EVENTS & ENVIRONMENT

Daniel Jackson · Lipari Summer School · July 18-22, 2005
leader election: review
election progress: first attempt

from this:

```plaintext
assert AtMostOneElected { 
  lone elected.Time
}
```

just try this?

```plaintext
assert AtLeastOneElected { 
  some elected.Time
}
```

counterexample:
> just skips in every step!
election progress: again

add progress filtering constraint
> if some process has an ID to send, some process doesn’t skip

```plaintext
defined progress () {  
    all t: Time - to/last() |  
    let t' = to/next (t) |  
        some Process.toSend.t => some p: Process | not skip (t, t', p)  
}

assert AtLeastOneElected {  
    progress () => some elected.Time  
}

check AtLeastOneElected for 5 Process, 10 Time
```
topics for today

some new idioms
› events as explicit objects
› Reiter-style frame conditions

environment
› assumptions about environment
   at heart of many requirements failures
frame conditions
frame conditions

in declarative models

› unmentioned ≠ unchanged

so need frame conditions to say that

› relation doesn’t change
  \( \text{xs.buffer} = \text{xs'.buffer} \)

› relation changes only at some object
  \((b.\text{addr}[n] = a) \text{ and } (\text{all } m: \text{Name} - n | b'.\text{addr}[m] = b.\text{addr}[m])\)
mitigating frame conditions

generate automatically
▷ from ‘modifies at most’ clause
▷ from non-mention of relations
▷ loss of flexibility

structure constraints to minimize
▷ specify value of whole relation
   
   \[(b.\text{addr} [n] = a) \text{ and } (\text{all } m: \text{Name} - n | b'.\text{addr} [m] = b.\text{addr} [m])\]
   
   \[b'.\text{addr} = b.\text{addr} ++ n->a\]

▷ factor out

\[\text{pred noChangeExceptAt } (b, b': \text{Book}, n: \text{Name}) \{\]
   
   \[\text{all } m: \text{Name} - n |\]
   
   \[b'.\text{addr} [m] = b.\text{addr} [m] \text{ and } m <: b'.\text{names} = m <: b.\text{names}\]
   
\}
more radical mitigations

define components
  › eg, elected in leader election model

Ray Reiter’s scheme
  › add ‘explanation closure axioms’
    if field f changed, then event e happened
forms
form: explicit events

sig Time {}

sig O {f: X -> Time}

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

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

form: 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}
form: explanation closure

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

recoadable 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  
}
suppose you write

```
sig S1 {f: A}
sig S2 extends S1 {g: B}
```

then this introduces

- sets
  - \( S1 \)
  - \( S2 \) in \( S1 \)

- relations
  - \( f: S1 \rightarrow A \)
  - \( g: S2 \rightarrow B \)

aside: \( s1.g \) is not necessarily bad
event classification

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

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

**sig** Checkin **extends** RoomKeyEvent { }
{
  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 Enter extends RoomKeyEvent { }
  {card in guest.cards.pre}

sig NormalEnter extends Enter { }
  {card.fst = room.fst.pre and card.snd = room.snd.pre}

sig RecodeEnter extends Enter { }
  {
    card.fst = room.snd.pre
    fst.post = fst.pre ++ room -> card.fst
    snd.post = snd.pre ++ room -> card.snd
  }
free variables

what’s going on here?

why are explicit events good?
› appear as atoms in visualization
› can classify events

why can’t you classify with predicates?
› you can, but it’s uglier
› free vs. bound variables

```
pred enter (t, t': Time, r: Room, g: Guest) {...}
pred normalEnter (t, t': Time, r: Room, g: Guest) {
    enter (t, t', r, g) and ...
}"
```
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 RecodeEnter
    prev.t = prev.t' and issued.t = issued.t' and cards.t = cards.t'
    or e in Checkin
    occ.t = occ.t' or e in Checkin + Checkout
  }
}
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
after checking in, guest immediately enters room:

```
fact NoIntervening {
    all c: Checkin |
    some e: Enter {
        e.pre = c.post
        e.room = c.room
        e.guest = c.guest
    }
}
```
specification is at machine interface, but requirement might not be
more generally: domains

see: *Problem Frames*, Michael Jackson, Addison Wesley, 2001
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
checking code against relational logic specifications

- basic idea and optimizations [Vaziri]
- iterative refinement of procedure summaries [Taghdiri]
test case generation

generating test cases from invariants [Khurshid]
› easier to write invariant than test cases
› random generation fails when precondition is strong
› Alloy’s symmetry breaking eliminates redundant tests
reminder

return memory sticks to alfredo in next break!
acknowledgments

current students & collaborators
Greg Dennis
Derek Rayside
Robert Seater
Mana Taghdiri
Emina Torlak
Jonathan Edwards
Vincent Yeung

former students
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

dnj@mit.edu
  › happy to hear from you!

Software Abstractions
  › MIT Press, 2006
that’s all folks!