AN ADVANCED INTRODUCTION TO ALLOY

Daniel Jackson · ASE'07 · Atlanta · Nov 8, 2007

CSAIL
MIT COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE LABORATORY

introduction

premises

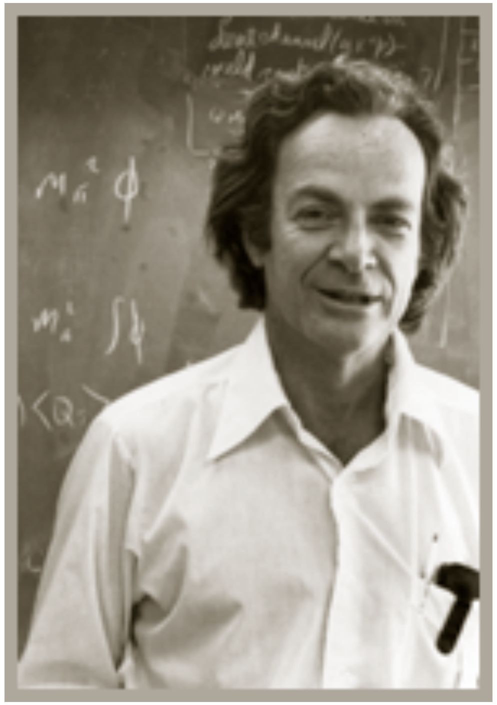
software development needs

- 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

what motivated Alloy?

an expressive, natural syntax [from object modelling]

- classification hierarchies, multiplicities, navigations

a simple semantics [from Z, conceptual data modelling, Tarski]

- relations and sets, behaviour as constraints

a powerful analysis [from model checking]

- full automation, sound counterexamples

transatlantic alloy

transatlantic alloy

Oxford, home of Z

transatlantic alloy

Oxford, home of Z

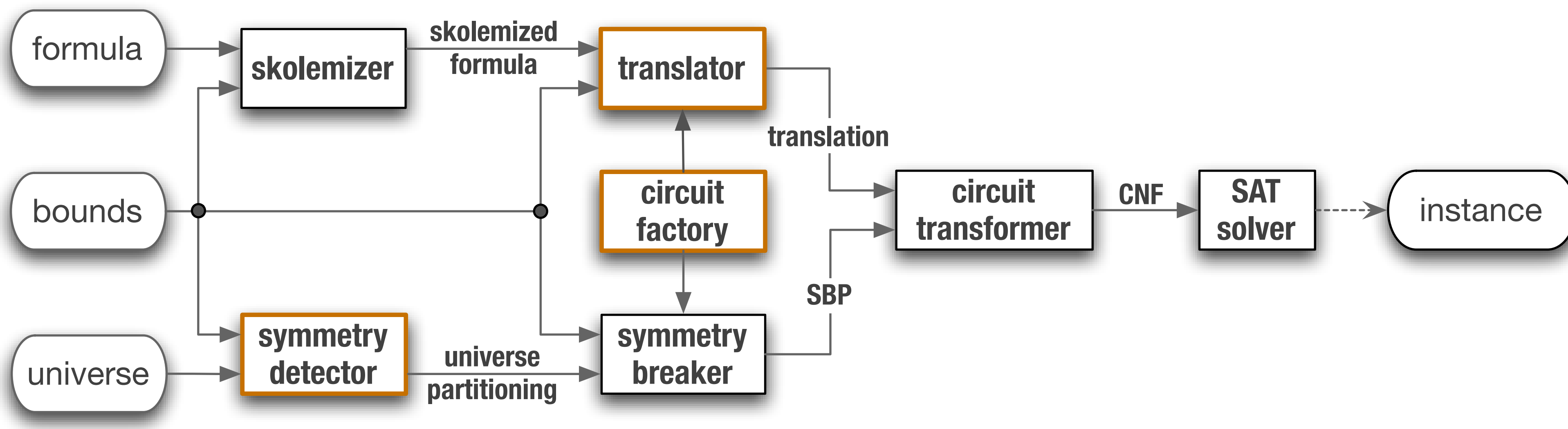
Pittsburgh, home of SMV

Alloy features: true confessions

relational structures, not just trees	sequences are second-class citizens
first-order logic with join & closure	limited higher-order logic
supports fully declarative specification	no loop construct; frame conditions
subtypes, union types, overloading	module system a bit clunky
symbolic model finding	state space limited to a few 100 bits
coverage analysis	experimental feature
basic arithmetic constraints	limited scalability
refinement checks, temporal checks	no temporal logic

what's new in Alloy (since 2006)

Kodkod, a new engine*



unsat core: a new coverage analysis

improved visualizer: new interface, magic layout

language: better overloading, sequences, syntactic shorthands

supported API: for use of Alloy as a backend

*http://web.mit.edu/emina/www/kodkod.html

plan for today's tutorial

Alloy by example

- some comments on formal semantics only at the end

3 styles of modelling

- static object modelling (cf. OCL)
- state/operation modelling (cf. Z)
- event modelling

essence of Alloy

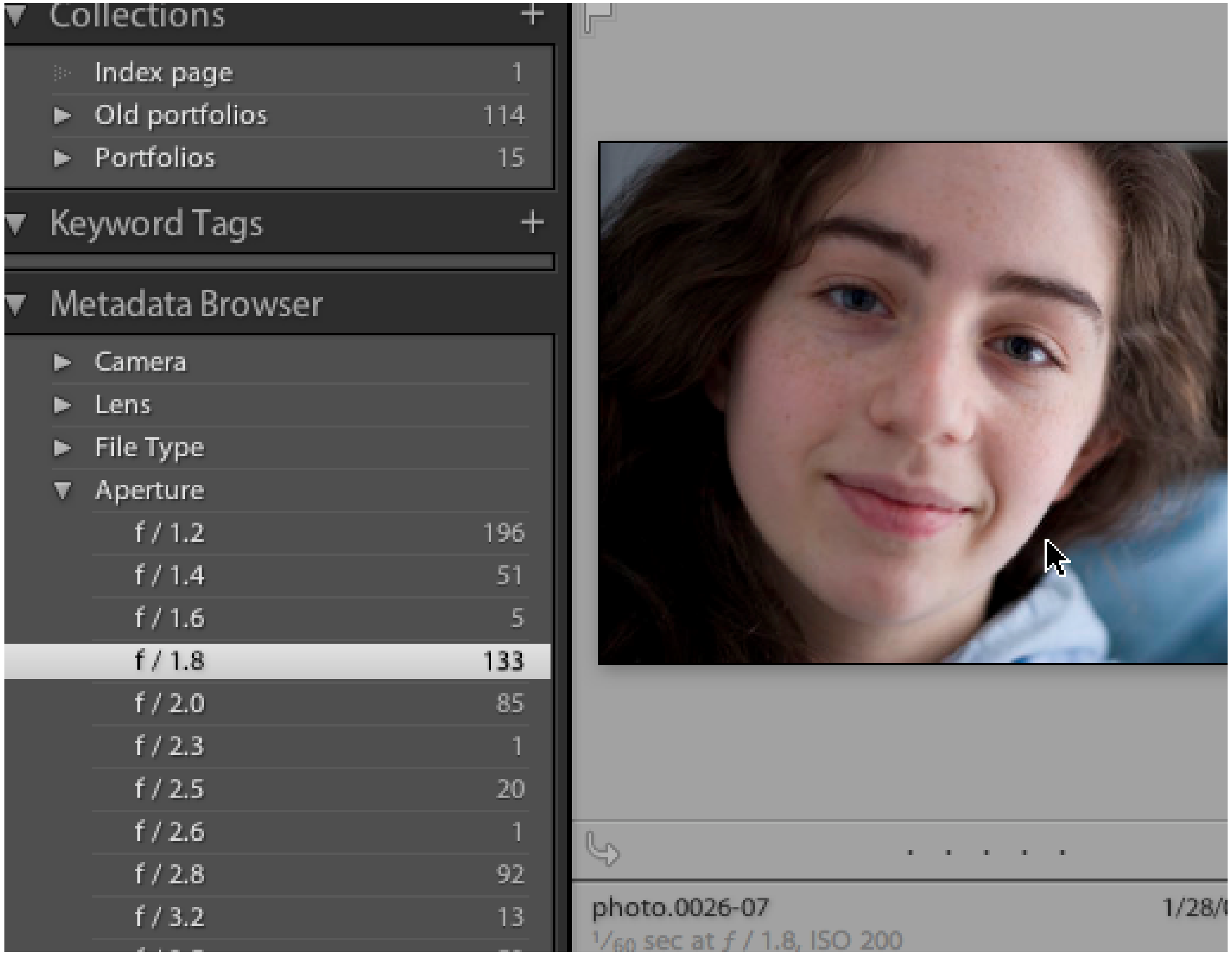
- what makes Alloy tick?

static object modelling

problem: EXIF filtering

design a scheme for

- storing EXIF data for photos
- displaying photos matching a filter



objects & classification

our first Alloy model!

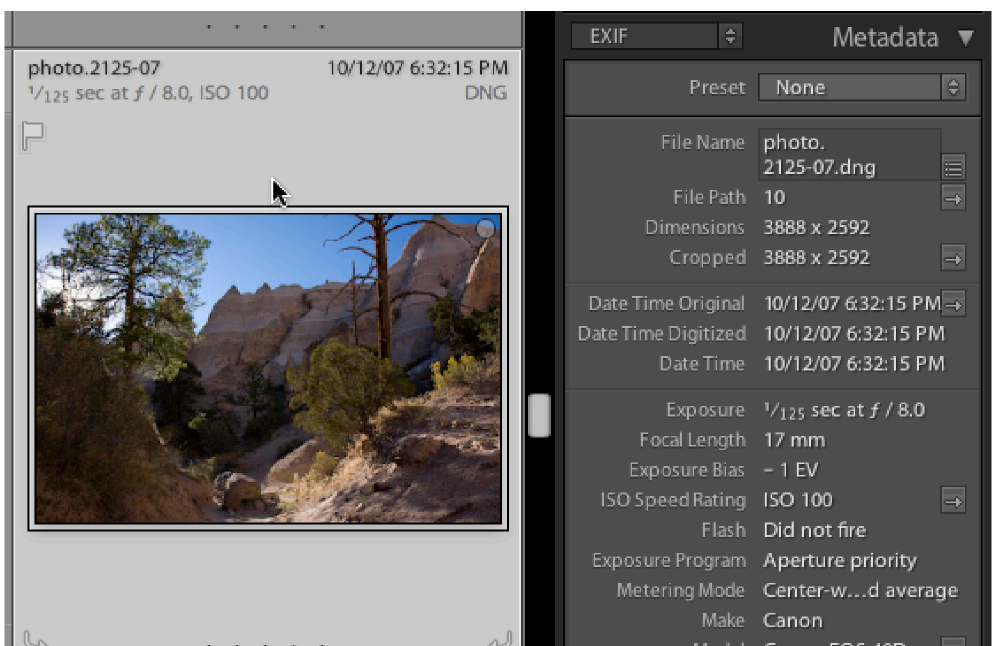
```
sig Photo { file: File, tags: set Tag }
sig Tag { key: Key, val: Value }

sig File {}

abstract sig Key {}
one sig Aperture, FocalLength, ShutterSpeed extends Key {}

abstract sig Value {}
sig Number, String, Date extends Value {}
```

sig: declares set of objects

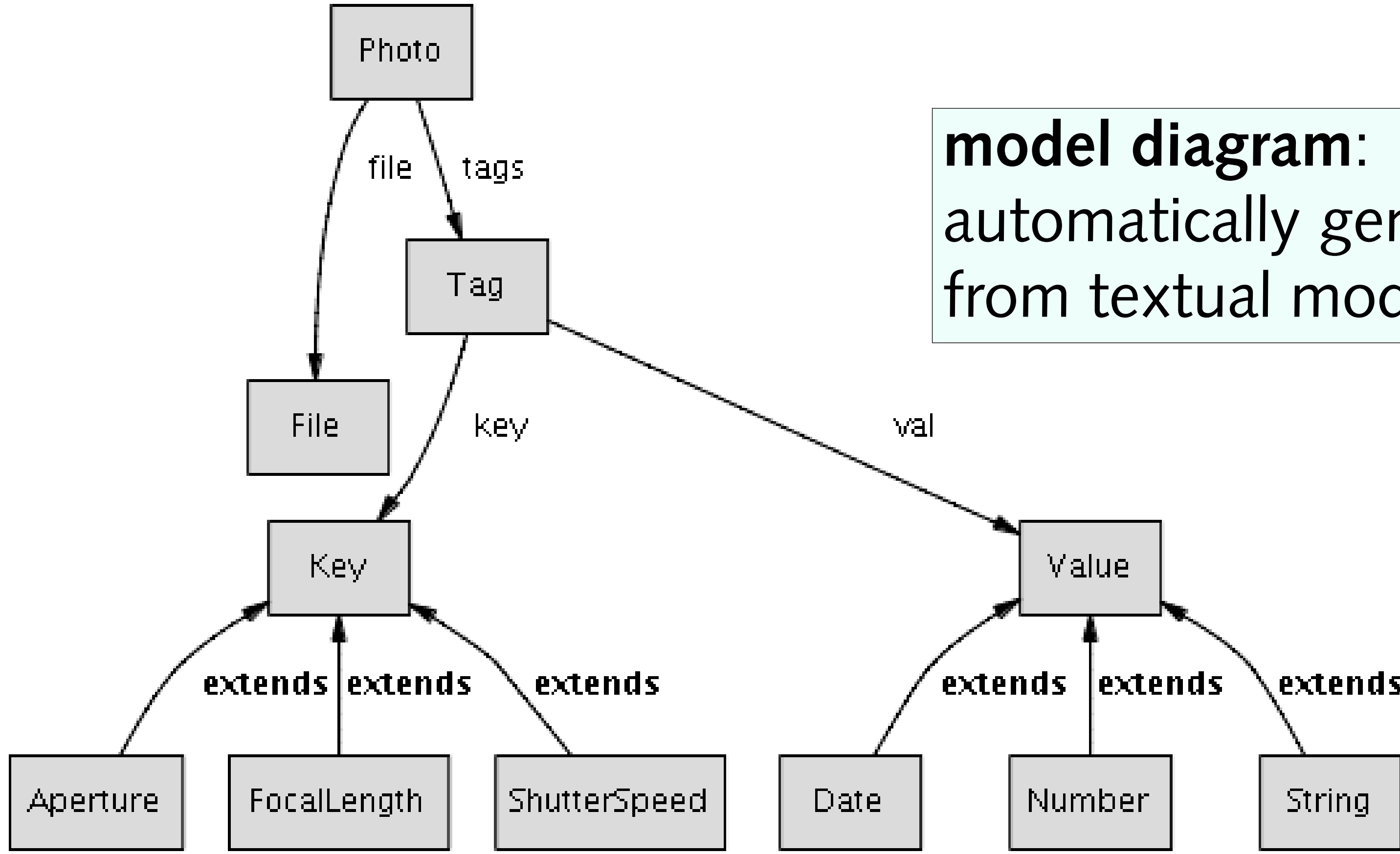
field: relation, association

extends: subset

abstract: subsets exhaust



model diagram

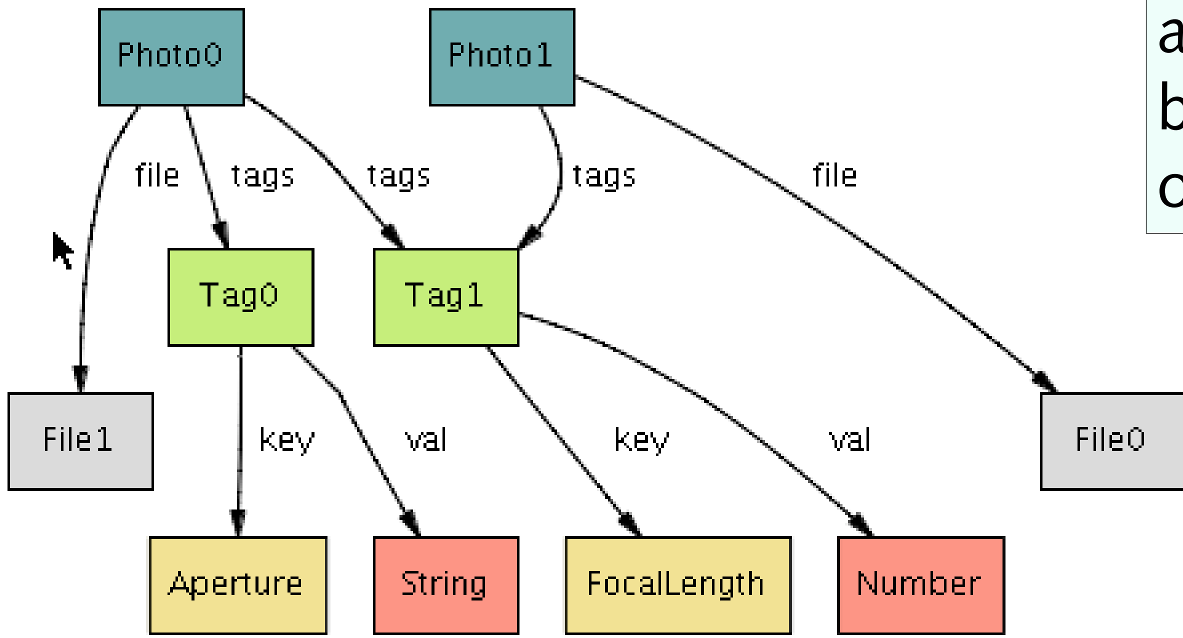


model diagram: automatically generated from textual model

sig Photo { file: File, tags: **set** Tag }
sig Tag { key: Key, val: Value }
sig File {}
abstract sig Key {}
one sig Aperture, FocalLength, ShutterSpeed **extends** Key {}
abstract sig Value {}
sig Number, String, Date **extends** Value {}

an example

example, instance: automatically generated by solver, in an arbitrary order




```
sig Photo { file: File, tags: set Tag }
sig Tag { key: Key, val: Value }
sig File {}
abstract sig Key {}
one sig Aperture, FocalLength, ShutterSpeed extends Key {}
abstract sig Value {}
sig Number, String, Date extends Value {}

run{}
```

navigations

fact: records assumption, always holds

no, some, one, all: quantifiers on formulas and expressions (bar **all**)

p.tags: navigation expr

&, +, -, in: set operators

let's say photos don't share files or tags

```
fact NoSharing {
    no disj p, p': Photo |
        p.file = p'.file or some p.tags & p'.tags
    }
```

and that tags are typed

```
fact TypedTags {
    all t: Tag | t.key in Aperture implies t.val in Number
    ...
    }
```

checking a reformulation

two ways to say no tag sharing:

```
pred NoTagShare {
    no disj p, p′: Photo |
        some p.tags & p′.tags
    }
pred NoTagShare′{
    all t: Tag | #t.~tags > 1
    }
```

t.~tags: backwards navigation

#e: cardinality

ask analyzer if they're the same:

```
check {NoTagShare iff NoTagShare′}
```

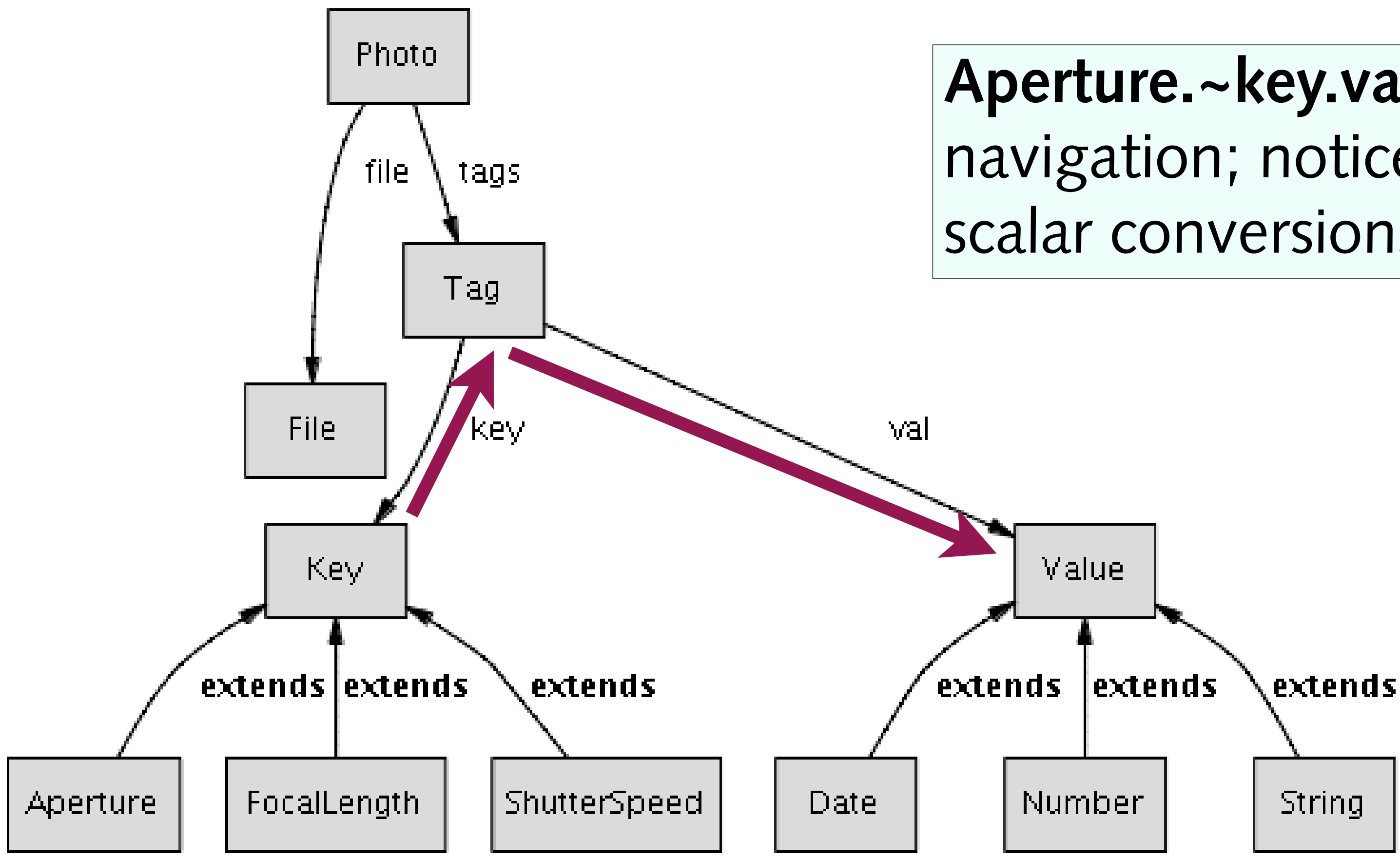
check: asks solver to generate counterexample to **assertion**

```
Executing "Check assert$1 for 3"
  Solver=minisatprover(jni) Bitwidth=4 MaxSeq=3 Symmetry=20
  516 vars. 54 primary vars. 818 clauses. 47ms.
  No counterexample found. Assertion may be valid. 128ms.
```

another reformulation

two ways to say that tags are typed:

pred TypedTag {**all** t: Tag | t.key **in** Aperture **implies** t.val **in** Number}
pred TypedTag'{ Aperture.~key.val in Number}



Aperture.~key.value: multistep navigation; notice no set/ scalar conversions

inherited fields

key: Key in sig decl of **BasicFilter** means **b.key** is a scalar, for every **b** in **BasicFilter**

basic and compound filters

```
abstract sig Filter {matches: set Photo}

abstract sig BasicFilter extends Filter {key: Key, range: set Value}
sig NumFilter extends BasicFilter {}{range in Number}
sig StringFilter extends BasicFilter {}{range in String}
sig DateFilter extends BasicFilter {}{range in Date}

abstract sig CompoundFilter extends Filter {filters: some Filter}
sig AndFilter extends CompoundFilter {}
sig OrFilter extends CompoundFilter {}
```

some: multiplicity symbol

signature facts: implicitly quantified over all objects in sig, just like receivers in Java

type checking

```
abstract sig Filter {matches: set Photo}

abstract sig BasicFilter extends Filter {key: Key, range: set Value}
sig NumFilter extends BasicFilter {}{range in Number}
sig StringFilter extends BasicFilter {}{range in String}
sig DateFilter extends BasicFilter {}{range in Date}

abstract sig CompoundFilter extends Filter {filters: some Filter}
sig AndFilter extends CompoundFilter {}
sig OrFilter extends CompoundFilter {}
```

no false alarms: and type system consistent with modelling

```
all f: Filter | some f.key implies f in BasicFilter
```

```
all b: BasicFilter | some b.filters
```

Warning #1
The join operation here always yields an empty set.
Left type = {this/BasicFilter}
Right type = {this/CompoundFilter->this/Filter}

overloading resolution

```
sig Tag {
    key: Key,
    val: Value
    }
abstract sig BasicFilter extends Filter {
    key: Key,
    range: set Value
    }

all disj t, t': Tag | t.~tags = t'.~tags implies t.key != t'.key

all k: Key | some k.~key

all k: Key | some k.~key & Tag
```

resolvent is determined automatically from context

unlike Java, not limited to resolution based on argument to left of dot

recursive definitions

set comprehensions,
standard meaning

```
fact Matching {
  all f: Filter | f.matches = (
      f in BasicFilter then
        {p: Photo | all t: p.tags | t.key = f.key implies t.val in f.range}
      else f in AndFilter then {p: Photo | all cf: f.filters | p in cf.matches}
      else f in OrFilter then {p: Photo | some cf: f.filters | p in cf.matches}
      else none)
  }
```

checking a theorem

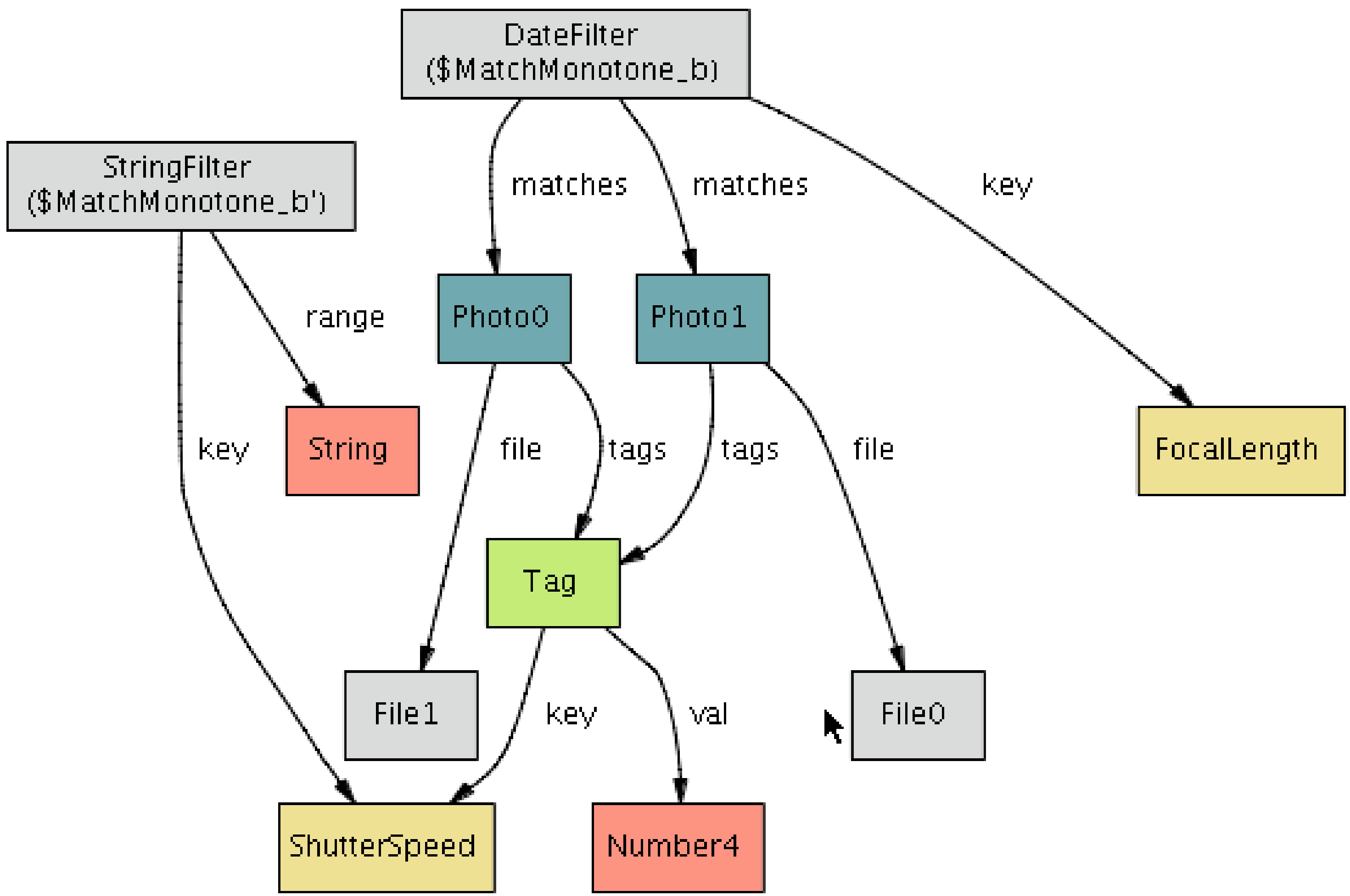
```
assert MatchMonotonic {
    all b, b': BasicFilter |
        b.range in b'.range
            implies b.matches in b'.matches
    }
check MatchMonotone for 10 but 2 Filter
```

assert: just declares assertion

check: instructs solver to check assertion in given 'scope'

counterexample!

Executing "Check MatchMonotone for 10 but 2 Filter"
Solver=minisat(jni) Bitwidth=4 MaxSeq=7 Symmetry=20
6049 vars. 460 primary vars. 12956 clauses. 113ms.
Counterexample found. Assertion is invalid. **50ms.**



checking a theorem, again

```
assert MatchMonotonic {
    all b, b': BasicFilter |
        b.range in b'.range and b.key = b'.key
            implies b.matches in b'.matches
    }
check MatchMonotone for 10 but 2 Filter
```

assert: just declares assertion

check: instructs solver to check assertion in given ‘scope’

```
Executing "Check MatchMonotone for 10 but 2 Filter"
  Solver=minisat(jni) Bitwidth=4 MaxSeq=7 Symmetry=20
  6073 vars. 460 primary vars. 13022 clauses. 139ms.
  No counterexample found. Assertion may be valid. 249ms.
```

state/operation modelling

simple memory

```
sig Memory {
    data: Addr -> lone Data
    }
pred init (m: Memory) {no m.data}
pred write (m, m': Memory, a: Addr, d: Data) {
    m'.data = m.data ++ a -> d
    }
pred read (m: Memory, a: Addr, d: Data) {
    some m.data [a] implies d = m.data [a]
    }
```

predicate: a parameterized constraint

++: relational override

a->d: the tuple/relation that maps **a** to **d**

m.data: a relation

m.data[a]: [] is lookup

cache memory

```
sig CacheSystem {
	main, cache: Addr -> lone Data
	}
pred init (c: CacheSystem) {no c.main + c.cache}
pred write (c, c': CacheSystem, a: Addr, d: Data) {
	c'.main = c.main
	c'.cache = c.cache ++ a -> d
	}
pred CacheSystem.read (a: Addr, d: Data) {
	some d and d = this.cache [a]
	}
```

this: alternative syntax, first argument treated as receiver

load and flush

```
pred load [c, c': CacheSystem] {
    some entries: c.main |
        c'.cache = c.cache + entries
    c'.main = c.main
    }

pred flush [c, c': CacheSystem] {
    some entries: c.cache {
        c'.main = c.main ++ entries
        c'.cache = c.cache - entries
        }
    }
```

higher-order: OK if existential: **entries** ranges over relational subsets of **c.main**

declarative spec: try saying this in an operational notation

checking refinement

```
fun alpha (c: CacheSystem): Memory {
    {m: Memory | m.data = c.main ++ c.cache}
    }

ReadOK: check {
    all c: CacheSystem, a: Addr, d: Data |
        c.read[a, d] implies c.alpha.read[ a, d]
    } for 2 but 10 Addr, 10 Data

FlushOK: check {
    all c, c': CacheSystem |
        flush[c, c'] implies c.alpha = c'.alpha
    } for 2 but 10 Addr, 10 Data
```

abstraction function: maps cache system to memory

refinement check: not a special feature; just finding counterexample of assertion

```
Executing "Check WriteOK for 2 but 10 Addr, 10 Data"
   Solver=minisat(jni) Bitwidth=4 MaxSeq=2 Symmetry=20
   10003 vars. 652 primary vars. 23157 clauses. 115ms.
   No counterexample found. Assertion may be valid. 164ms.
```

event modelling

hotel locking

recodable locks (since 1980)

- new guest gets a different key
- lock is ‘recoded’ to new key
- last guest can no longer enter

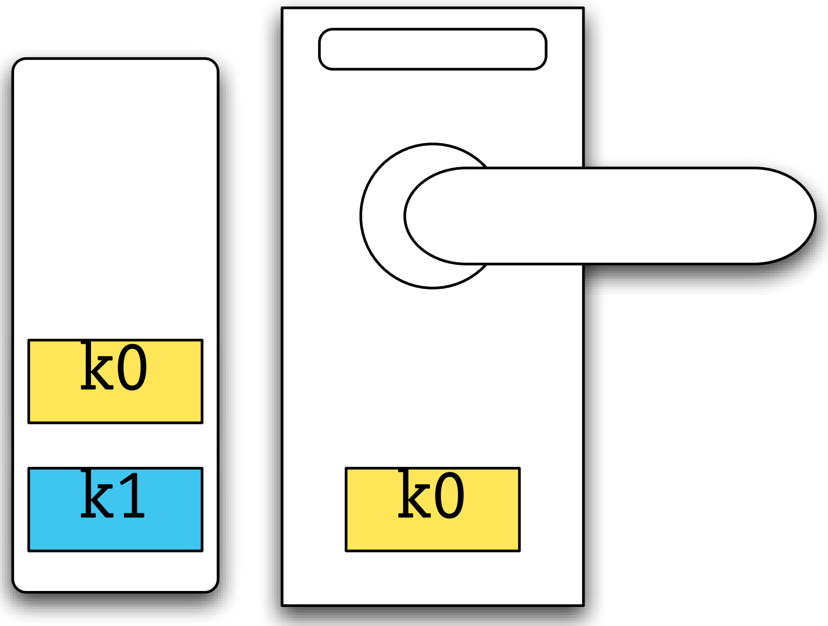
how does it work?

- locks are standalone, not wired

a recodable locking scheme

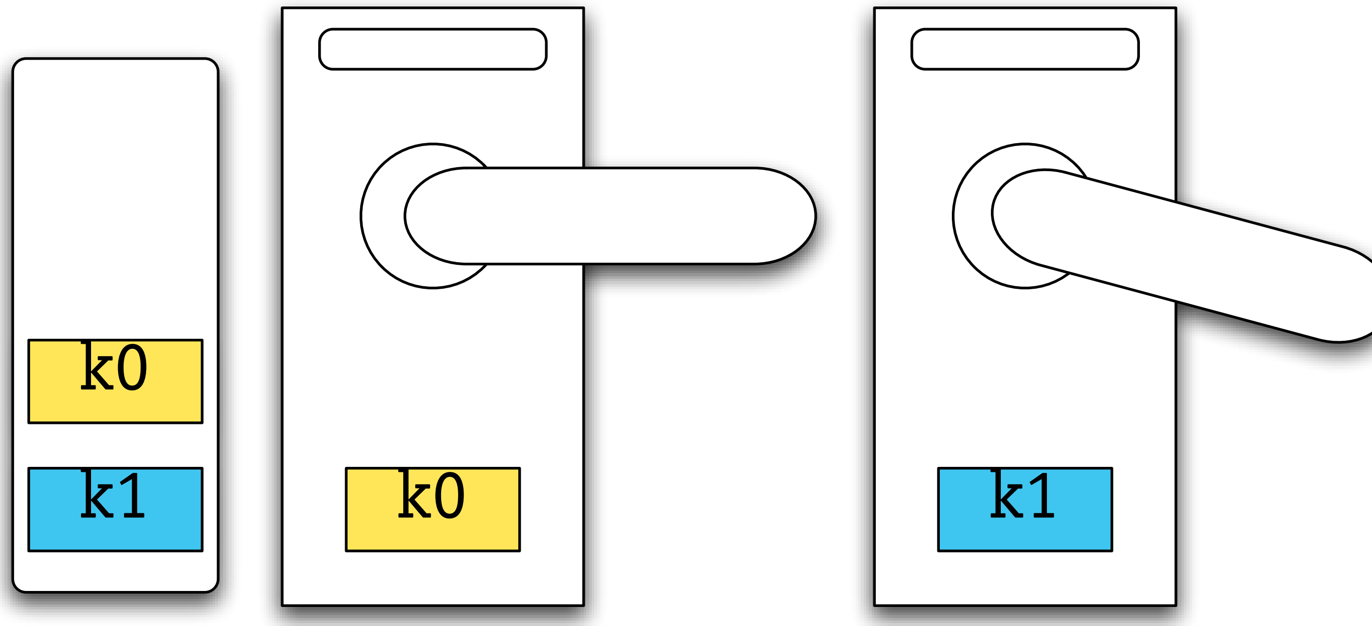
a recodable locking scheme

card has two keys
if first matches lock,
recode with second



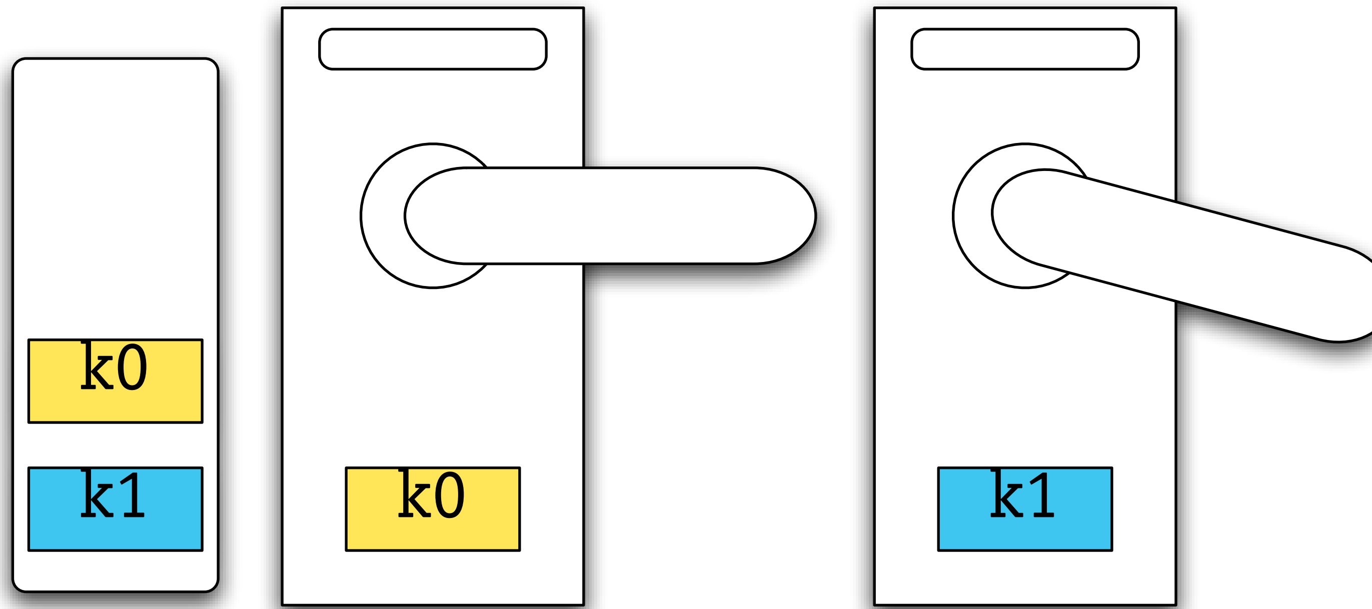
a recodable locking scheme

card has two keys
if first matches lock,
recode with second

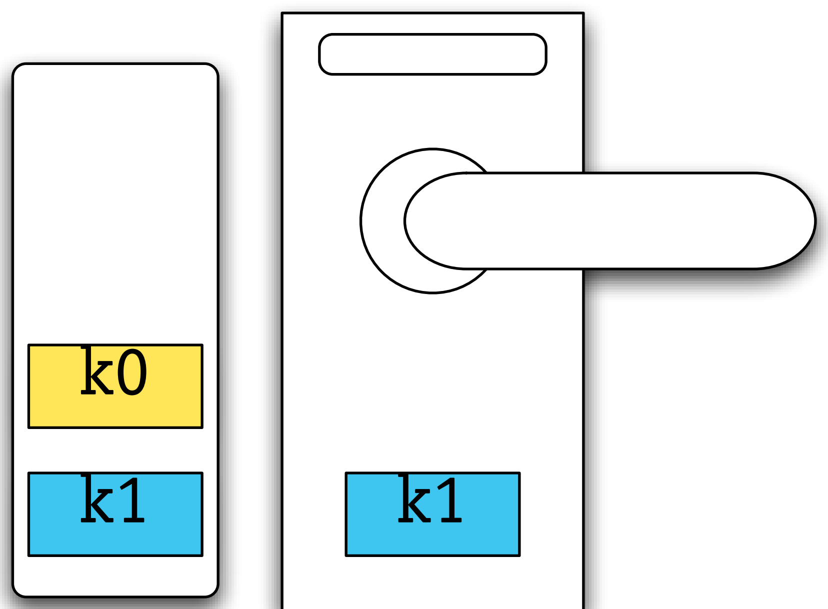


a recodable locking scheme

card has two keys
if first matches lock,
recode with second



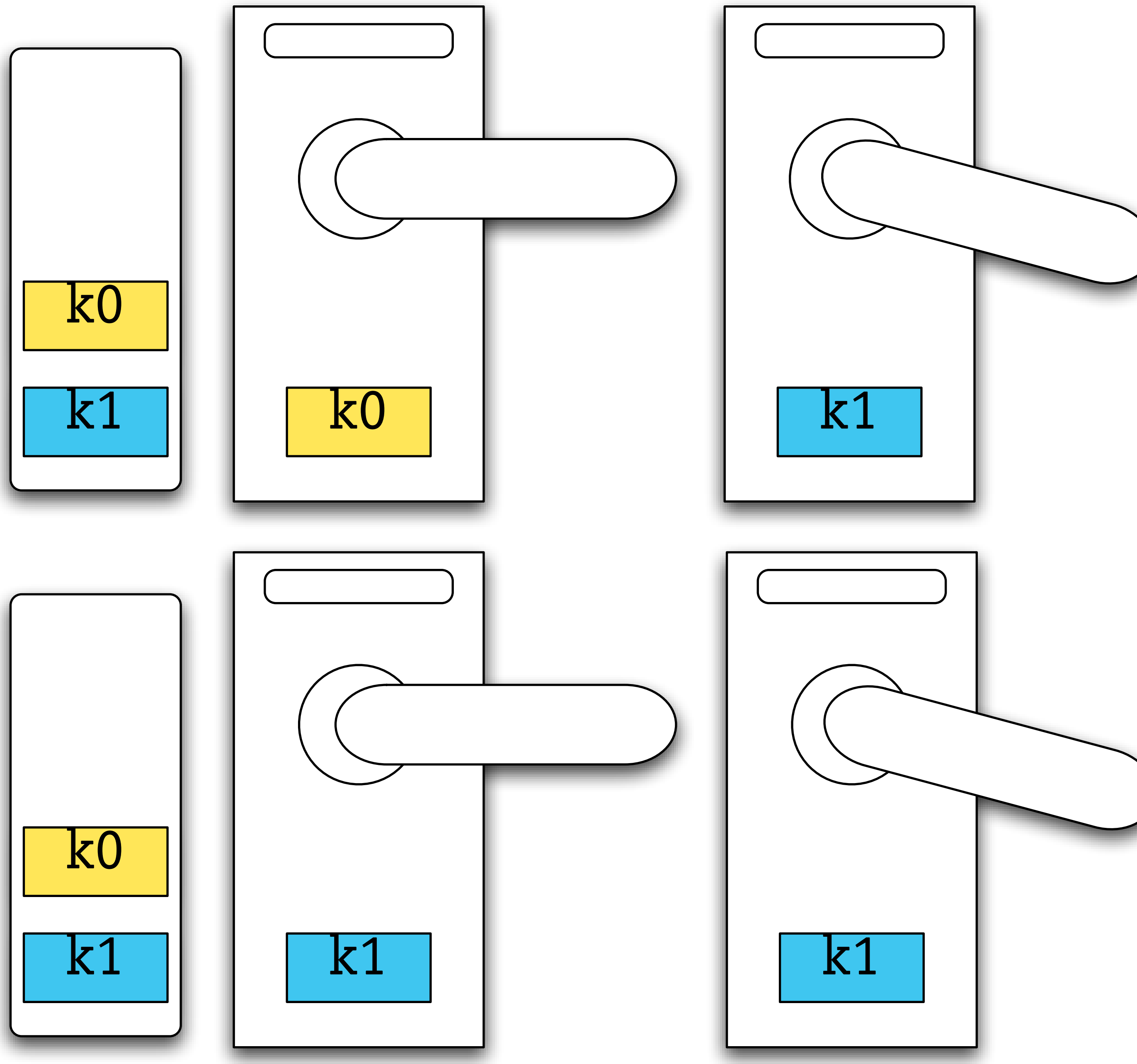
if second matches,
just open



a recodable locking scheme

card has two keys
if first matches lock,
recode with second

if second matches,
just open



event framework

module system: simple parametric polymorphism

util/ordering: library module with built-in symmetry breaking

events and traces: no built-in idiom, so you can roll your own

```
module events
open util/ordering[Time] as time
sig Time {}
abstract sig Event {
   pre, post: Time
   }
fact Traces {
   all t: Time - last | one e: Event | e.pre = t and e.post = t.next
   }
pred Event.unchanged (field: univ -> Time) {
   field.(this.pre) = field.(this.post)
   }
```

local state

```
module hotel
open events

sig Key {}

sig Card {k1, k2: Key}
-- c.k1 is first key of card c

sig Guest {
   holds: Card -> Time
   }
-- g.holds.t is set of cards g holds at time t

sig Room {
   key: Key one -> Time,
   prev:  Key lone -> Time,
   occ: Guest -> Time
   }
-- r.key.t is key of room r at time t
```

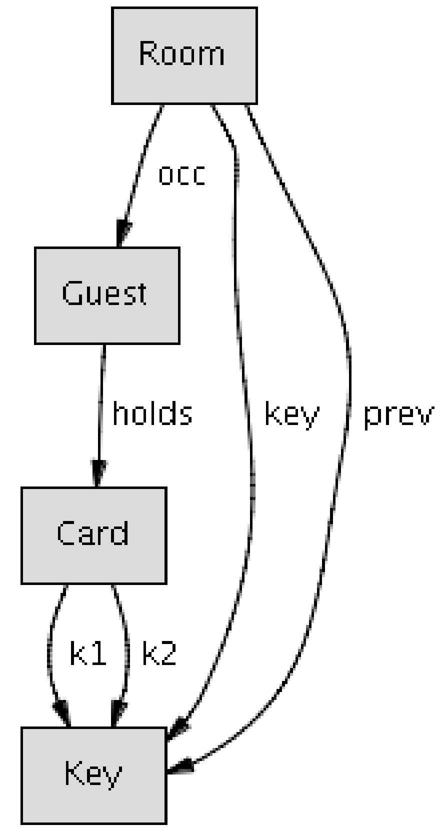
r.s: denotes set of objects that **r** maps to members of set **s**

projection: model diagram is projected over **Time**

local state

```
module hotel
open events

sig Key {}

sig Card {k1, k2: Key}
-- c.k1 is first key of card c

sig Guest {
   holds: Card -> Time
   }
-- g.holds.t is set of cards g holds at time t

sig Room {
   key: Key one -> Time,
   prev:  Key lone -> Time,
   occ: Guest -> Time
   }
-- r.key.t is key of room r at time t
```

r.s: denotes set of objects that **r** maps to members of set **s**



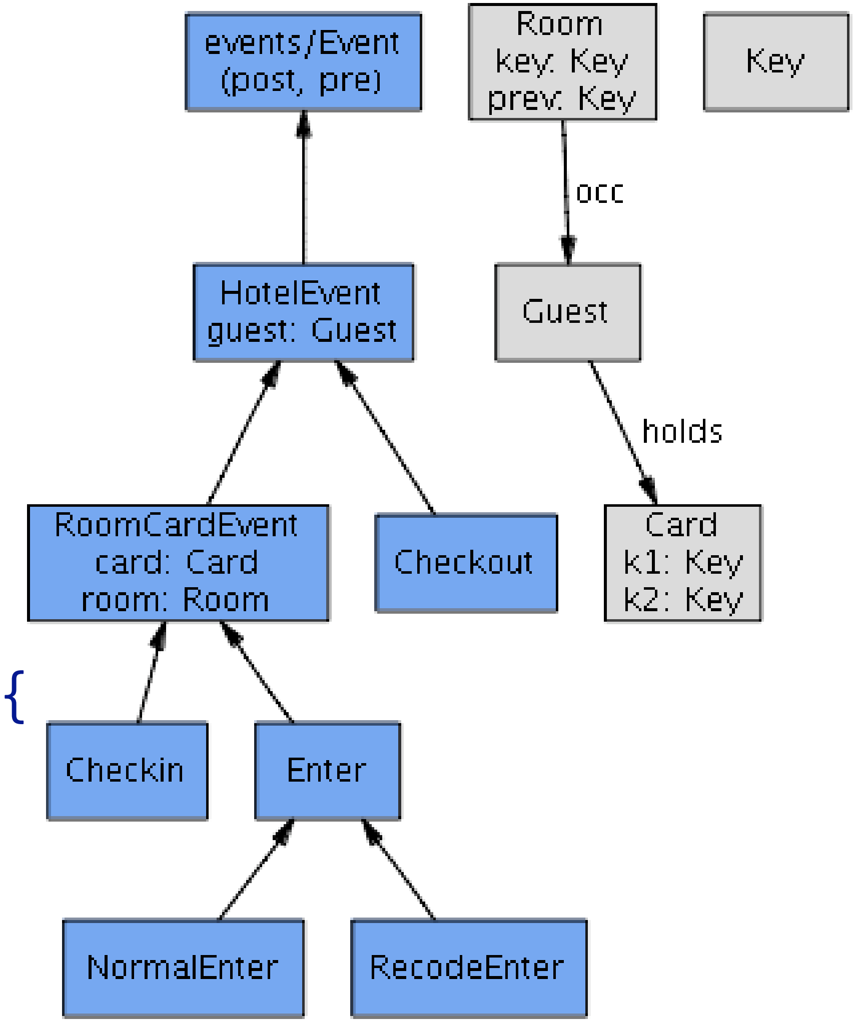
projection: model diagram is projected over **Time**

events as objects

```
abstract sig Event {
  pre, post: Time }
  }

abstract sig HotelEvent extends Event {
  guest: Guest
  }

sig Checkout extends HotelEvent { }

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

sig Checkin extends RoomCardEvent { }

abstract sig Enter extends RoomCardEvent { }
sig NormalEnter extends Enter { }
sig RecodeEnter extends Enter { }
```

events as objects

```
abstract sig Event {
   pre, post: Time }
   }

abstract sig HotelEvent extends Event {
   guest: Guest
   }

sig Checkout extends HotelEvent { }

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

sig Checkin extends RoomCardEvent { }

abstract sig Enter extends RoomCardEvent { }
sig NormalEnter extends Enter { }
sig RecodeEnter extends Enter { }
```



constraining events

```
sig Room {
  key: Key one -> Time
  ...
  }

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

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

event occurrences constrained by signature facts; as in Z, no explicit preconditions

key.post: relation from **Room** to **Key**

frame conditions

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

   prev.unchanged
   holds.unchanged
   occ.unchanged
   }

pred Event.unchanged (field: univ -> Time) {
   field.(this.pre) = field.(this.post)
   }
```

prev.unchanged: could also be written **unchanged[prev]** or **unchanged[this, prev]**, or **this.unchanged[prev]** …

reiter-style frame conditions

standard scheme

- for each operation, say which state components are unchanged

Ray Reiter's explanation closure axioms

- if x changed, e happened

Borgida et al's application to specs

- no frame conditions in operations
- instead, a global frame condition

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

```
sig Room {
   key: Key one -> Time,
   prev:  Key lone -> Time,
   occ: Guest -> Time
   } {
   Checkin.modifies [prev]
   (Checkin + Checkout).modifies [occ]
   RecodeEnter.modifies [key]
   }

pred modifies (es: set Event, field: univ -> Time) {
   all e: Event - es | field.(e.pre) = field.(e.post)
   }
```

note that an event sig name (eg, **Checkin**) just denotes a set of events

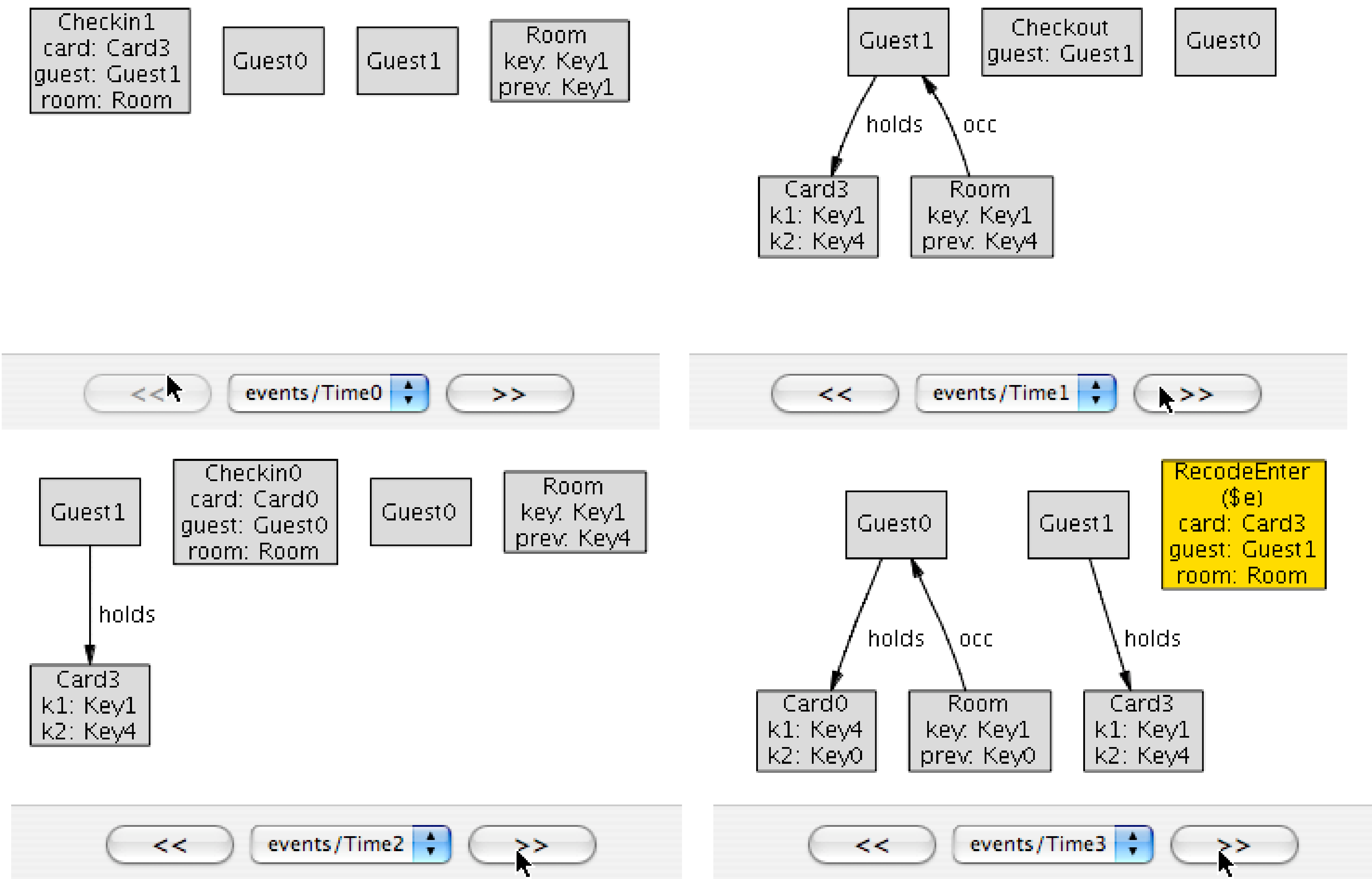
is it safe?

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 = occ.(e.pre) [e.room] |
        some occs => e.guest in occs
    }

check NoBadEntry for 5
```

intruder!


Checkin1
card: Card3
guest: Guest1
room: Room
Guest0
Guest1
Room
key: Key1
prev: Key1
<<
events/Time0
>>
Guest1
Checkout
guest: Guest1
Guest0
holds
occ
Card3
k1: Key1
k2: Key4
Room
key: Key1
prev: Key4
<<
events/Time1
>>
Guest1
Checkin0
card: Card0
guest: Guest0
room: Room
Guest0
Room
key: Key1
prev: Key4
holds
Card3
k1: Key1
k2: Key4
<<
events/Time2
>>
Guest0
Guest1
RecodeEnter
($e)
card: Card3
guest: Guest1
room: Room
holds
occ
holds
Card0
k1: Key4
k2: Key0
Room
key: Key1
prev: Key0
Card3
k1: Key1
k2: Key4
<<
events/Time3
>>

one solution

environmental assumption

- **no events intervene between checking in and entering room**

```
fact NoIntervening {
    all c: Checkin - pre.last |
        some e: Enter |
            e.pre = c.post and e.room = c.room and e.guest = c.guest
    }
```

```
Executing "Check NoBadEntry for 5"
  Solver=minisat(jni) Bitwidth=4 MaxSeq=5 Symmetry=20
  19163 vars. 720 primary vars. 50682 clauses. 280ms.
  No counterexample found. Assertion may be valid. 591ms.
```

coverage

when no counterexample found

- can ask analyzer to show 'coverage'
- highlights constraints used in proof

in this case

- only events shown: **init**, **Checkout**
- **Checkin** can't happen?

the culprit, also highlighted

- omitted **disj** keyword

```
fact FreshIssue {
    -- don't issue same key twice
    all  e1, e2: Checkin | e1.card.k2 != e2.card.k2
    -- don't issue key initially installed in lock
    all e: Checkin | e.card.k2 !in Room.key.first
    }
```

```
module hotel
open events as events

sig Key {}

sig Card {k1, k2: Key}
sig Room {
    key: Key one -> Time,
    prev:  Key lone -> Time,
    occ: Guest -> Time
    }

sig Guest {
    holds: Card -> Time
    }

pred init (t: Time) {
    prev.t = key.t
    key.t in Room lone -> Key
    no holds.t and no occ.t
    }
fact {first.init}

abstract sig HotelEvent extends Event {
    guest: Guest
    }

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

sig Checkin extends RoomCardEvent { }
    {
    no room.occ.pre
    card.k1 = room.prev.pre
    holds.post = holds.pre + guest -> card
    prev.post = prev.pre ++ room -> card.k2
    occ.post = occ.pre + room -> guest

    key.unchanged
    }

 bstract sig Enter extends RoomCardEvent { }
    {
    card in guest.holds.pre
    }

sig NormalEnter extends Enter { }
    {
    card.k2 = room.key.pre

    prev.unchanged
    holds.unchanged
    occ.unchanged
    key.unchanged
    }

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

    prev.unchanged
    holds.unchanged
    occ.unchanged
    }

sig Checkout extends HotelEvent { }
    {
    some occ.pre.guest
    occ.post = occ.pre - Room -> guest

    prev.unchanged
    holds.unchanged
    key.unchanged
    }

fact FreshIssue {
    -- don't issue same key twice
    all  e1, e2: Checkin | e1.card.k2 != e2.card.k2
    -- don't issue key initially installed in lock
    all e: Checkin | e.card.k2 !in Room.key.first
    }

assert MustHoldKey {
    all e: Enter | e.card in e.guest.holds.(e.pre)
    }
check MustHoldKey for 3

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

essence of Alloy

what makes Alloy tick?

all values are relations

- not just functions and sequences, but sets and scalars too

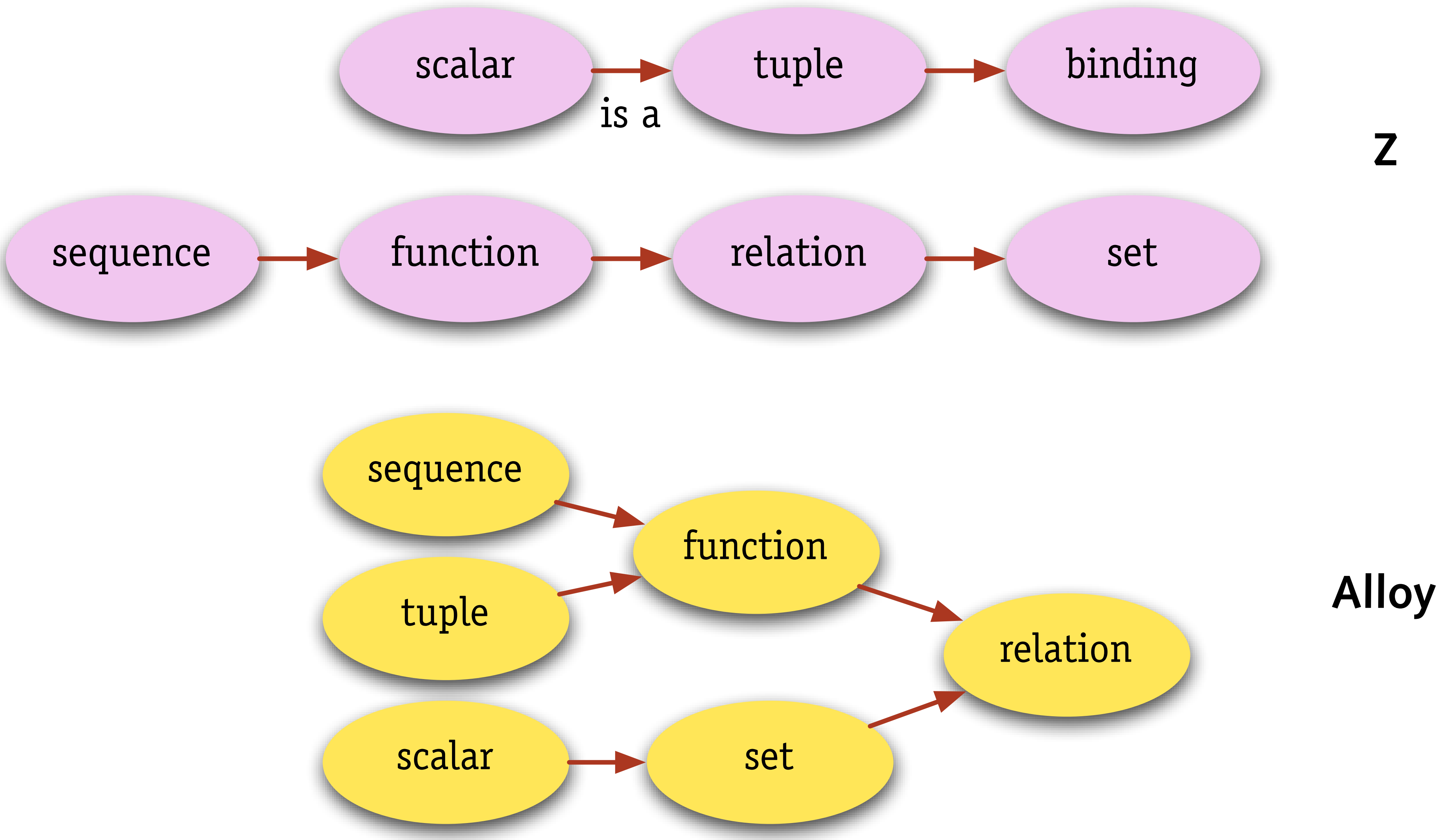
relations are flat

- dot is just generalized relational join

analysis is model finding

- result of analysis is always one instance
- visualizer's projection turns into cartoon

relations from Z to A



everything's a relation

a **relation** is a set of tuples

{(a0, d0), (a0, d1), (a1, d1)}

a **function** is a relation that's functional

{(a0, d0), (a1, d1)}

a **sequence** is a function over a prefix of the integers

{(0, d0), (1, d0), (2, d1)}

a **tuple** is a relation with one tuple

{(a0, d0)}

a **set** is a relation containing only one-tuples

{(a0), (a1)}

a **scalar** is a singleton set

{(a0)}

arrow product

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

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
  (in conventional notation, Alloy says $r \subseteq A \times B$ not $r \in 2^{(A \times B)}$)

when x and y are scalars

- x -> y is a tuple

dot join

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

$q.r\,[p] = p.(q.r)$

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 or f[x] is application of f to x

consequences

mostly preserve traditional syntax

```
r: s -> t
```

simple semantics: no undefined terms

```
all p: Person | p != p.wife        -- ill-typed in OCL
some i: I | a[i] = e               -- undefined in Z
```

'dereferencing' is just join

```
m.data [a]                         -- two applications of join
```

simple syntax: no set/scalar conversions

```
all p: Person |                    -- even though p is a scalar and p.parents is a set
   p.name.last in p.parents.name.last
```

small example

recall from our EXIF example

```
sig Photo { file: File, tags: set Tag }
no disj p, p′: Photo | p.file = p′.file or some p.tags & p′.tags
```

can rewrite more uniformly, despite set/scalar difference

```
no disj p, p′: Photo | some p.file & p′.file or some p.tags & p′.tags
```

(and can actually write more easily like this)

```
file in Photo lone -> one File
tags in Photo lone -> Tag
```

uniform counting symbols

formula quantifiers

all p: Photo | **one** f: File | p.file = f

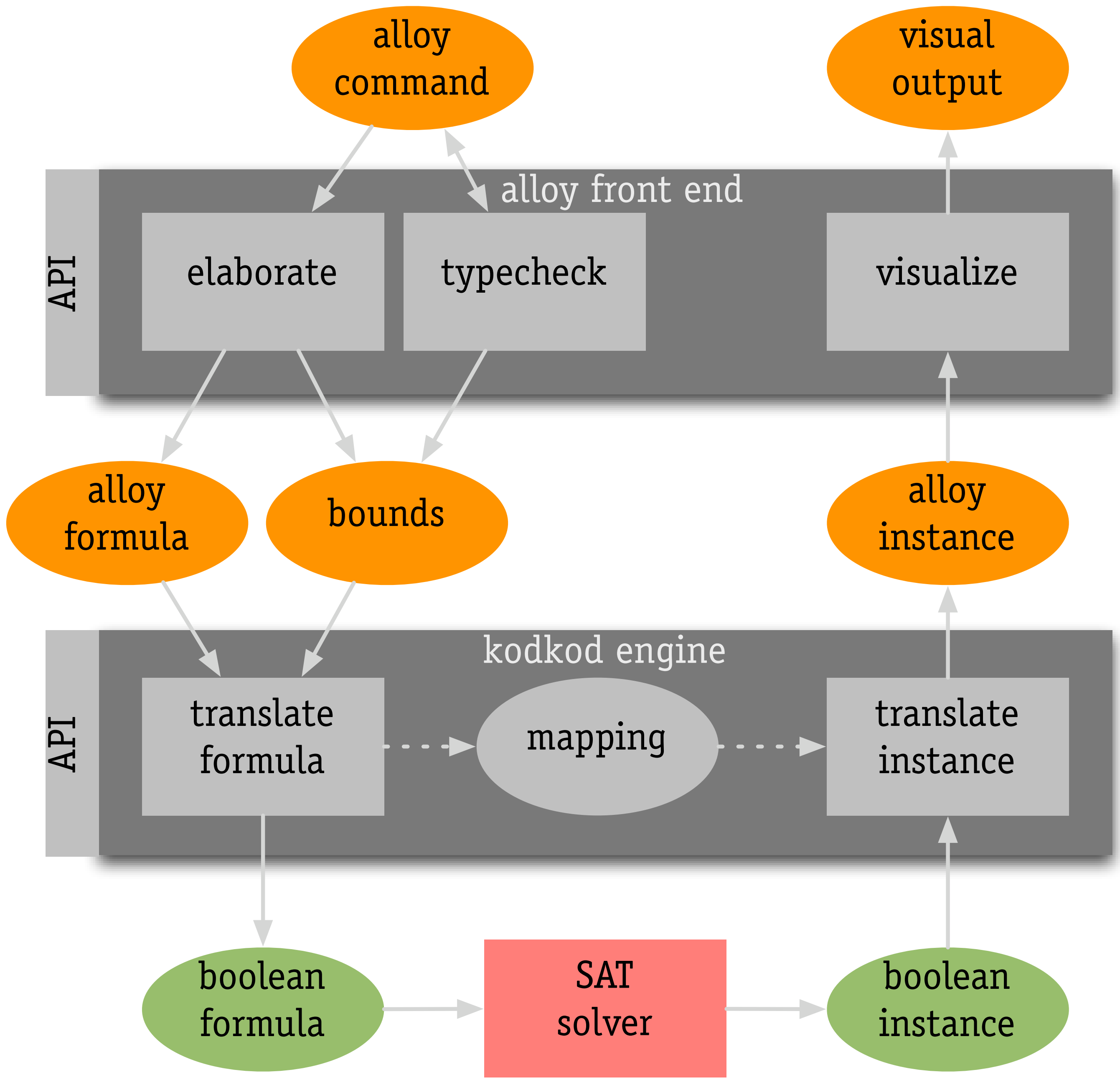
expression quantifiers

all p: Photo | **one** p.file

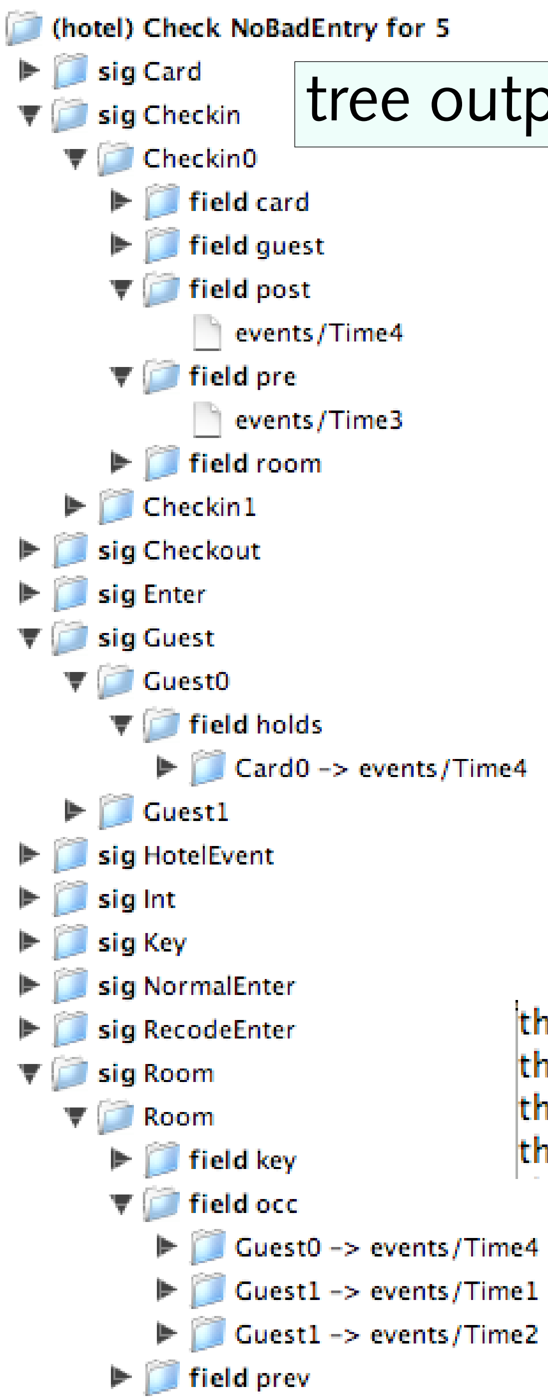
multiplicity markings

file **in** Photo -> **one** File

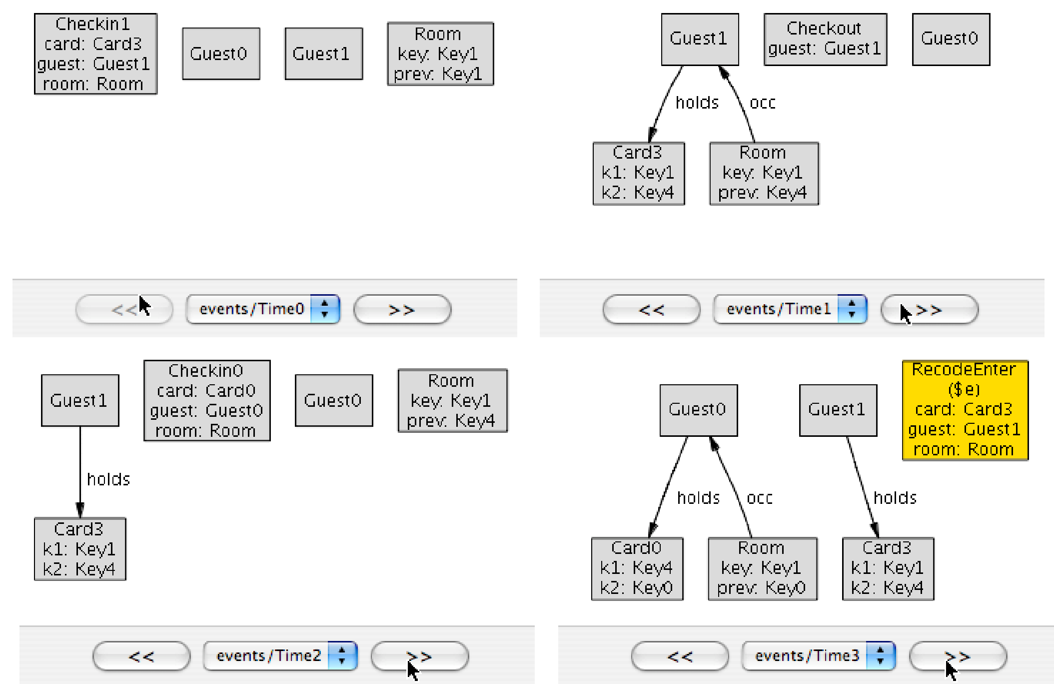
alloy architecture



cartoon is one instance



tree output



visualization, projected on Time

```
this/Room.key={Room$0->Key$0->events/Time$0, Room$0->Key$0->events/Time$1, Roo
this/Room.prev={Room$0->Key$0->events/Time$0, Room$0->Key$1->events/Time$4, Ro
this/Room.occ={Room$0->Guest$0->events/Time$4, Room$0->Guest$1->events/Time$1
this/Guest.holds={Guest$0->Card$0->events/Time$4, Guest$1->Card$3->events/Time$1
```

textual output

acknowledgments

current students
& collaborators
who've contributed to Alloy

Felix Chang
lead developer, A4

Emina Torlak
developer, Kodkod

Greg Dennis
Derek Rayside
Robert Seater
Mana Taghdiri
Jonathan Edwards
Vincent Yeung

former students
who've contributed to Alloy

Ilya Shlyakhter
Manu Sridharan
Sarfraz Khurshid
Mandana Vaziri
Andrew Yip
Sam Daitch
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, tutorial

http://softwareabstractions.org

- sample chapters, model repository

http://tech.groups.yahoo.com/group/alloy-discuss/

- discussion group, 560 members

alloy@mit.edu

- tool support and comments

dnj@mit.edu

- happy to hear from you!