idioms of logical modelling

Daniel Jackson · MIT

SBMF/ICGT · Natal · Sept 20, 2006

CSAIL
MIT COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE LABORATORY

introduction

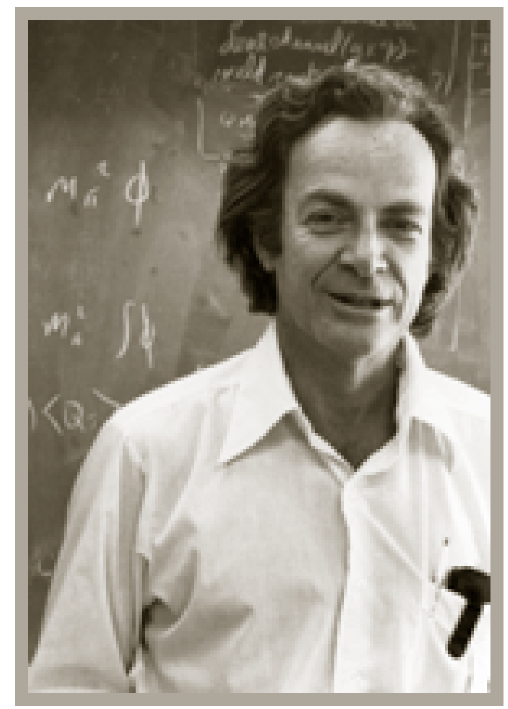
premises

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

syntax: flexible and easy to use

› eg, declarations & navigations like OMT, Syntropy, etc

semantics: simple and uniform

› eg, relational logic like Z

analysis: fully automatic and interactive

› eg, symbolic model checking like SMV

transatlantic alloy

Oxford, home of Z

Pittsburgh, home of SMV

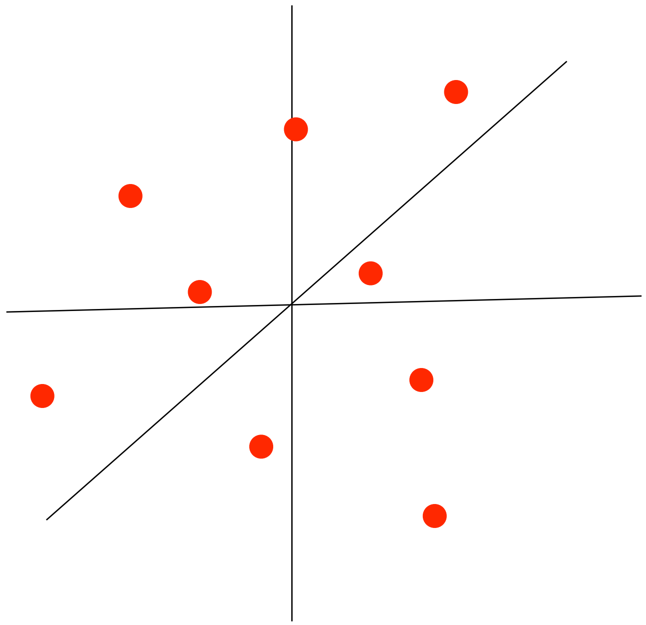
alloy project

version	*language*	*analysis*	*sample case study*
Nitpick (1995)	relational calculus subset of Z	relation enumeration	IPv6 routing
Alloy 1 (1999)	+ navigation exps quantifiers	WalkSAT, DP	intentional naming
Alloy 2 (2001)	+ relational ops higher arity	Chaff, Berkmin symmetry, sharing	key management, Unison filesync
Alloy 3 (2004)	+ subtyping, overloading	+ atomization (bad)	Mondex electronic purse
Alloy 4 (2007)	+ imperative features	sparse matrices better sharing	

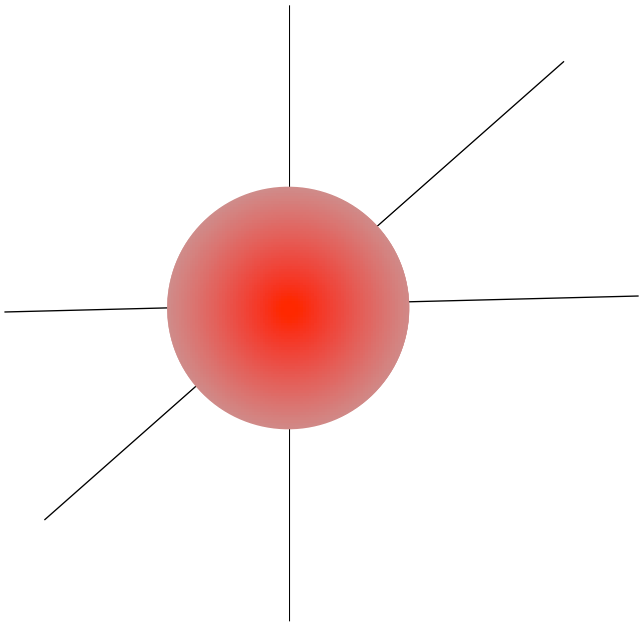
scope-complete analysis

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

pure logic modelling

traditional approach

built-in notions

- state, invariant, operation, trace

standard idiom

- a fixed view of software systems

examples

- state-invariant-operation (Z, B, VDM, OCL)
- state-update-formula (SMV, Murphi)
- state-guarded command-formula (SPIN)
- heap-stack-if-while (Pathfinder, Bandera)

pure logic modelling

suppose we had

- no built-in notions
- no fixed idiom

what might the language look like?
what idioms could we express?
how naturally could we simulate standard idioms?

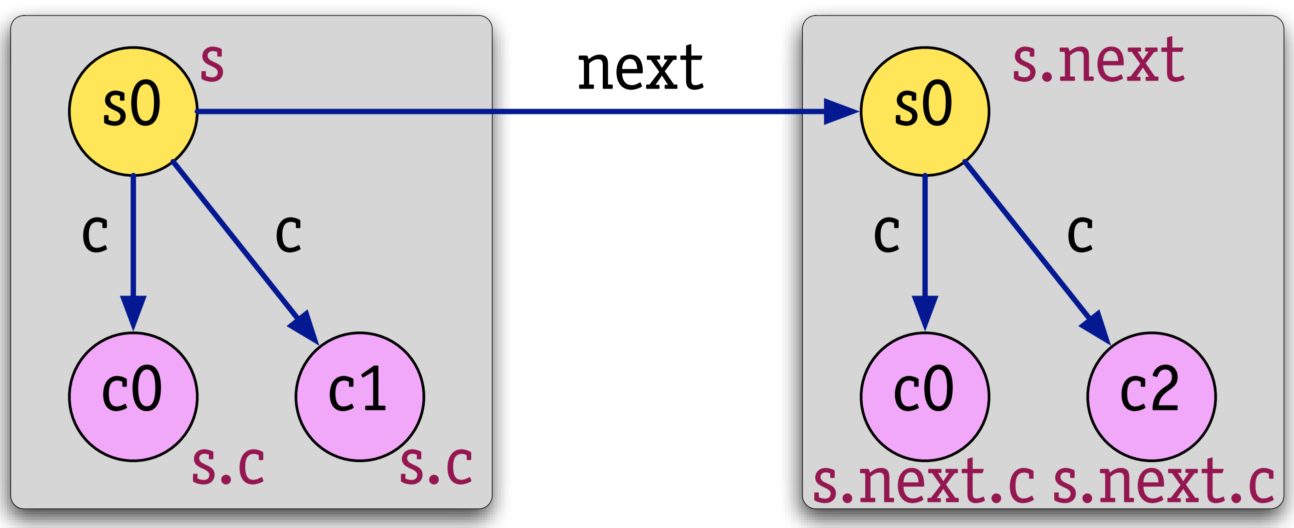
everything's a relation

Alloy uses relations for

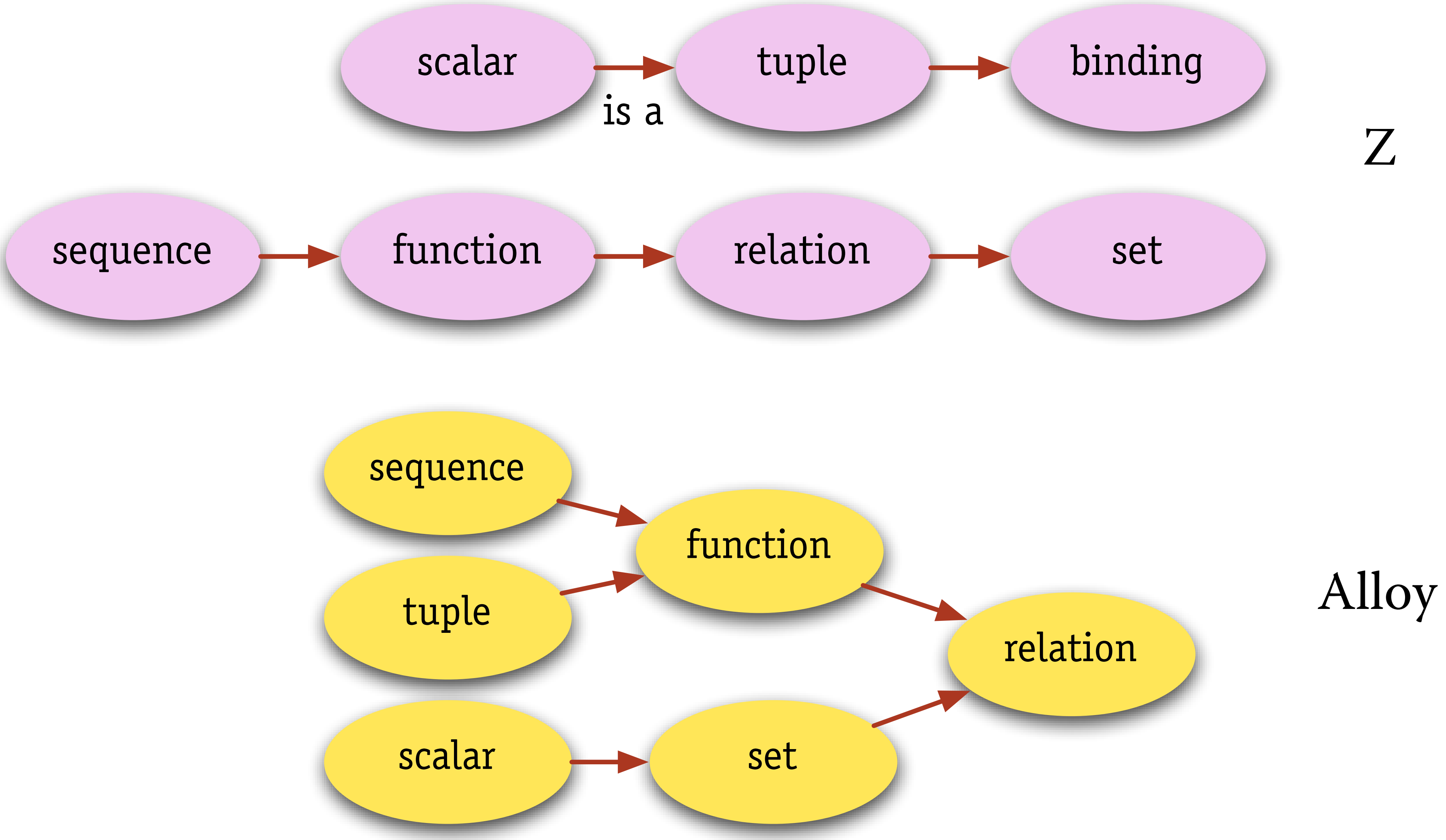
- all datatypes -- even sets, scalars and tuples
- structures in space *and* time

key operator is **dot join**

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



relations from Z to A

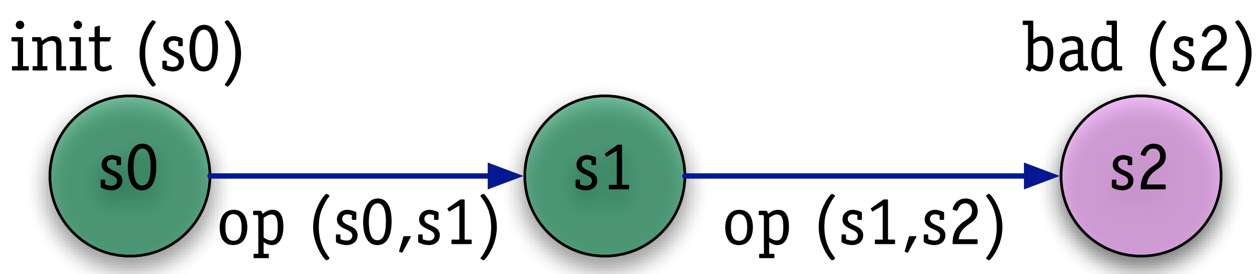


everything's a constraint

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

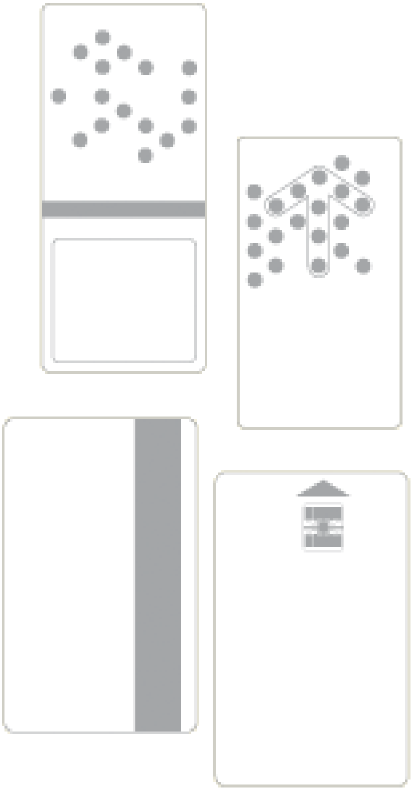


an example: hotel locking

hotel locking

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

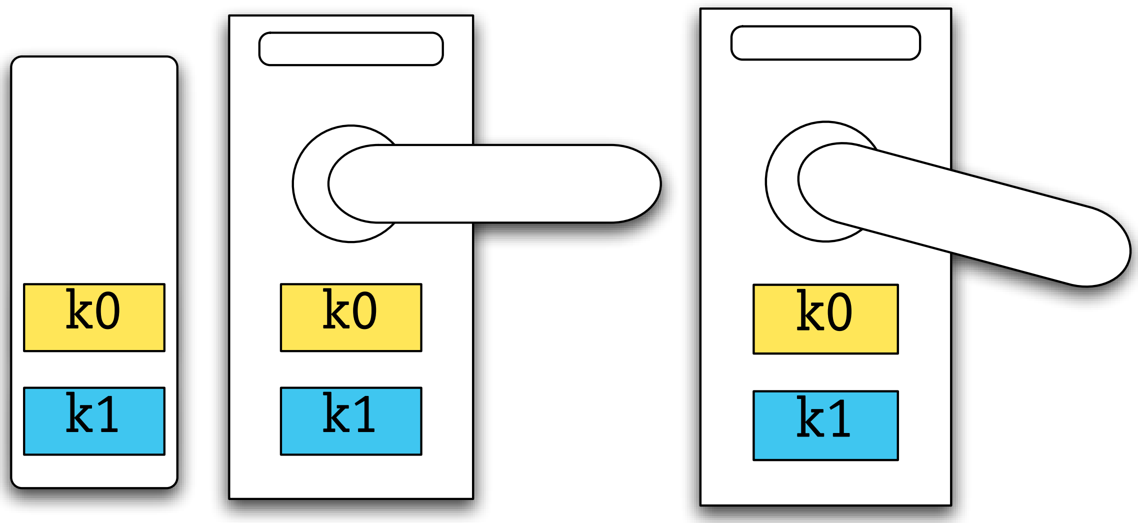
how does it work?
- locks are standalone, not wired



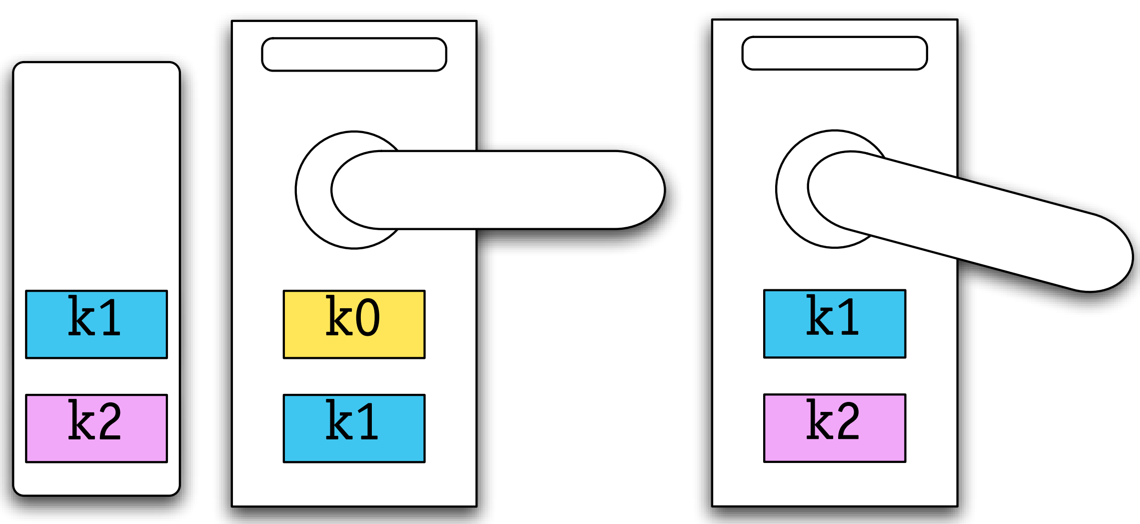
a recodable locking scheme

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



challenge

model this scheme

```
LOCKS() -- locking mechanism
GUESTS() -- how guests and hotel staff are supposed to behave
```

formulate a requirement

```
SAFE() -- only guest who owns a room can enter it
```

check

LOCKS() **and** GUESTS() **implies** SAFE() ??

elements of alloy

alloy in 3 slides

signatures
- provide classification hierarchy for sets
- composite structure of objects
- local name space for relations
- incremental development

relational logic
- unusually simple and uniform
- generalized join

facts, predicates and assertions
- simple packaging of constraints

signatures & fields

```
sig A {}
-- introduces a set of atoms called A
sig B extends A {}
-- introduces a subset B of A
sig C extends A {}
-- introduces a subset C of A disjoint from B
sig A {f: B}
-- introduces a binary relation from A to B called f
sig A {f: B->C}
-- introduces a ternary relation from A to B to C called f
```

relational operators

p + q	$\{t \mid t \in p \lor t \in q\}$
p - q	$\{t \mid t \in p \land t \notin q\}$
p & q	$\{t \mid t \in p \land t \in q\}$
p -> q	$\{(p_1, \ldots p_n, q_1, \ldots q_m) \mid (p_1, \ldots p_n) \in p \land (q_1, \ldots q_m) \in q$
p . q	$\{(p_1, \ldots p_{n-1}, q_2, \ldots q_m) \mid (p_1, \ldots p_n) \in p \land (p_n, q_2, \ldots q_m) \in q\}$
p in q	$\{(p_1, \ldots p_n) \in p\} \subseteq \{(q_1, \ldots q_n) \in q\}$
p = q	$\{(p_1, \ldots p_n) \in p\} = \{(q_1, \ldots q_n) \in q\}$

eg, given **sig** A {f: B->C}
some expressions and their types:
a.f: B->C
f.c: A->B
b.(a.f): set C

constraints & commands

fact {F}
-- establishes formula F, as an assumption

pred P () {Fp}
-- declares predicate P; invocation equivalent to inlining Fp

assert A () {Fa}
-- declares assertion A, claiming that formula Fa is valid

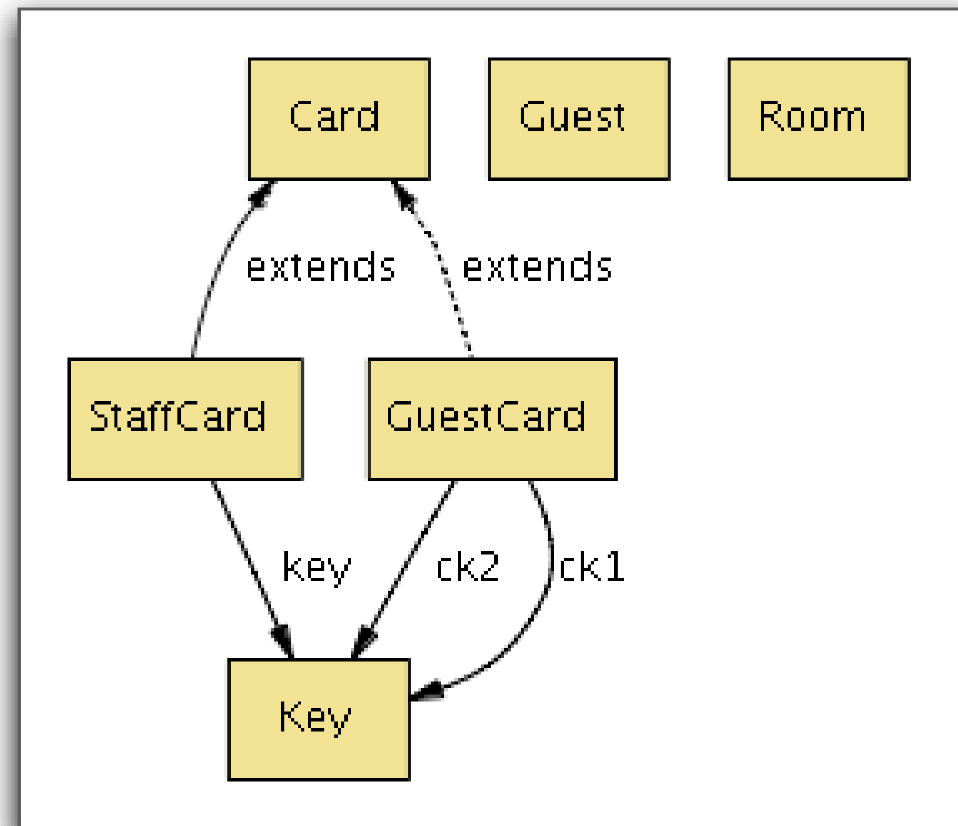
run P
-- instructs analyzer to find instance satisfying facts and Fp

check A
-- instructs analyzer to find instance satisfying facts and **not** Fa

a parade of idioms

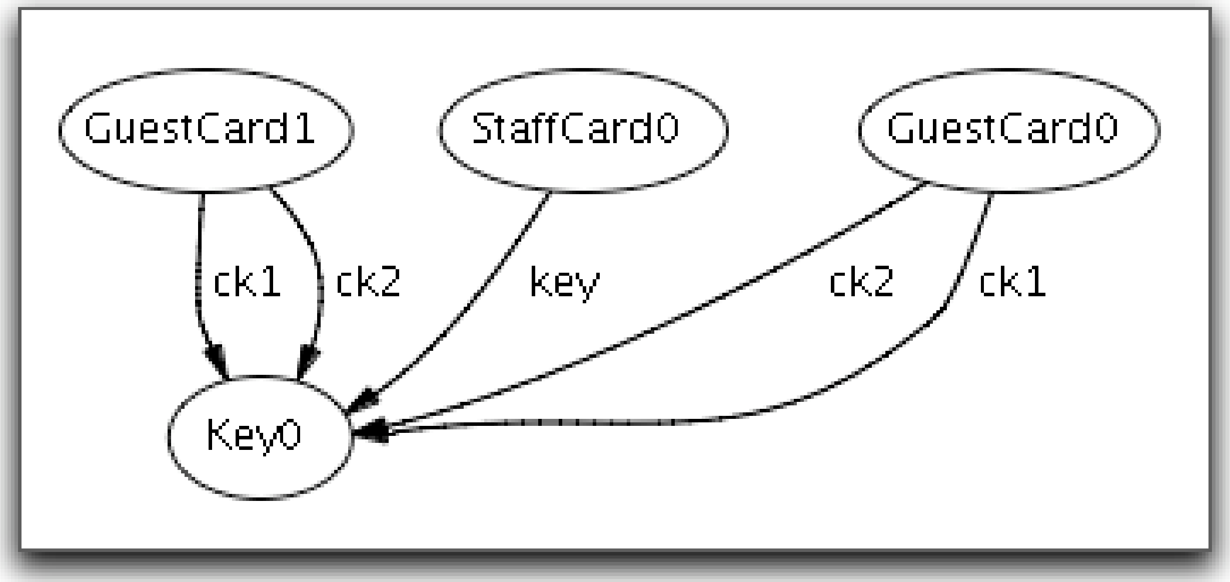
object model, OCL style

```
sig Room, Guest, Key {}
sig Card {}
sig StaffCard extends Card {key: Key}
sig GuestCard extends Card {ck1, ck2: Key}
```



generating an instance

```
pred show () {}
run show
```



state/operation, Z style

```
sig Room, Guest {}
sig State {
    owns: Room -> Guest
    }
pred checkin (s, s': State, r: Room, g: Guest) {
    s'.owns = s.owns + r -> g
    }
run checkin
```

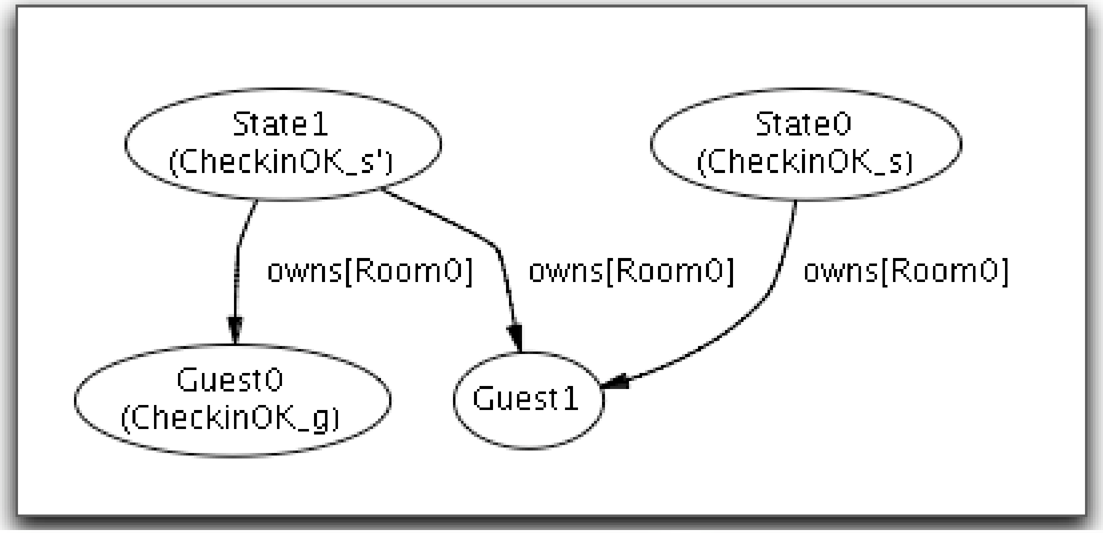
no special interpretation for ‘ mark

checking an invariant

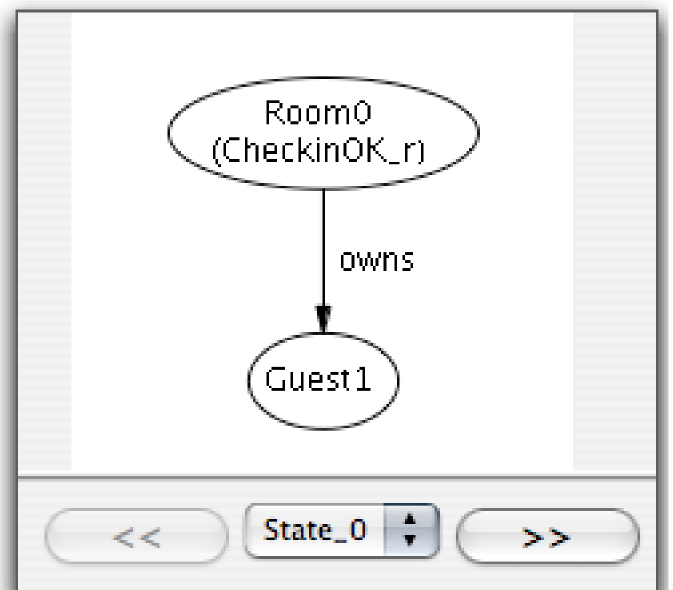
```
pred NoDoubleBooking (s: State) {
   s.owns : Room -> lone Guest
   }

assert CheckinOK {
   all s, s': State, r: Room, g: Guest |
      NoDoubleBooking (s) and checkin (s, s', r, g)
         implies NoDoubleBooking (s')
   }

check CheckinOK for 3 but 2 State
```

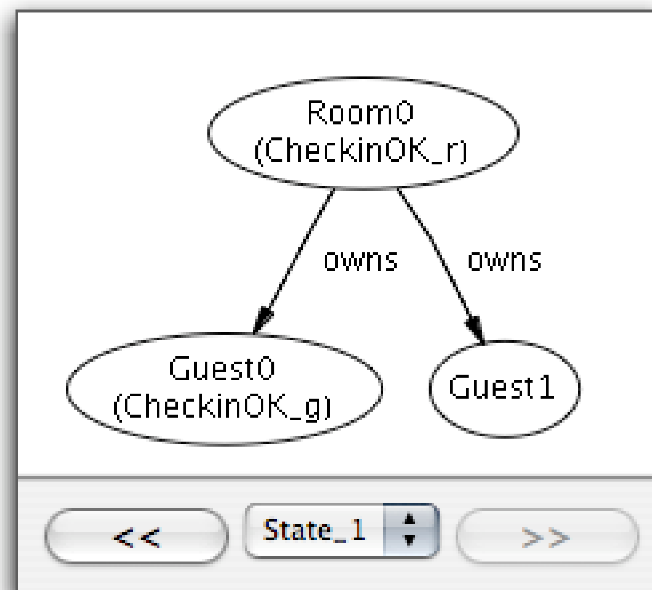
no metalanguage for theorems

counterexample!



default visualization





visualization after 'projecting' *State*

can project *any* type

sig State {owns: Room -> Guest}

implicit precondition, Z style

```
pred checkin (s, s': State, r: Room, g: Guest) {
   no r.(s.owns)
   s'.owns = s.owns + r -> g
   }
```

no *e* means *e* is empty relation

analyzer says:

No counterexample found: CheckinOK is valid within the specified scope. (00:02)

no counterexample ⇒ valid? no!

growing the state, Z style

```
sig State {
    owns: Room -> Guest
    }

sig State1 extends State {
    issued: set Card
    }

pred checkin1 (s, s': State1, r: Room, g: Guest, c: Card) {
    checkin (s, s′, r, g)
    c not in s.issued
    s'.issued = s.issued + c
    }
```

unlike Z schema: **semantic, not syntactic**

can add defined component like this:

```
sig State2 extends State1 {empty: set Room}
    empty = {r: Room | no r.owns}
    }
fact {State2 = State1}
```

reachability, BMC style

```
module traces
open util/ordering [State]
sig Room, Guest {}
sig State {owns: Room -> Guest}

pred init (s: State) {no s.owns}
pred checkin (s, s': State, r: Room, g: Guest) {s'.owns = s.owns + r -> g}

fact traces {
  init (first())
  all s: State - last () |
    some r: Room, g: Guest | checkin (s, next(s), r, g)
  }

assert NoDoubleBooking {
  all s: State | s.owns : Room -> lone Guest
  }
check NoDoubleBooking
```

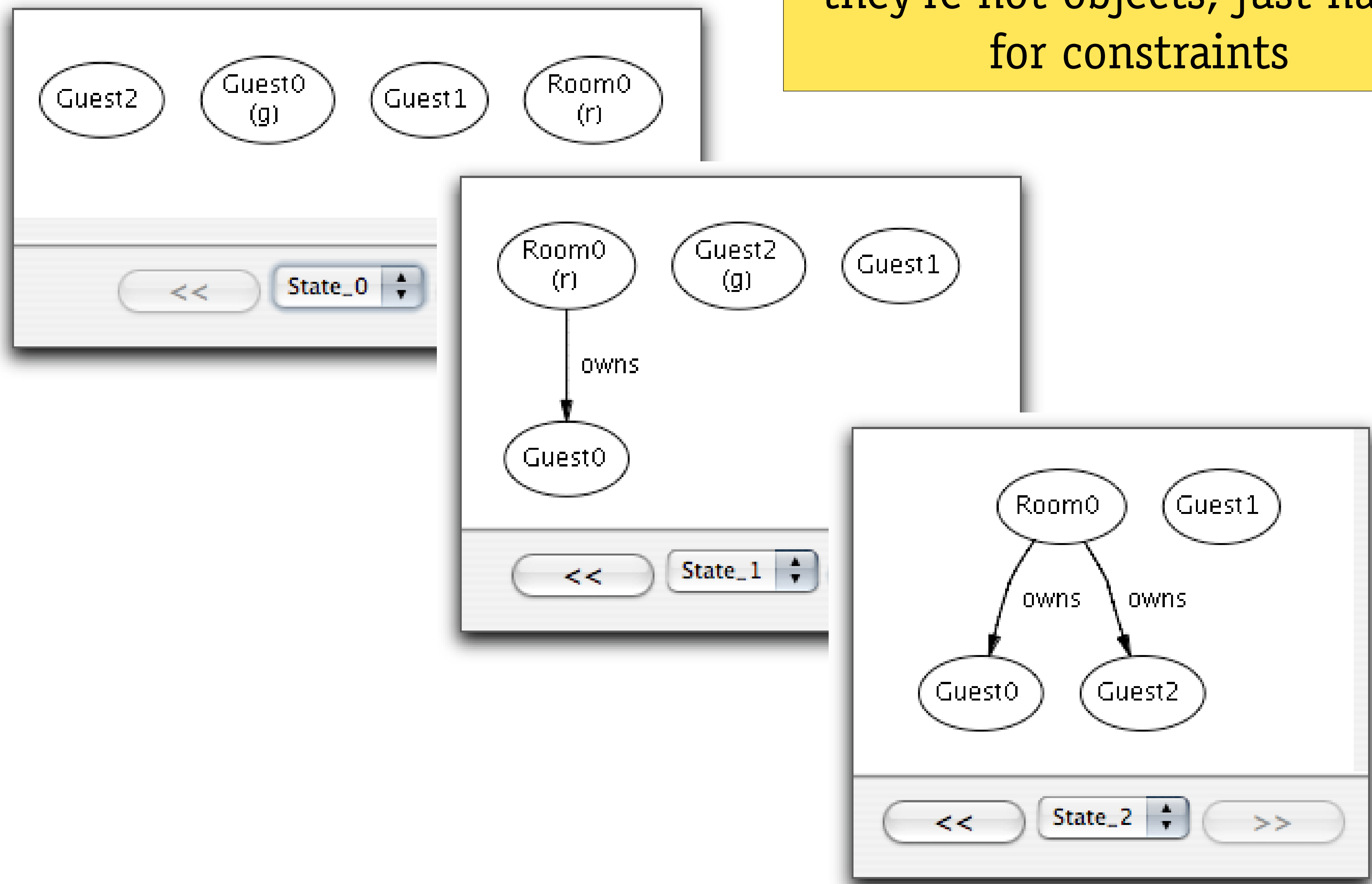
order states with library module

constrain order to satisfy ops

an assertion over *reachable* states

counterexample trace

operation names don't appear: they're not objects, just names for constraints



objects with local state

```
sig Key, Time {}
sig Card {k1, k2: Key}

sig Room {
  k1, k2: Key one -> Time
  }

pred enter (r: Room, c: Card, t, t': Time) {
  c.k1 = r.k2.t
  k1.t' = k1.t ++ r -> c.k1
  k2.t' = k2.t ++ r -> c.k2
  }
```

signatures define local namespaces; overloading resolved automatically

mutable component has Time column

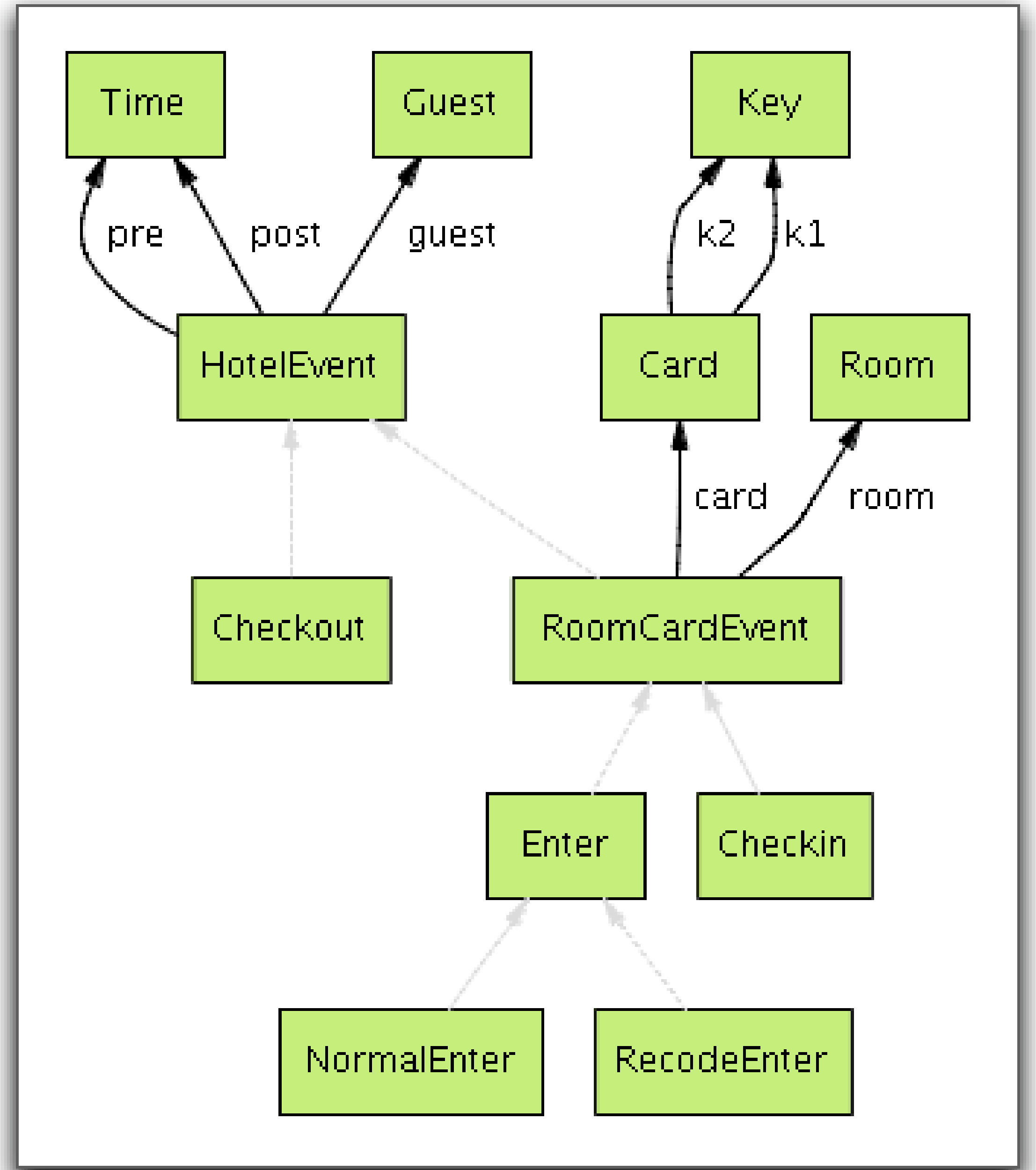
f.t is field f at time t

events as objects

like Z's schema components
and Java's instance variables,
fields of signatures are *free variables*
in extending signature

```
sig Key, Time {}
sig Card {k1, k2: Key}
sig Room {k1, k2: Key one -> Time}
sig Guest {cards: Card -> Time}

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

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

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

sig NormalEnter extends Enter { } { card.k1 = room.k1.pre }

sig RecodeEnter extends Enter { } {
    card.k1 = room.k2.pre
    k1.post = k1.pre ++ room -> card.k1
    k2.post = k2.pre ++ room -> card.k2
    }
```

object model for events



Reiter’s frame conditions

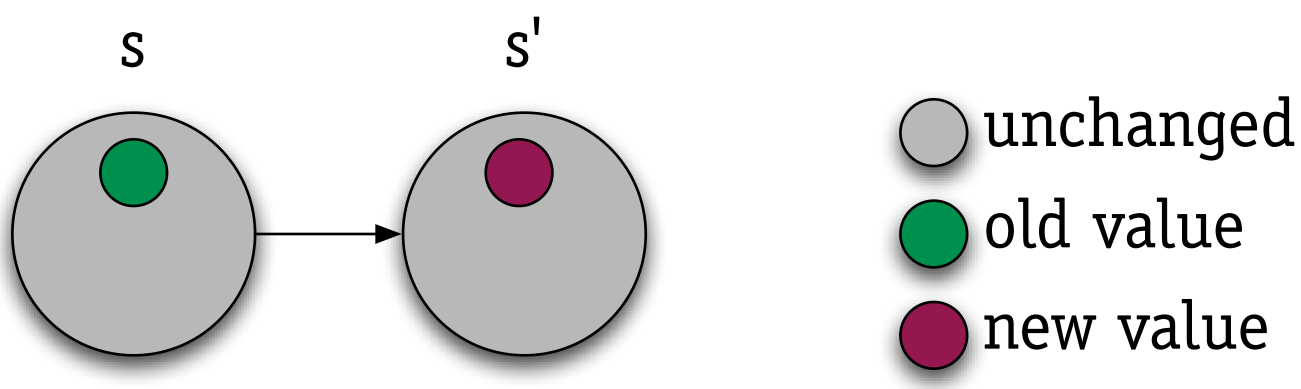
in declarative models

› 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.

frame conditions, Reiter style

```
fact Traces {
  all t: Time - last () | let t' = next (t) |
    some e: HotelEvent {
      e.pre = t and e.post = t'
      k1.t = k1.t' and k2.t = k2.t' or e in RecodeEnter
      issued.t = issued.t' and cards.t = cards.t' or e in Checkin
      owns.t = owns.t' or e in Checkin + Checkout
      }
  }
```

if k1 or k2 changed, then
RecodeEnter must have happened

a safety assertion

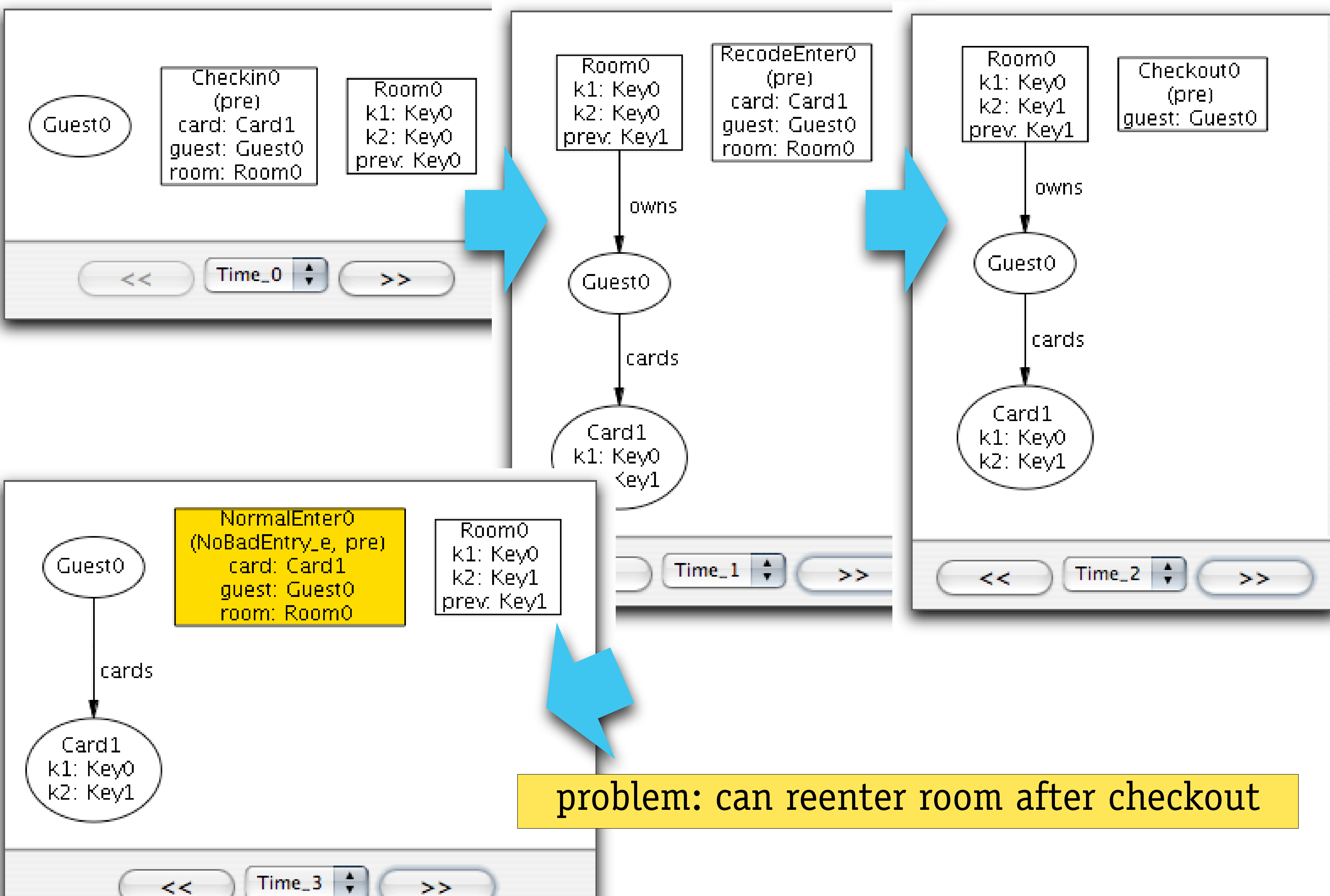
safety condition

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

```
assert NoBadEntry {
  all e: Enter | e.guest in e.room.owns.(e.pre)
  }
```

this assertion is about events, and is not expressible in purely state-based formalisms

counterexample!



problem: can reenter room after checkout

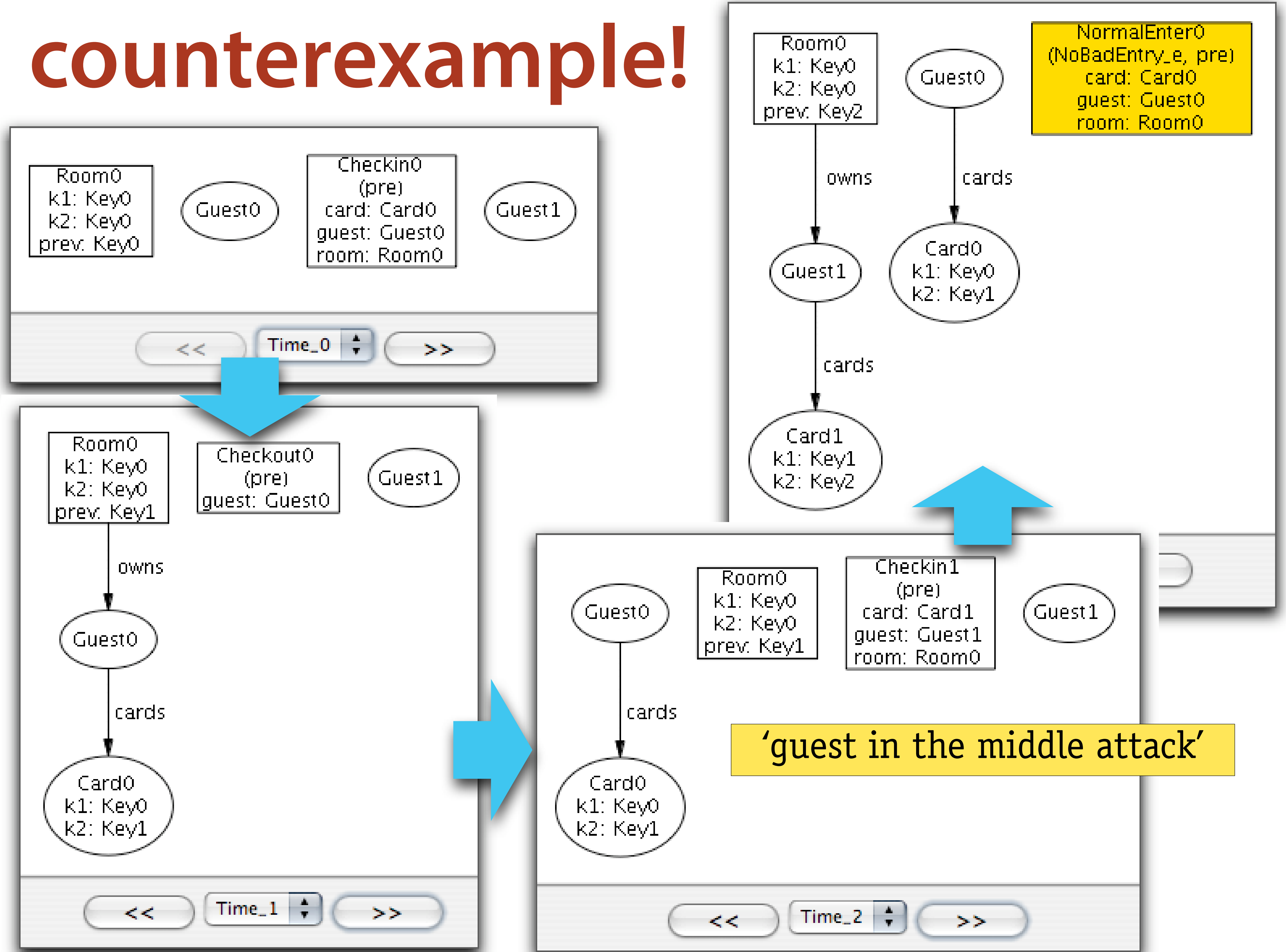
a relaxed safety assertion

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 owners = e.r.owns.(e.pre) |
      some owners => e.g in owners
  }
```

counterexample!



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
         }
   }
```

conclusions

how to be safe in a hotel

don't let the bellboy open your door!

- must open it yourself to satisfy NoIntervening

pluralistic modelling

Alloy supports a wide range of idioms and styles

good for teaching
- what you see is what you get
- simple underlying logic
- all analysis is model finding

good for research
- can experiment easily with new idioms

good for practice
- can tailor idiom to the problem
- example: Jazayeri's model of Apple's Bonjour
  mentioned 'two states ago'

hotel locking case study

contributions in my book from

- Martin Gogolla (OCL)
- Jim Woodcock (Z)
- Peter Gorm Larsen and John Fitzgerald (VDM)
- Michael Butler (B)

chapter available at http://softwareabstractions.org/

recently also by Tobias Nipkow in Isabelle

- proves safety for weaker (localized) condition
- shows equivalence of trace- and state-based models

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

alloy.mit.edu

- downloads, papers, tutorial

alloy@mit.edu

- questions about Alloy
- send us a challenge

dnj@mit.edu

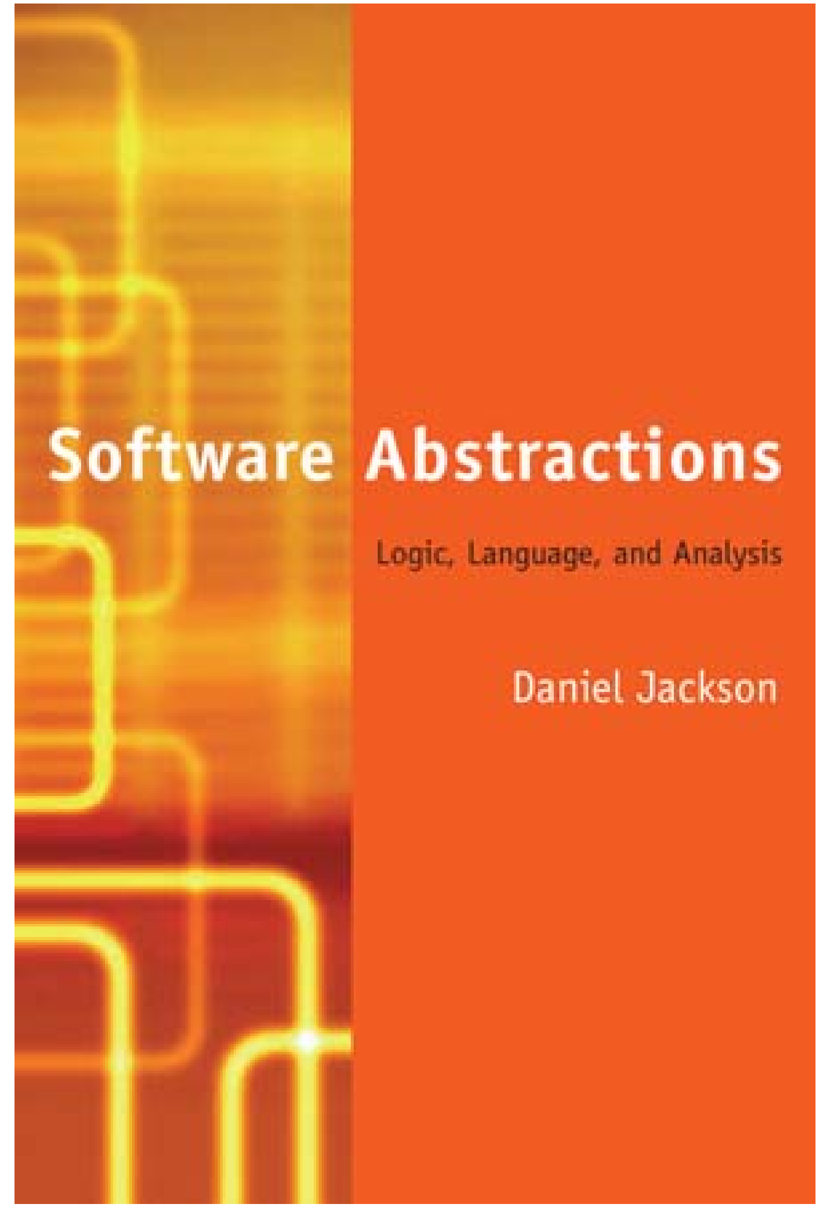
- happy to hear from you!

mit.edu/people/emina/kodkod.html

- Alloy as an API

Software Abstractions

- MIT Press, March 2006
- discount available to ICGT/SBMF



spare slides: evaluation of Alloy

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
- Mondex electronic purse

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

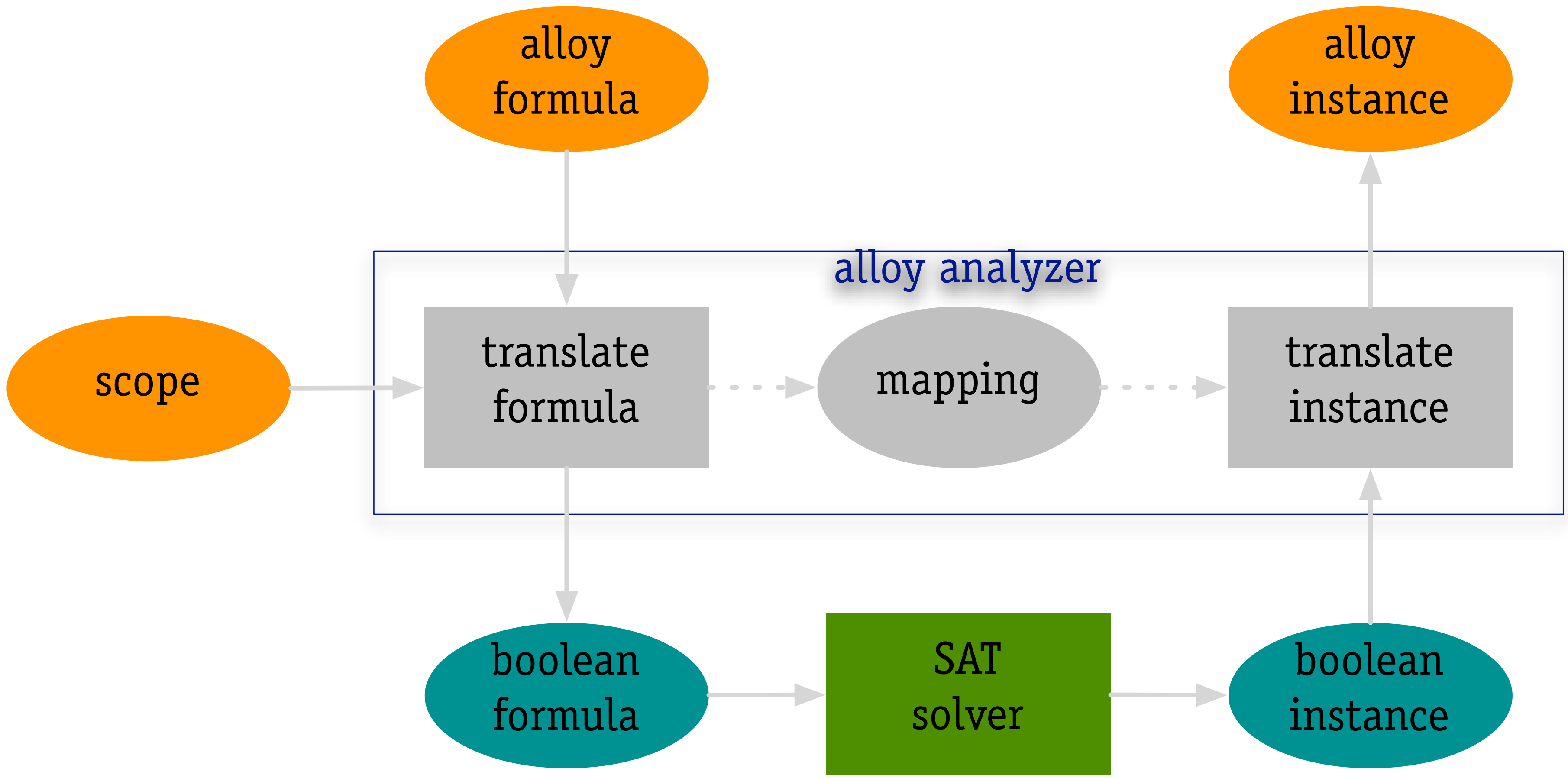
relations aren't a panacea
- sequences are awkward
- treatment of integers limited

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

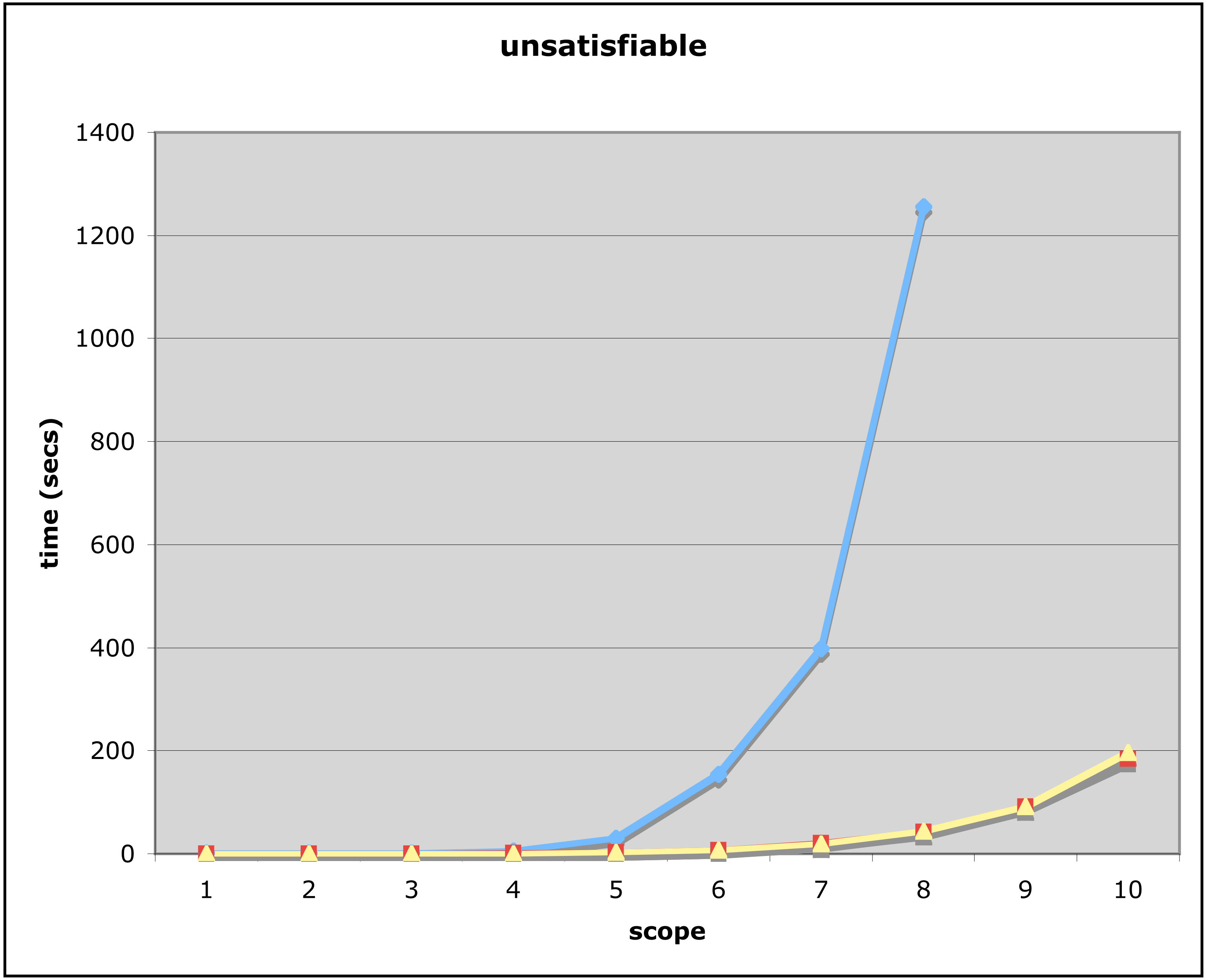
limitations of language
- module system doesn't offer real encapsulation

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

alloy analyzer architecture



performance (unsat)



performance (sat)