STATES, OPERATIONS & TRACES

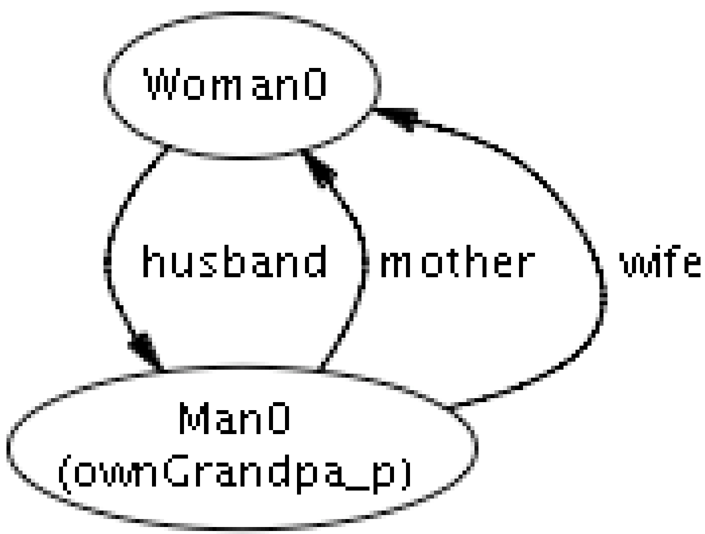
Daniel Jackson · Lipari Summer School · July 18-22, 2005

self-grandpa, version 2

```
module examples/grandpa/grandpa2
abstract sig Person {father: lone Man, mother: lone Woman}
sig Man extends Person {wife: lone Woman}
sig Woman extends Person {husband: lone Man}
fact {
   no p: Person | p in p.^(mother+father)
   wife = ~husband
   }
fun grandpas (p: Person): set Person {
  let parent = mother + father + father.wife +mother.husband |
      p.parent.parent & Man }
pred ownGrandpa (p: Person) {p in grandpas (p)}
run ownGrandpa for 4 Person
```

self-grandpa, solution 1

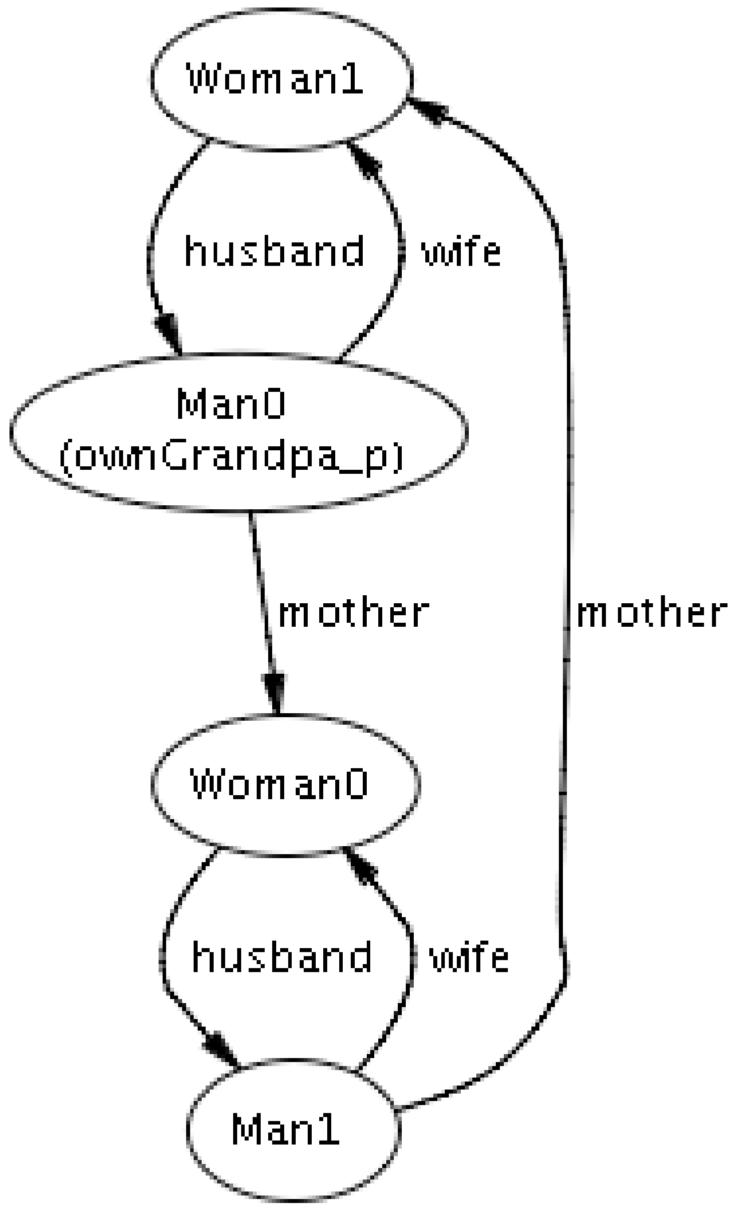
not suitable for a popular song



self-grandpa, version 3

```
module examples/grandpa/grandpa2
abstract sig Person {father: lone Man, mother: lone Woman}
sig Man extends Person {wife: lone Woman}
sig Woman extends Person {husband: lone Man}
fact {
   no p: Person | p in p.^(mother+father)
   wife = ~husband
   no wife & *(mother+father).mother
   no husband & *(mother+father).father
   }
fun grandpas (p: Person): set Person {
  let parent = mother + father + father.wife +mother.husband |
      p.parent.parent & Man }
pred ownGrandpa (p: Person) {p in grandpas (p)}
run ownGrandpa for 4 Person
```

self-grandpa, solution 2



topics for today

idioms for dynamic behaviour

idioms for modelling
- states, operations & invariants
- composite state
- local state
- execution traces

idioms for analysis
- inductive invariants
- algebraic properties
- temporal properties

going slower …

less material for today
opportunity to ask questions about logic/language in context
stupid questions welcome!

example: media management

just look at a few tiny features
› show/hide
› select
› cut/paste

premise
› simple, powerful abstractions make good user interfaces
› no point doing a usability study on an incoherent design

on the benefits of software

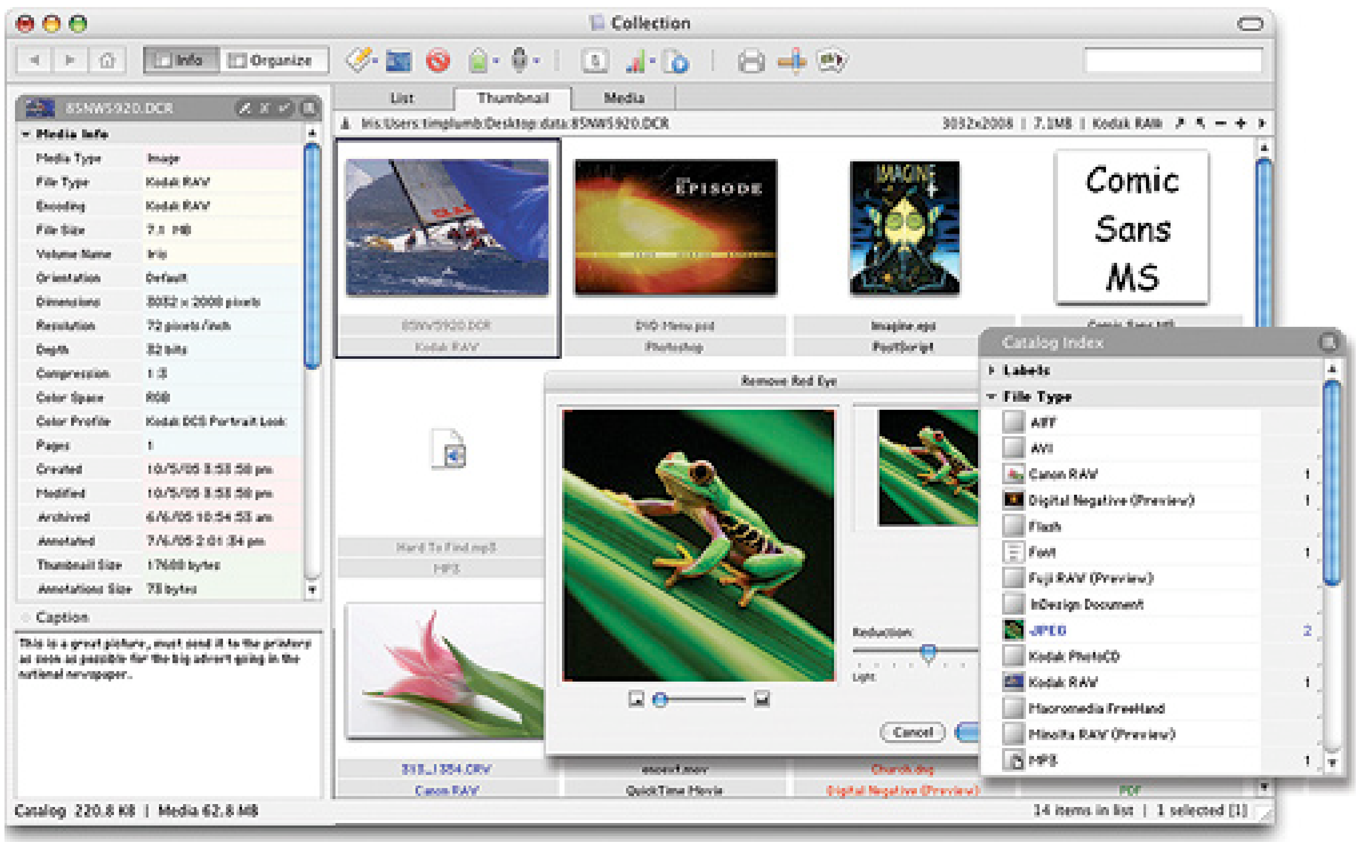
I have always wished that my computer would be as easy to use as my telephone. My wish has come true. I no longer know how to use my telephone.

--Bjarne Stroustrup

intro to media management

media asset management

applications for organizing photos, fonts, videos, sound tracks, etc
eg, iView Media Pro



demo of IVMP

IVMP model

form: state, op & invariant

```
sig State {…}

pred op (s, s′: State) {…}

pred inv (s: State) {…}

assert opPreservesInv {
   all s, s′: State | inv (s) and op (s, s′) implies inv (s′)
   }
check opPreservesInv
```

IVMP state

```
module examples/assets/assets

sig Catalog {}
sig Asset {}
one sig Undefined {}

sig ApplicationState {
  catalogs: set Catalog,
  catalogState: catalogs -> one CatalogState,
  currentCatalog: catalogs,
  buffer: set Asset }

sig CatalogState {
  assets: set Asset,
  part hidden, showing: set assets,
  selection: set assets + Undefined }
```

an IVMP invariant

```
pred appInv (xs: ApplicationState) {
   all cs: xs.catalogs | catalogInv (xs.catalogState[cs])
   }

pred catalogInv (cs: CatalogState) {
   cs.selection = Undefined
        or (some cs.selection and cs.selection in cs.showing)
   }
```

show/hide ops

```
pred showSelected (cs, cs': CatalogState) {
   cs.selection != Undefined
   cs'.showing = cs.selection
   cs'.selection = cs.selection
   cs'.assets = cs.assets
   }

pred hideSelected (cs, cs': CatalogState) {
   cs.selection != Undefined
   cs'.hidden = cs.hidden + cs.selection
   cs'.selection = Undefined
   cs'.assets = cs.assets
   }
```

note: asymmetry, frame conditions

paste op

```
pred paste (xs, xs': ApplicationState) {
   xs'.catalogs = xs.catalogs
   xs'.currentCatalog = xs.currentCatalog
   let cs = xs.catalogState[xs.currentCatalog], buf = xs.buffer {
      xs'.buffer = buf
      some cs': CatalogState {
         cs'.assets = cs.assets + buf
         cs'.showing = cs.showing + buf
         cs'.selection = buf
         xs'.catalogState = xs.catalogState ++ xs.currentCatalog -> cs'
         }
   }
}
```

checking invariant

```
assert PastePreservesInv {
  all xs, xs': ApplicationState |
     appInv (xs) and paste (xs, xs') => appInv (xs')
     }

check PastePreservesInv
```

counterexample!

```
sig ApplicationState
   catalogState =
       {ApplicationState_0 -> Catalog_0 -> CatalogState_1,
       ApplicationState_1 -> Catalog_0 -> CatalogState_0}
   buffer = {}

sig CatalogState
   showing =
       {CatalogState_0 -> Asset_0, CatalogState_1 -> Asset_0}
   selection = {CatalogState_1 -> Asset_0}

PastePreservesInv_xs = {ApplicationState_0}
PastePreservesInv_xs' = {ApplicationState_1}
paste0_cs' = {CatalogState_0}
appInv_cs = {Catalog_0}
```

paste revisited

```
pred paste (xs, xs': ApplicationState) {
   xs'.catalogs = xs.catalogs
   xs'.currentCatalog = xs.currentCatalog
   let cs = xs.catalogState[xs.currentCatalog], buf = xs.buffer {
      some cs': CatalogState {
         xs'.buffer = buf
         cs'.assets = cs.assets + buf
         cs'.showing = cs.showing + buf
         cs'.selection = if some buf then buf else Undefined
         xs'.catalogState = xs.catalogState ++ xs.currentCatalog -> cs'
         }
   }
}
```

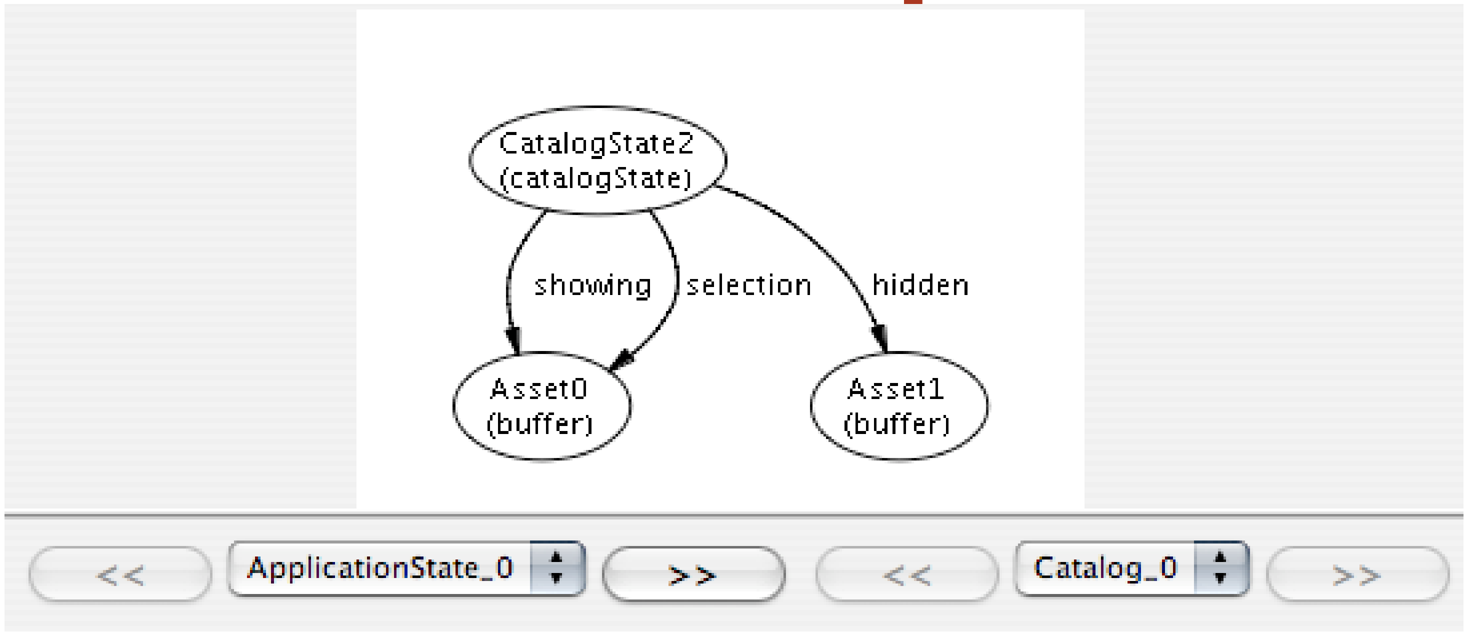
form: checking inverses

```
sig State {…}

pred op1 (s, s’: State) {…}

pred op2 (s, s’: State) {…}

assert Inverses {
  all s, s’, s”: State | op1 (s, s’) and op2 (s’, s”) => s = s”
  }
check Inverses
```

cut/paste

```
assert CutPaste {
   all xs, xs', xs": ApplicationState |
      (appInv (xs) and cut (xs, xs') and paste (xs', xs")) =>
         sameApplicationState (xs, xs")
   }

check CutPaste for 3 but 2 Asset
```

counterexample!



problem: old buffer is lost

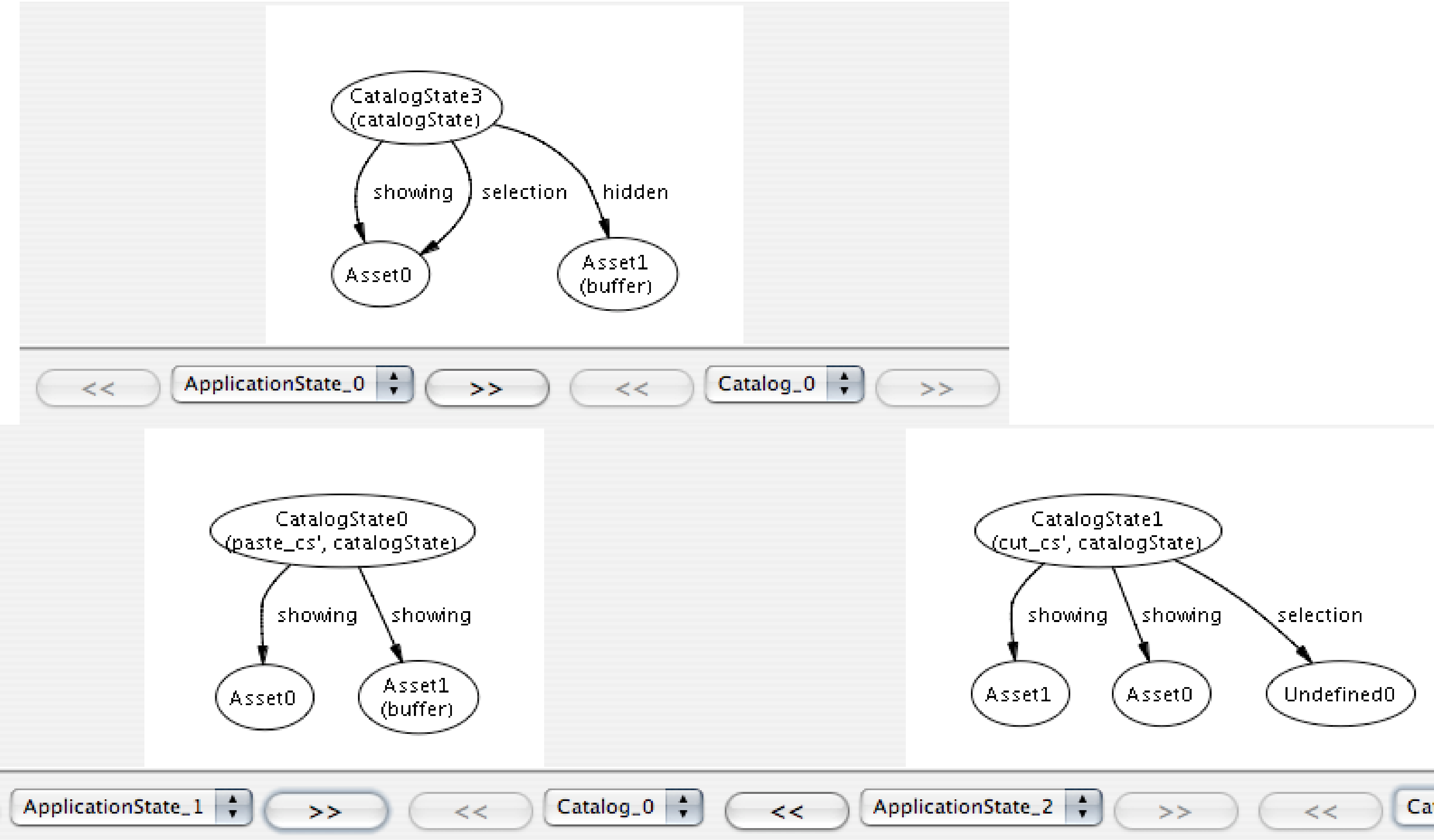
state equivalence, revisited

```
pred sameApplicationState (xs, xs': ApplicationState) {
  xs'.catalogs = xs.catalogs
  all c: xs.catalogs |
     sameCatalogState (c.(xs.catalogState), c.(xs'.catalogState))
  xs'.currentCatalog = xs.currentCatalog
  /* xs'.buffer = xs.buffer */
  }
```

paste/cut

```
assert PasteCut {
  all xs, xs', xs": ApplicationState |
      (appInv (xs) and paste (xs, xs') and cut (xs', xs")) =>
          sameApplicationState (xs, xs")
  }

check PasteCut for 3 but 2 Asset
```

counterexample!



two problems: selection lost & pasting of hidden asset

paste revisited, again

```
pred paste (xs, xs': ApplicationState) {
  xs'.catalogs = xs.catalogs
  xs'.currentCatalog = xs.currentCatalog
  let cs = xs.catalogState[xs.currentCatalog], buf = xs.buffer {
      some cs': CatalogState {
          xs'.buffer = buf
          cs'.assets = cs.assets + buf
          cs'.showing = cs.showing + (buf - cs.assets)
          cs'.selection = if some buf then buf - cs.assets else Undefined
          xs'.catalogState = xs.catalogState ++ xs.currentCatalog -> cs'
          }
  }
}
```

lessons

like many design problems

- seems trivial at first
- but getting it right is hard

local state & traces: leader election model

form: local state

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

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

leader election in a ring

problem

- elect a leader
- processes in a ring
- distinguished only by ID

Chang & Roberts

- each process passes its ID to the right (say)
- on receipt of an ID `i`

    `i > my ID`: pass it on

    `i < my ID`: drop it

    `i = my ID`: elect myself leader

state: topology & process state

```
module examples/election/election
open util/ordering[Time] as to -- import library module for time order
open util/ordering[Process] as po  -- ordering on process ids

sig Time {}
sig Process {
   succ: Process,                      -- successor in ring
   toSend: Process -> Time,            -- pool of ids to send at time t
   elected: set Time                   -- times at which elected leader
   }

fact ring {
   all p: Process | Process in p.^succ     -- constrain succ so it's a ring
   }
```

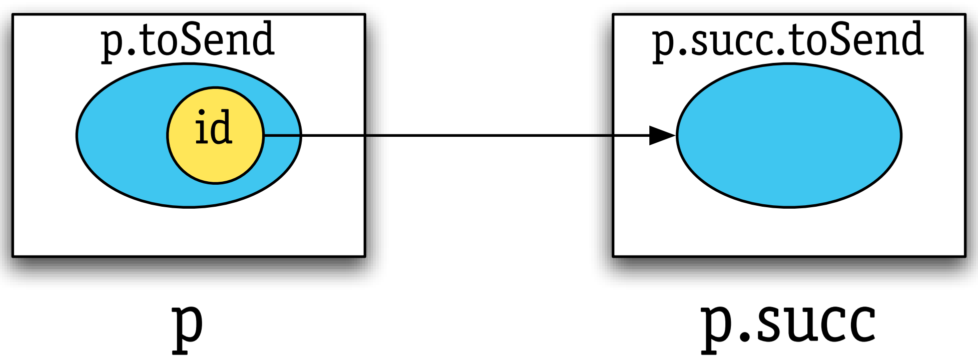
initialization

initially, each process is ready to send its own ID

```
pred init (t: Time) {
  all p: Process | p.toSend.t = p
  }
```

transition step

```
pred step (t, t': Time, p: Process) {
   let from = p.toSend, to = p.succ.toSend |
      some id: from.t {
         from.t' = from.t - id
         to.t' = to.t + (id - po/prevs(p.succ))
         }
      }
```



turning transitions into traces

```
fact traces {
   init (to/first ())
   all t: Time - to/last() | let t' = to/next (t) |
       all p: Process |
           step (t, t', p) or step (t, t', succ.p) or skip (t, t', p)
   }

pred skip (t, t': Time, p: Process) {
   p.toSend.t = p.toSend.t'
   }
```

defining election

define elected with a fact

- no process elected in first time instant
- processes elected at t are those that got their own ID at t

```
fact defineElected {
  no elected.to/first()
  all t: Time - to/first()|
     elected.t = {p: Process | p in p.toSend.t - p.toSend.(to/prev(t))}
  }
```

alternatively, update elected in step

- but this is better separation of concerns

simulation

```
pred show () {
  some elected
  }
run show for 3 but 4 Time
```

checking

```
assert AtMostOneElected {
  lone elected.Time
  }
check AtMostOneElected for 5 Process, 10 Time
```

demo

machine diameter

scoping the trace length

'small scope hypothesis'

› most bugs have counterexamples in small scopes

but is this really plausible for trace length?

› scope (Time) bounds number of steps in trace
› maybe trace is too short to reach interesting states

can mitigate this problem

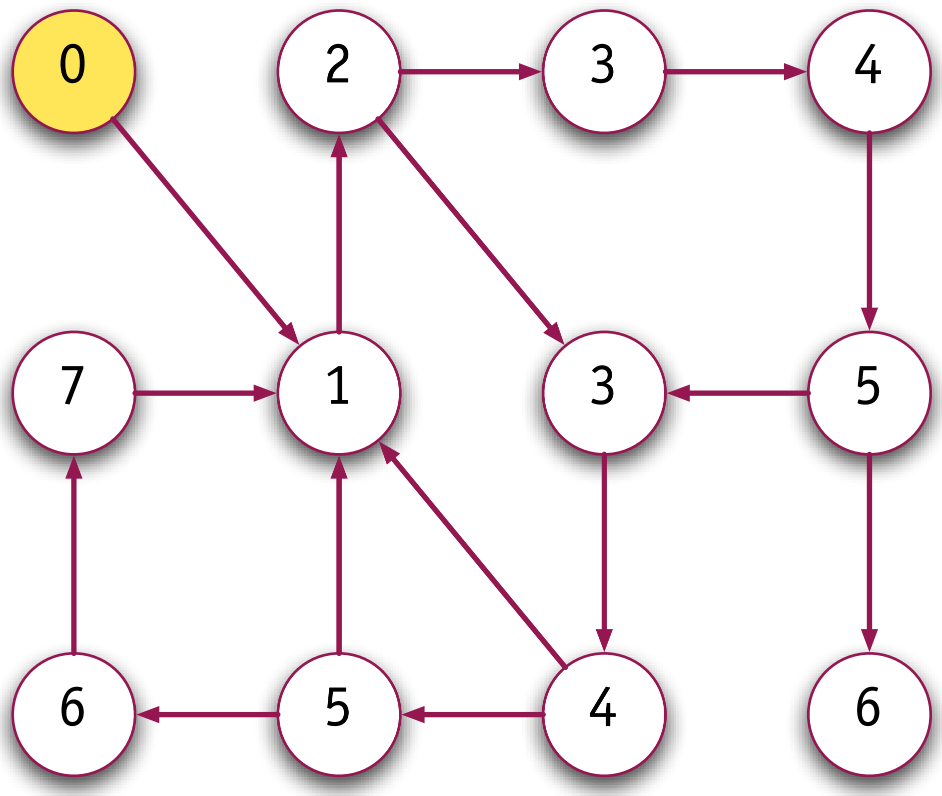
› for small models
› using ideas from Biere et al
› hardwired in BMC, but directly expressible in Alloy logic

defining diameter

idea: set trace length to **diameter**

definition

- diameter (M) is smallest k such that
  every state can be reached in k steps from initial state



approximating diameter

suppose ∄ loopless path of length k

› then diameter < k

can use analyzer to find k :

```
pred loopless () {
 no disj t, t': Time | toSend.t = toSend.t'
 }

run loopless for 12 Time, 3 Process -- instance found
run loopless for 13 Time, 3 Process -- no instance found
```

approximated diameter grows fast

› for 5 Process, computed diameter is 33

homework

homework: election progress

check that at least one process is elected

- formulate an assertion & check it
- change model if necessary

```
assert AtLeastOneElected {
  ...
  }
```