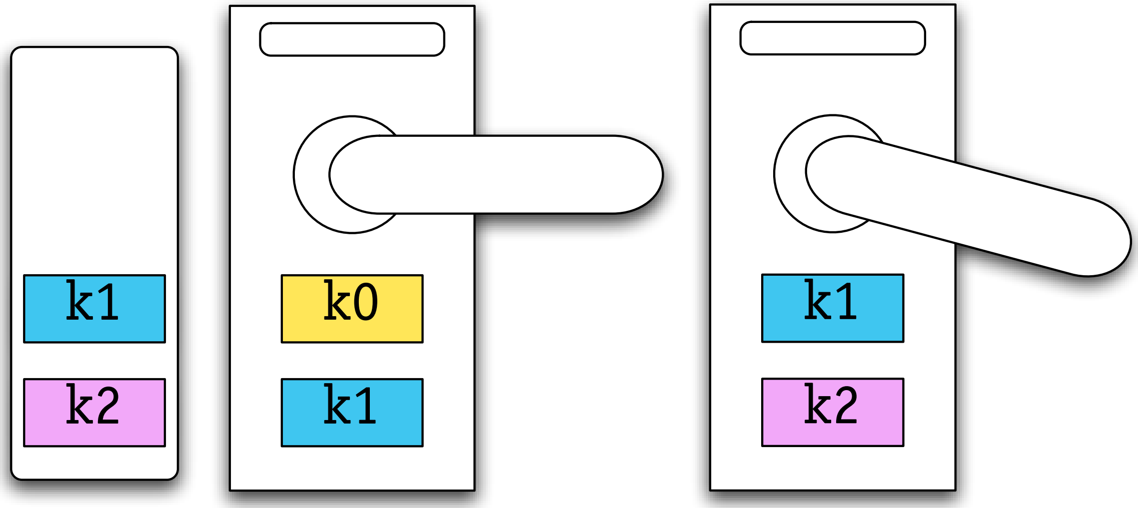
a recodable locking scheme

from US patent 4511946; many other similar schemes

card & lock have two keys
if both match, door opens



if first card key matches
second door key, door opens
and lock is recoded



modelling in alloy: state

```
sig Key, Time {}
sig Card {fst, snd: Key}
sig Room {fst, snd: Key one -> Time}

one sig Desk {
   prev: (Room -> lone Key) -> Time,
   issued: Key -> Time,
   occ: (Room -> Guest) -> Time
   }

sig Guest {cards: Card -> Time}
```

initialization

```
pred init (t: Time) {
   -- room's previous key is its second key
   Desk.prev.t = snd.t
   -- each key is the first or second key of at most one room
   (fst + snd).t : Room lone -> Key
   -- set of keys issued is first and second keys of all rooms
   Desk.issued.t = Room.(fst+snd).t
   -- no cards handed out, and no rooms occupied
   no cards.t and no occ.t
   }
```

event classification

```
abstract sig HotelEvent {
   pre, post: Time,
   guest: Guest
   }

abstract sig RoomCardEvent extends HotelEvent {
   room: Room,
   card: Card
   }
```

checking in

```
sig CheckinEvent extends RoomCardEvent { }
  {
  card.fst = room.(Desk.prev.pre)
  card.snd not in Desk.issued.pre
  cards.post = cards.pre + guest -> card
  Desk.issued.post = Desk.issued.pre + card.snd
  Desk.prev.post = Desk.prev.pre ++ room -> card.snd
  Desk.occ.post = Desk.occ.pre + room -> guest
  }
```

entering a room

```
abstract sig EnterEvent extends RoomCardEvent { }
   {card in guest.cards.pre}

sig NormalEnterEvent extends EnterEvent { }
   {card.fst = room.fst.pre and card.snd = room.snd.pre}

sig RecodeEnterEvent extends EnterEvent { }
   {
   card.fst = room.snd.pre
   fst.post = fst.pre ++ room -> card.fst
   snd.post = snd.pre ++ room -> card.snd
   }
```

reiter-style frame conditions

```
fact Traces {
   init (first ())
   all t: Time - last () | let t' = next (t) |
        some e: HotelEvent {
             e.pre = t and e.post = t'
             fst.t = fst.t' and snd.t = snd.t' or e in RecodeEnterEvent
             prev.t = prev.t' and issued.t = issued.t' and cards.t = cards.t'
                  or e in CheckinEvent
             occ.t = occ.t' or e in CheckinEvent + CheckoutEvent
             }
   }
```

does the scheme work?

safety condition

- if an enter event occurs, and the room is occupied, then the guest who enters is an occupant

```
assert NoBadEntry {
  all e: Enter | let occs = Desk.occ.(e.pre) [e.room] |
     some occs => e.guest in occs
  }
```

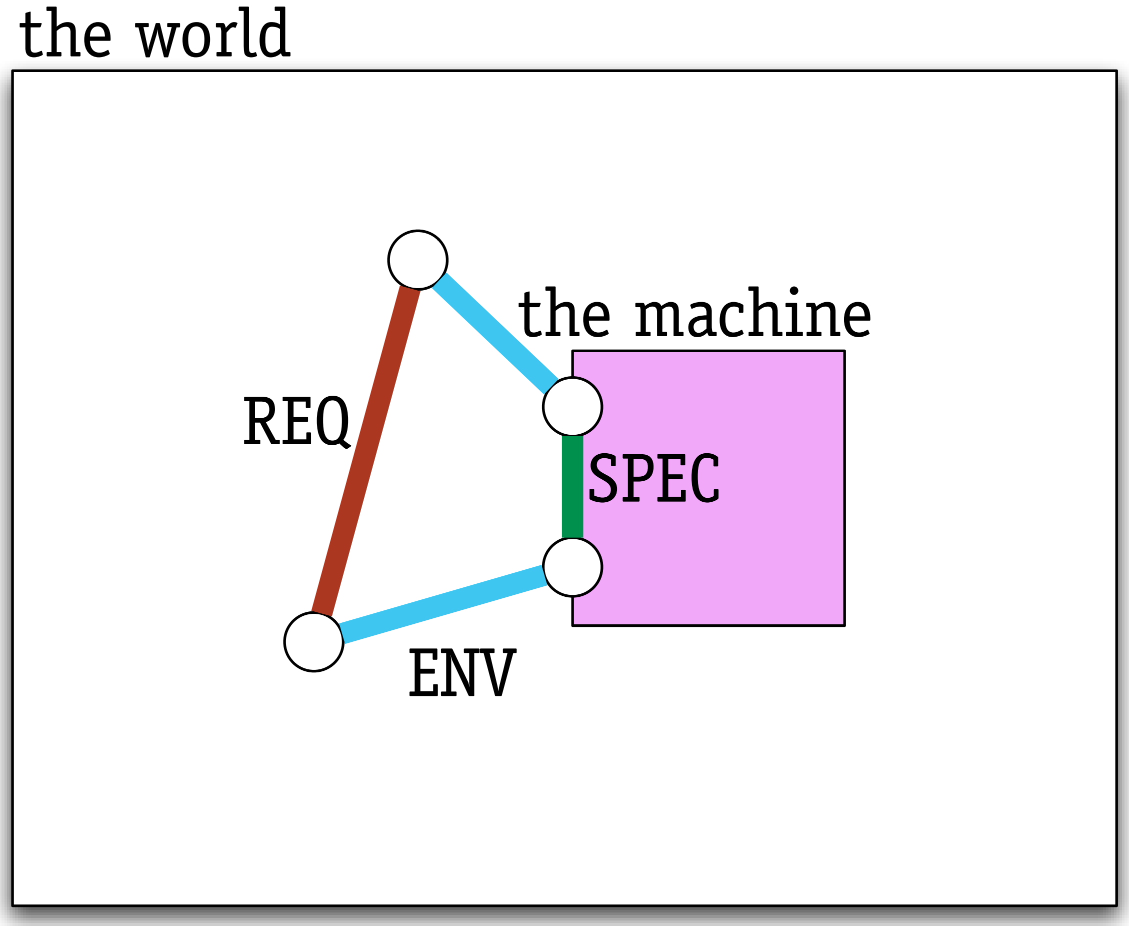
demo

constraining the environment

after checking in, guest immediately enters room:

```
fact NoIntervening {
   all c: CheckinEvent |
      some e: EnterEvent {
         e.pre = c.post
         e.room = c.room
         e.guest = c.guest
         }
   }
```

machines & environments



specification is at machine interface,
but requirement might not be

homework: hacking the hotel

in an earlier patent
- lock required match only on **first** key

suppose guest can make new cards
- using keys from cards she holds

is system secure?

your task
- make one line change to `NormalEnter` event to reflect this
- rerun `NoBadEntry` check to expose attack

evaluation

alloy case studies at MIT

many small case studies

- intentional naming [Balakrishnan+]
- Chord peer-to-peer lookup [Kaashoek+]
- Unison file sync [Pierce+]
- distributed key management
- beam scheduling for proton therapy

typically

- 100-1000 lines of Alloy
- analysis in 10 secs - 1 hour
- 3-20 person-days of work

some alloy applications

in industry

- animating requirements (Venkatesh, Tata)
- military simulation (Hashii, Northtrop Grumman)
- role-based access control (Zao, BBN)
- generating network configurations (Narain, Telcordia)

in research

- exploring design of switching systems (Zave, AT&T)
- checking semantic web ontologies (Jin Song Dong)
- enterprise modelling (Wegmann, EPFL)
- checking refinements (Bolton, Oxford)
- security features (Pincus, MSR)

alloy in education

courses using Alloy at Michigan State (Laura Dillon), Imperial College (Michael Huth), National University of Singapore (Jin Song Dong), University of Iowa (Cesare Tinelli), Queen's University (Juergen Dingel), University of Waterloo (Joanne Atlee), Worcester Polytechnic (Kathi Fisler), University of Wisconsin (Somesh Jha), University of California at Irvine (David Rosenblum), Kansas State University (John Hatcliff and Matt Dwyer), University of Southern California (Nenad Medvidovic), Georgia Tech (Colin Potts), Politecnico di Milano (Carlo Ghezzi), Rochester Institute of Technology (Michael Lutz), University of Auckland (John Hamer, Jing Sun), Stevens Institute (David Naumann), USC (David Wilczynski)

good things

conceptual simplicity and minimalism
- very little to learn
- WYSIWYG: no special semantics (eg, for state machines)
- expressive declarations

high-level notation
- constraints -- can build up incrementally
- relations flexible and powerful
- much more succinct than most model checking notations

automatic analysis
- no lemmas, tactics, etc
- counterexamples are never spurious
- visualization a big help
- can do many kinds of analysis: refinement, BMC, etc

bad things

relations aren’t a panacea

- sequences are awkward
- treatment of integers limited

limitations of logic

- recursive functions hard to express
- sometimes, want iteration and mutation

limitations of language

- module system doesn’t offer real encapsulation

limitations of tool

- tuned to generating instances (hard) rather than checking instances (easy)

acknowledgments

current students
& collaborators
who've worked on Alloy
Greg Dennis
Derek Rayside
Robert Seater
Mana Taghdiri
Emina Torlak
Jonathan Edwards
Vincent Yeung

former students
who've worked on Alloy
Sarfraz Khurshid
Mandana Vaziri
Ilya Shlyakhter
Manu Sridharan
Sam Daitch
Andrew Yip
Ning Song
Edmond Lau
Jesse Pavel
Ian Schechter
Li-kuo Lin
Joseph Cohen
Uriel Schafer
Arturo Arizpe

for more info

http://alloy.mit.edu

- downloads
- papers
- case studies

alloy@mit.edu

- questions about Alloy
- send us a challenge

dnj@mit.edu

- happy to hear from you!

Software Abstractions

- MIT Press, 2006