

# Alloy Dynamic Modeling

base on slides from

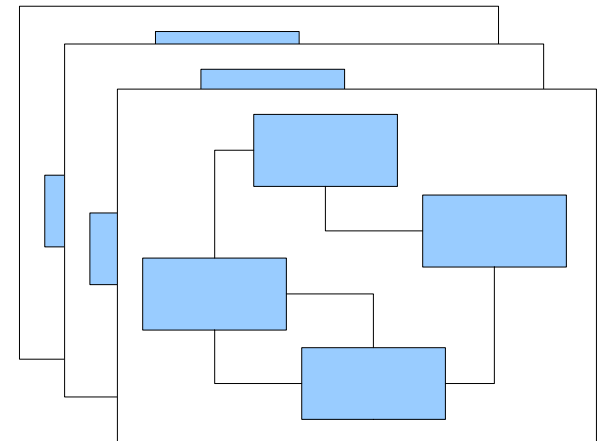
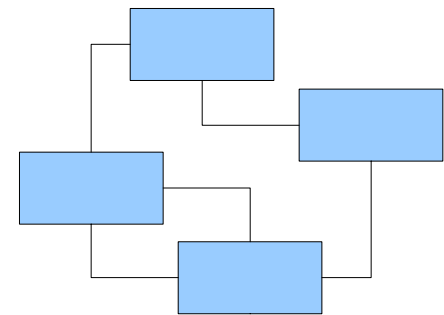
Greg Dennis and Rob Seater  
Software Design Group, MIT



# static vs. dynamic models

---

- static models
  - describes states, not behaviors
  - properties are *invariants*
  - e.g. that a list is sorted
- dynamic models
  - describe transitions between states
  - properties are *operations*
  - e.g. how a sorting algorithm works



# model of an address book

```
module addressBook

  abstract sig Target {}

  sig Addr extends Target {}

  abstract sig Name extends Target {}

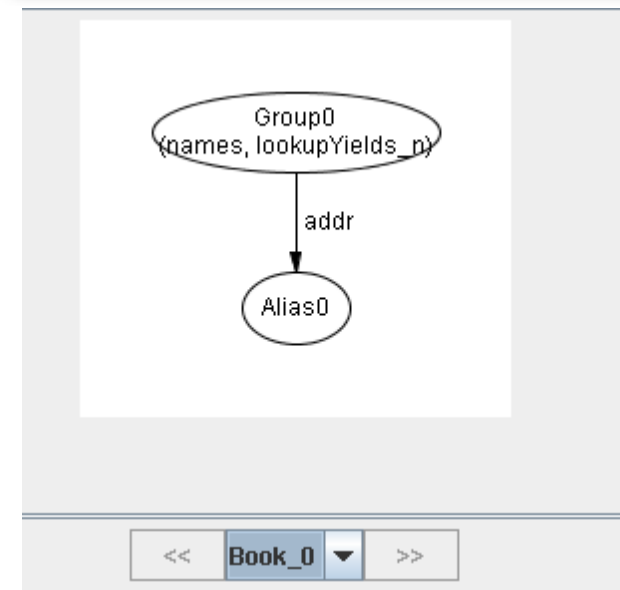
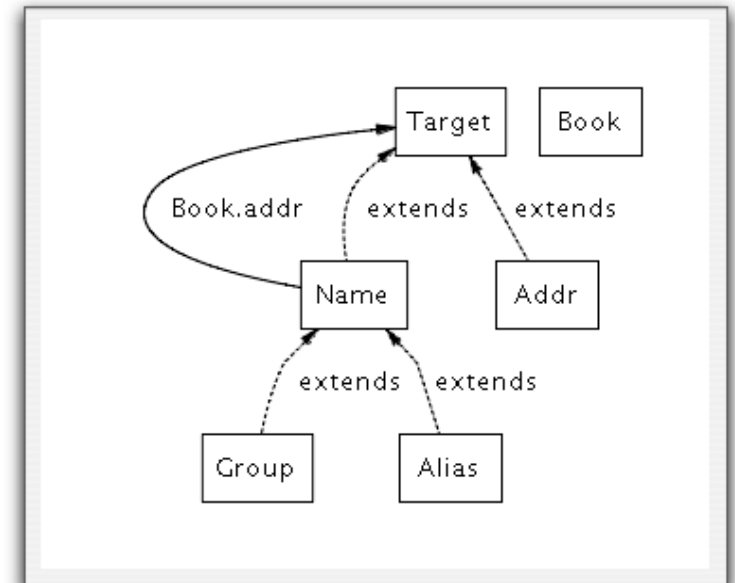
  sig Alias, Group extends Name {}

  sig Book {
    names: set Name,
    addr: names -> some Target }
  {
    no n: Name | n in n.^(addr)
    all a: Alias | lone a.addr
  }

  fun lookup [b: Book, n: Name]: set Addr {
    n.^(b.addr) & Addr
  }

  assert lookupYields {
    all b: Book, n: b.names | some lookup[b,n]
  }

  check lookupYields for 4 but 1 Book
```



# what about operations?

---

- how is a name & address added to a book?
- no built-in model of execution
  - no notion of time or mutable state
- need to model time/state explicitly
- Solution: can use a new “book” after each mutation:



```
pred add (b, b': Book, n: Name, t: Target) {  
    b'.addr = b.addr + n->t  
}
```

# address book: delete operation

---

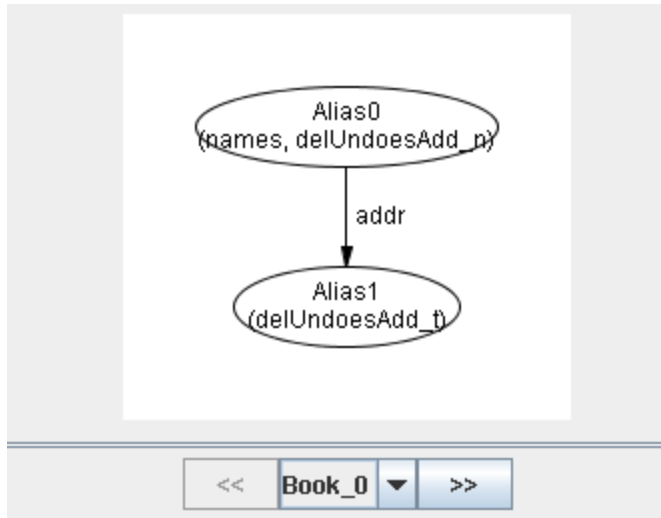
- The delete should remove a name mapping

```
pred del (b, b': Book, n: Name, t: Target) {  
    b'.addr = b.addr - n -> t  
}
```

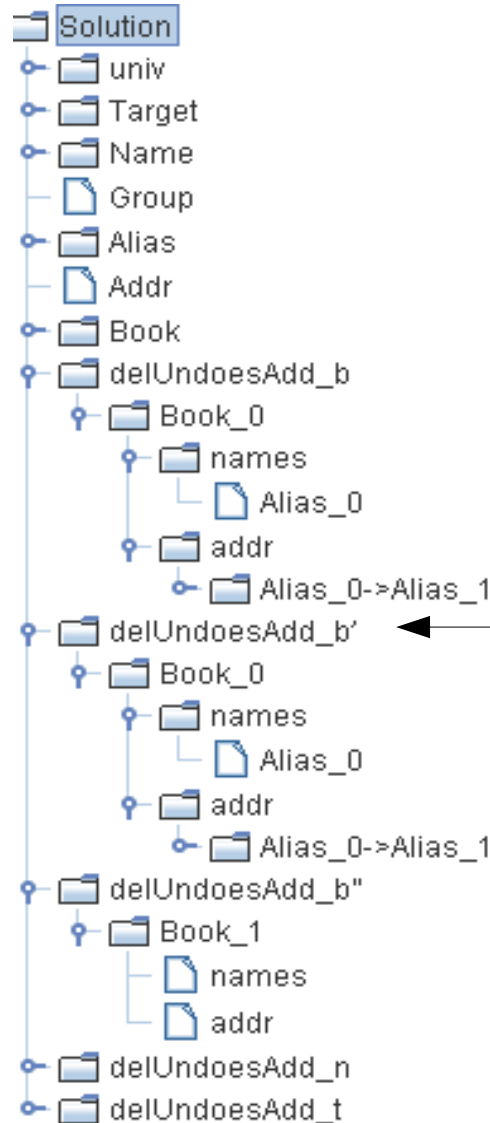
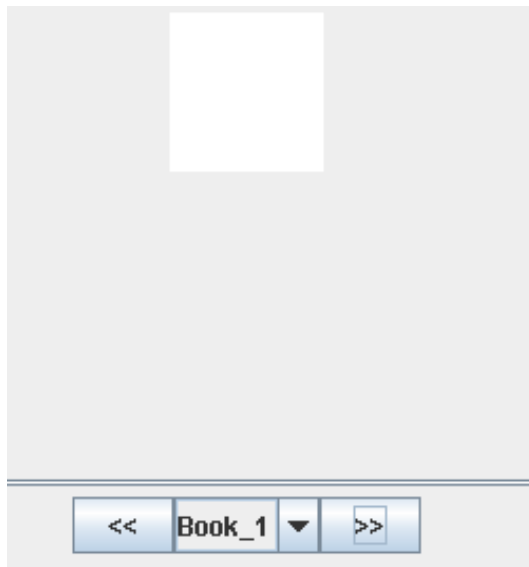
- Check whether delete undoes add

```
assert delUndoesAdd {  
    all b,b',b'': Book, n: Name, t: Target |  
    add (b,b',n,t) and del (b',b'',n,t)  
    implies b.addr = b''.addr  
}  
check delUndoesAdd for 3      ????????
```

# Counter Example



<< Book\_0 >>



The book after the add is again Book\_0

# address book: delete operation - 2

---

- Check whether delete undoes add

```
assert delUndoesAdd {  
  all b,b',b'': Book, n: Name, t: Target |  
  no n.(b.addr) and  
    add (b,b',n,t) and del (b',b'',n,t)  
    implies b.addr = b''.addr  
}  
check delUndoesAdd for 3      ?????
```

## Executing "Check delUndoesAdd for 3"

Solver=zchaff(jni) Bitwidth=4 MaxSeq=3 SkolemDepth=1 Symmetry=20  
1135 vars. 63 primary vars. 1897 clauses. 63ms.  
No counterexample found. Assertion may be valid. 0ms.

# pattern: abstract machine

---

- treat actions as operations on global state

```
sig State {...}

pred init [s: State] {...}

pred inv [s: State] {...}

pred op1 [s, s': State] {...}
...
pred opN [s, s': State] {...}
```

Possible invariant for Book

```
pred inv [b: Book] {
  let addr = b.addr | all n: Name {
    n not in n.^addr
    some addr.n => some n.addr
  }
}
```

- in addressBook, *State* is *Book*
  - each *Book* represents a new system state

# pattern: invariant preservation

---

- check that an operation preserves an invariant

```
assert initEstablishes {  
  all s: State | init[s] => inv[s]  
}  
check initEstablishes  
  
// for each operation  
assert opPreserves {  
  all s, s': State |  
    inv[s] && op[s, s'] => inv[s']  
}  
check opPreserves
```

- apply this pattern to the addressBook model
- do the *add* and *delete* ops preserve the invariant?

# pattern: operation preconditions

---

- include precondition constraints in an operation
  - operations no longer total
- the *add* operation with a precondition:

```
pred add[b, b': Book, n: Name, t: Target] {  
  // precondition  
  t in Name => (n !in t.*(b.addr) && some b.addr[t])  
  // postcondition  
  b'.addr = b.addr + n->t  
}
```

- check that *add* now preserves the invariant
- add a sensible precondition to the delete operation
  - check that it now preserves the invariant

# what about traces?

---

- we can check properties of individual transitions
- what about properties of sequences of transitions?
- entire system simulation
  - simulate the execution of a sequence of operations
- algorithm correctness
  - check that all traces end in a desired final state
- planning problems
  - find a trace that ends in a desired final state



# pattern: traces

---

- model sequences of executions of abstract machine
- create linear (total) ordering over states
- connect successive states by operations
  - constrains all states to be reachable

```
open util/ordering[State] as ord
...
fact traces {
  init (ord/first())
  all s: State - ord/last() |
    let s' = ord/next(s) |
      op1(s, s') or ... or opN(s, s')
}
```

- apply traces pattern to the address book model

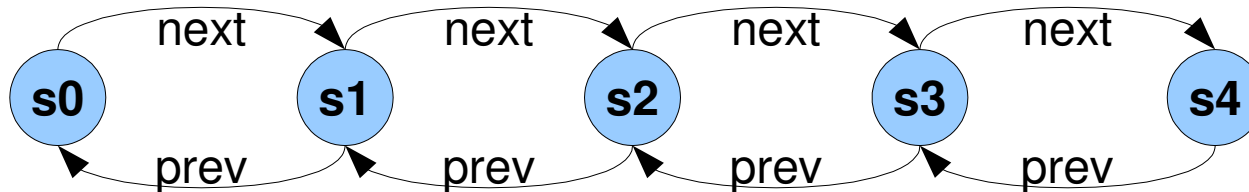
# ordering module

---

- establishes linear ordering over atoms of signature  $S$

```
open util/ordering[S]
```

$S = s0 + s1 + s2 + s3 + s4$



```
first() = s0  
last() = s4  
next(s2) = s3  
prev(s2) = s1  
nexts(s2) = s3 + s4  
prevs(s2) = s0 + s1
```

```
lt(s1, s2) = true  
lt(s1, s1) = false  
gt(s1, s2) = false  
lte(s0, s3) = true  
lte(s0, s0) = true  
gte(s2, s4) = false
```

# Initialize

---

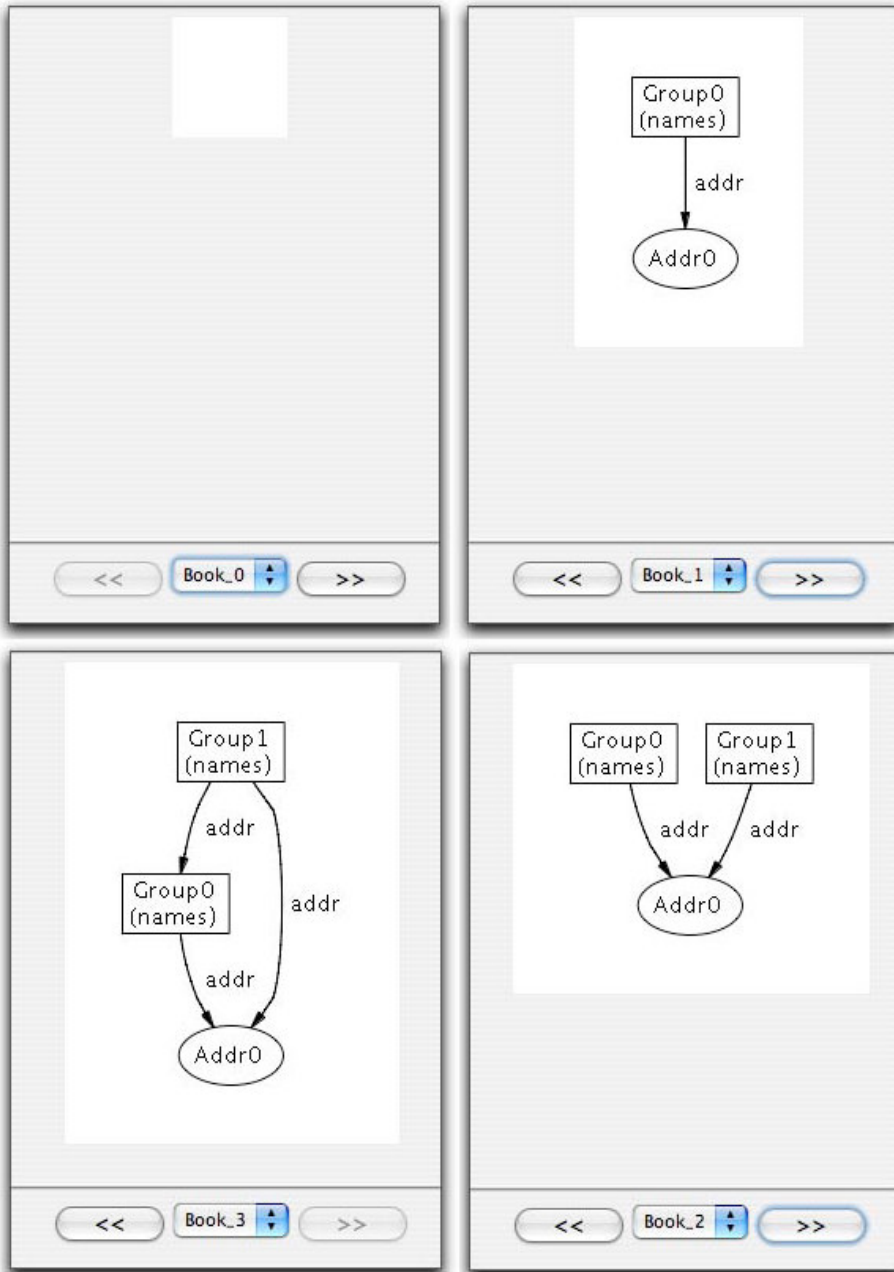
```
module tour/addressBook3
open util/ordering [Book]
...
pred init (b: Book) {no b.addr}

fact traces {
  init (first ())
  all b: Book - last () | let b' = next (b) |
    some n: Name, t: Target | add (b, b', n, t) or del (b, b', n, t)
}
```

The ordering on books is provided by the library module *util/ordering*. This module is generic—that is, it can order a set of any type—so when opened it must be instantiated with a type (in this case, *Book*). The module has its own signatures and fields, but is accessed through the functions *first*, *next* and *last*, giving the first element in the order, the element following a given element, and the last.

The predicate *init* gives the initial condition—that the address book is empty. The fact *traces* specifies the constraints that make the ordering a trace: that the initial condition holds for the first book in the trace, and that any book *b* (except the last) and its successor *b'* are related by the constraints of the *add* or *del* operation.

# run show for 4



The last book is interesting:  
it creates two routes to the same address.  
Should that be possible???

**NOW**  
check the empty lookup by using traces.

# Empty lookups

To investigate the empty lookup problem, we can check the same assertion as before:

```
assert lookupYields {all b: Book, n: b.names | some lookup(b,n)}  
check lookupYields for 3 but 4 Book
```

Counter example

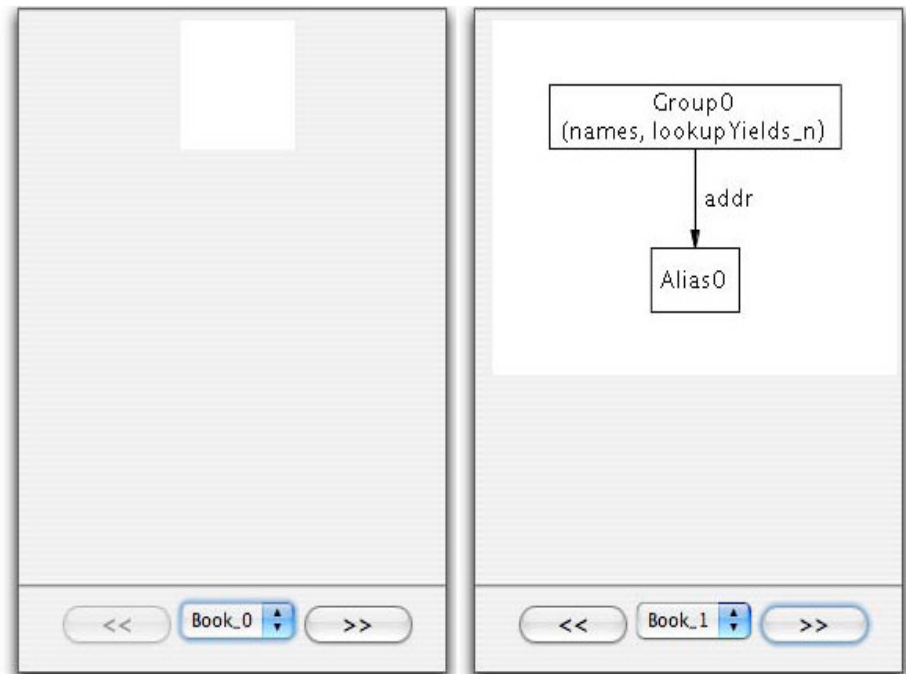


FIG. 2.16 Counterexample trace violating *lookupYields* with one step of *add*.

```
pred add (b, b': Book, n: Name, t: Target) {  
  b'.addr = b.addr + n->t  
}
```

# Solving meaningless adds

---

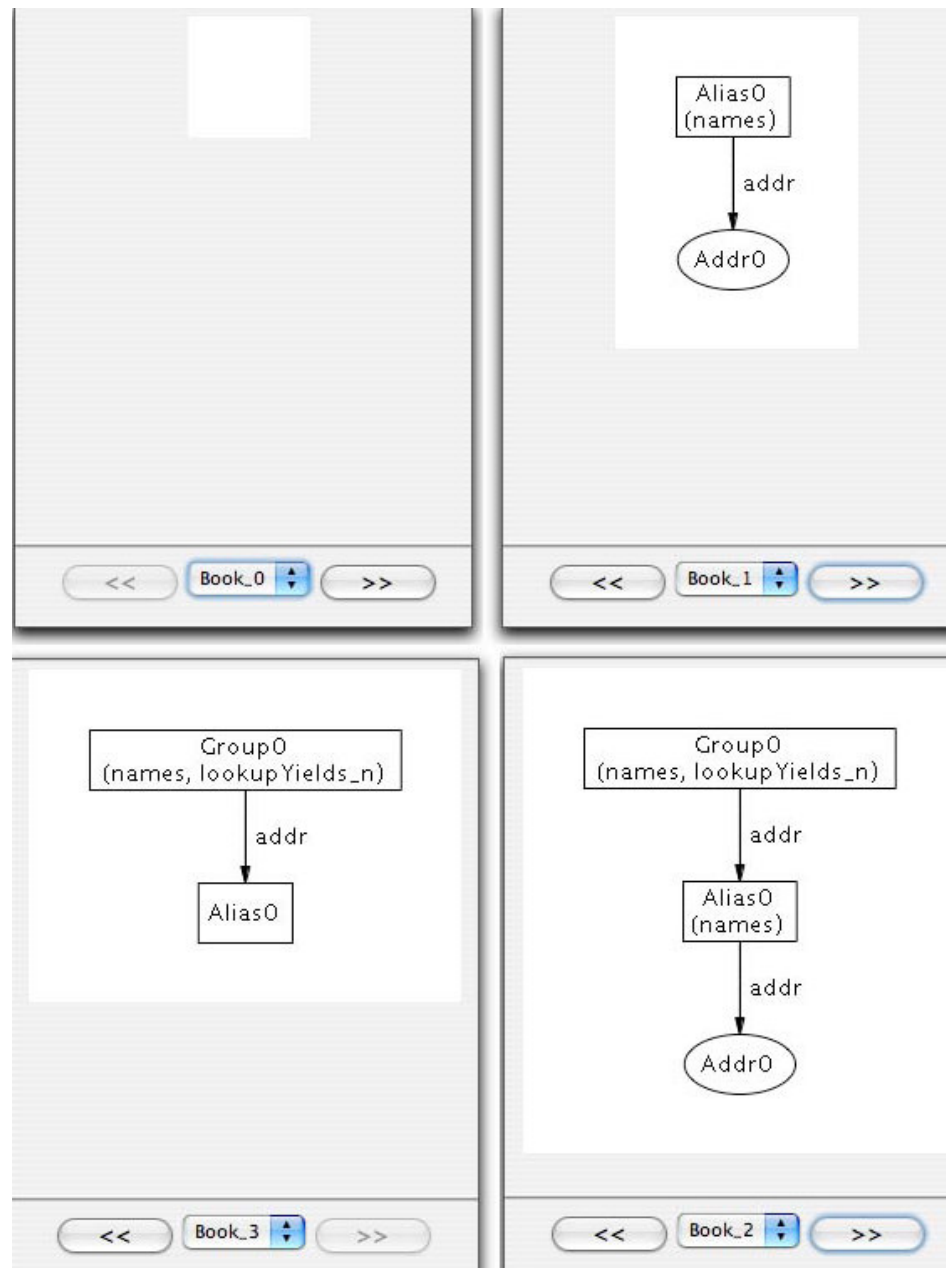
The problem here is that *add* allows a meaningless alias—one that refers to nothing—to be added to a group. To fix this, we might add a precondition to *add*, saying that the target given must either be an address, or else must resolve to at least one address on lookup:

```
pred add (b, b': Book, n: Name, t: Target) {  
  t in Addr or some lookup (b,t)  
  b'.addr = b.addr + n -> t  
}
```

Checking the assertion again, we get the counterexample of fig. 2.17.

# Violation again because of deletion

---



# Fix: Restrict deletion

---

```
pred del (b, b': Book, n: Name, t: Target) {  
  no b.addr.n or some n.(b.addr) - t  
  b'.addr = b.addr - n -> t  
}
```

The precondition says that  $n$  isn't itself mapped to, or it's mapped to some target besides  $t$ .

Now, no counterexample is found. So we crank up the scope to 6, and analyze for all scenarios involving 6 targets and 6 address books:

```
check lookupYields for 6
```

# checking safety properties

---

- can check safety property with one assertion
  - because now all states are reachable

```
pred safe[s: State] {...}

assert allReachableSafe {
  all s: State | safe[s]
}
```

- check addressBook invariant with one assertion
  - what's the difference between this safety check and checking that each operation preserves the invariant?

# non-modularity of abstract machine

---

- static traffic light model

```
sig Color {}  
sig Light {  
  color: Color  
}
```

- dynamic traffic light model with abstract machine
  - all dynamic components collected in one sig

```
sig Color {}  
sig Light {}  
sig State {  
  color: Light -> one Color  
}
```

# pattern: local state

---

- embed state in individual objects
  - variant of abstract machine
- move state/time signature out of first column
  - typically most convenient in last column

## global state

```
sig Color {}  
  
sig Light {}  
  
sig State {  
  color: Light -> one Color  
}
```

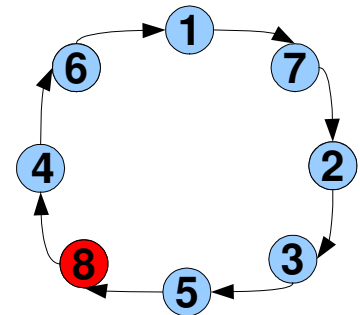
## local state

```
sig Time {}  
  
sig Color {}  
  
sig Light {  
  color: Color one -> Time  
}
```

# example: leader election in a ring

---

- many distributed protocols require “leader” process
  - leader coordinates the other processes
  - leader “elected” by processes, not assigned in advance
- leader is the process with the largest identifier
  - each process has unique identifier
- leader election in a ring
  - processes pass identifiers around ring
  - if identifier less than own, drops it
  - if identifier greater, passes it on
  - if identifier equal, elects itself leader



# leader election: topology

---

- beginning of model using local state abstract machine:
  - processes are ordered instead of given ids

```
open util/ordering[Time] as to
open util/ordering[Process] as po

sig Time {}
sig Process {
  succ: Process,
  toSend: Process -> Time,
  elected: set Time
}
```

- can be downloaded from the alloy web site
- constrain the successor relation to form a ring

# leader election: notes

---

- topology of the ring is static
  - *succ* field has no *Time* column

```
fact ring {  
  all p: Process | Process in p.^succ  
}
```

# leader election: notes

---

- no constraint that there be one elected process
  - that's a property we'd like to check
- set of elected processes is a definition
  - “elected” at one time instance then no longer

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

# leader election: operations

---

- write initialization condition *init[t: Time]*
  - every process has exactly itself to send

```
pred init [t: Time] {  
    all p: Process | p.toSend.t = p  
}
```

# leader election: operations

---

- write no-op operation *skip*[*t*, *t'*: *Time*, *p*: *Process*]
  - process *p* send no ids during that time step

```
pred skip [t, t': Time, p: Process] {  
    p.toSend.t = p.toSend.t'  
}
```

# leader election: operations

---

- write send operation *step*[*t*, *t'*: *Time*, *p*: *Process*]
  - process *p* sends one id to successor
  - successor keeps it or drops it

```
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 - p.succ.prevs)  
    }  
}
```

# leader election: traces

---

- use the following traces constraint

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

- why does traces fact need *step(t, t', succ.p)*?
- what's the disadvantage to writing this instead?

```
some p: Process | step[t, t', p] &&  
  all p': Process - (p + p.succ) | skip[t, t', p]
```

# Leader election: analysis

---

- At most one should be elected

```
assert AtMostOneElected {  
  lone elected.Time  
}  
check AtMostOneElected for 3  
Process but 7 Time
```

- What about

```
all t: Time | lone elected.t
```

# leader election: analysis

---

- check that at most one process is ever elected

```
assert AtLeastOneElected {  
  some t: Time | some elected.t  
}  
check AtLeastOneElected for 3 but 7 Time
```

```
pred progress {  
  all t: Time - to/last | let t' = t.next |  
    some Process.toSend.t =>  
      some p: Process | not skip[t, t', p]  
}
```

```
assert atLeastOneElected {  
  progress => some elected.Time  
}
```

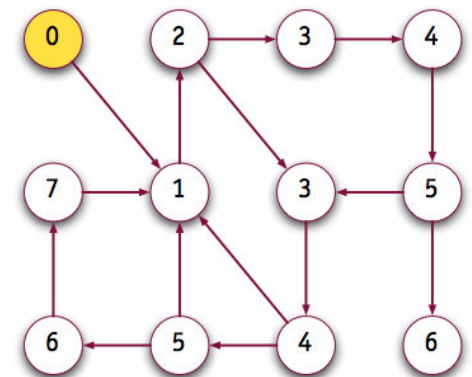
# machine diameter

---

- what trace length is long enough to catch all bugs?
  - does “at most one elected” fail in a longer trace?
- *machine diameter* = max steps from initial state
  - longest loopless path is an upper bound
- run this predicate for longer traces until no solution

```
pred looplessPath {  
  no disj t, t': Time | toSend.t = toSend.t'  
}  
run looplessPath for 3 Process, ? Time
```

- for three processes, what trace length is sufficient to explore all possible states?  
13



# Summary

---

- Just modeled the domain and the correct states (predicates, facts, assertions).
- Just described what are legal traces (with time).
- No code was written.
- Used only logic to describe valid states.

But:

- What about generating code???