

FM 2006 Alloy Tutorial

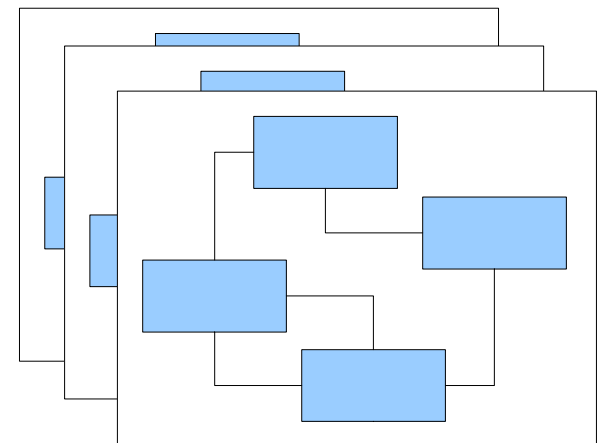
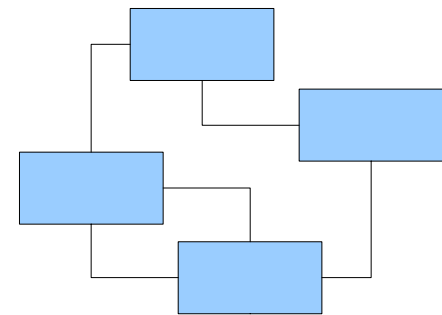
Dynamic Modeling

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



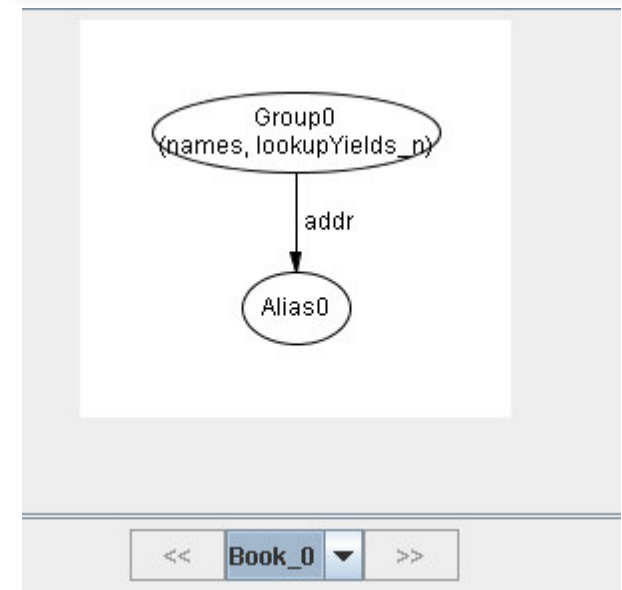
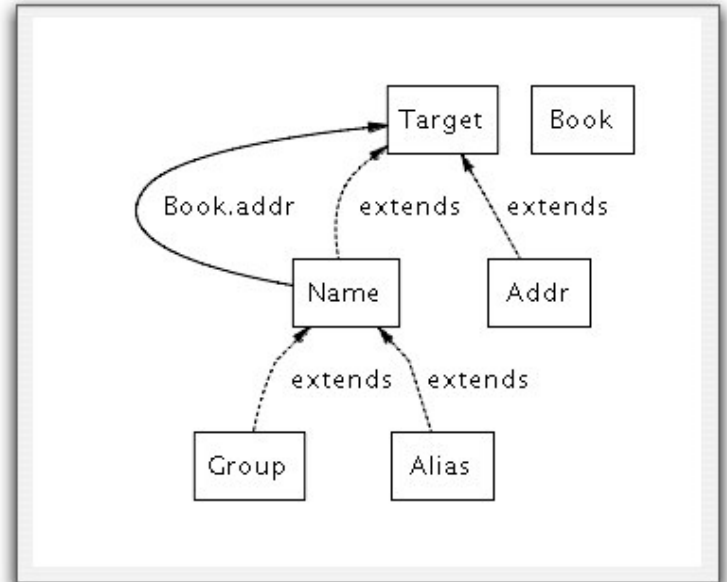
model of an address book

```
module tour/addressBook2
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
- 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

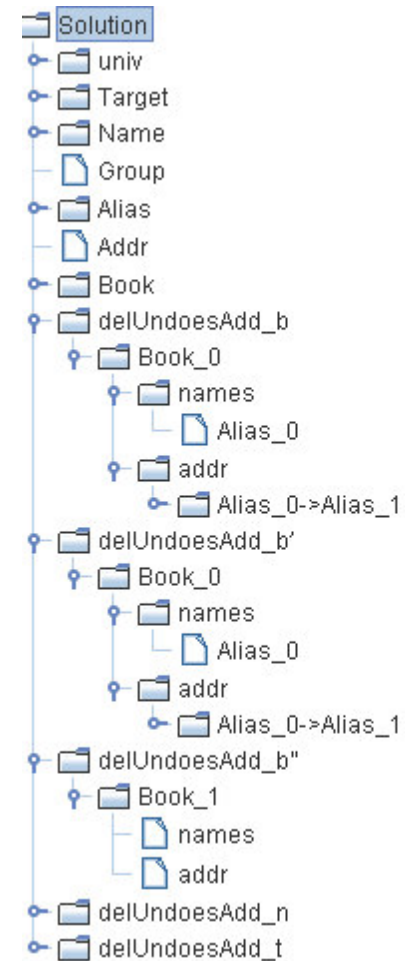
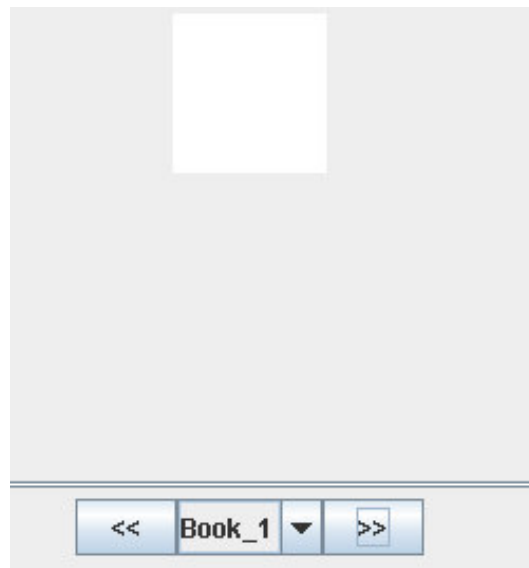
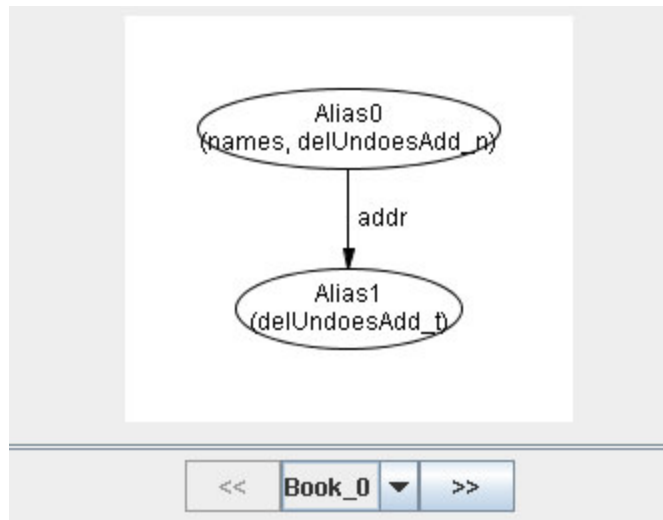
- The delete should remove a name mapping

```
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



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 --> no counter example
```

```
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
```

```
INFO : Command is: C:\TEMP\alloy18148\berkmin.exe -s solution.slv -r 0 -i 1 clauses.cnf  
INFO : berkmin.exe: initialized: new argc=2  
INFO : berkmin.exe: duration= 0.015  
INFO : berkmin.exe:  
INFO : berkmin.exe: =====> Outcome: 0  
No counterexample found: delUndoesAdd is valid within the specified scope. (00:01)
```

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')
}
```

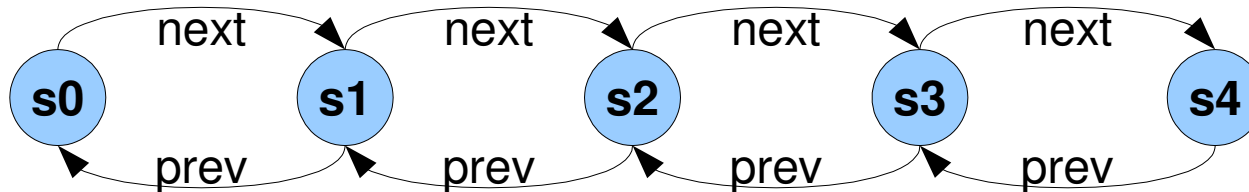
- apply traces pattern to the address book model

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

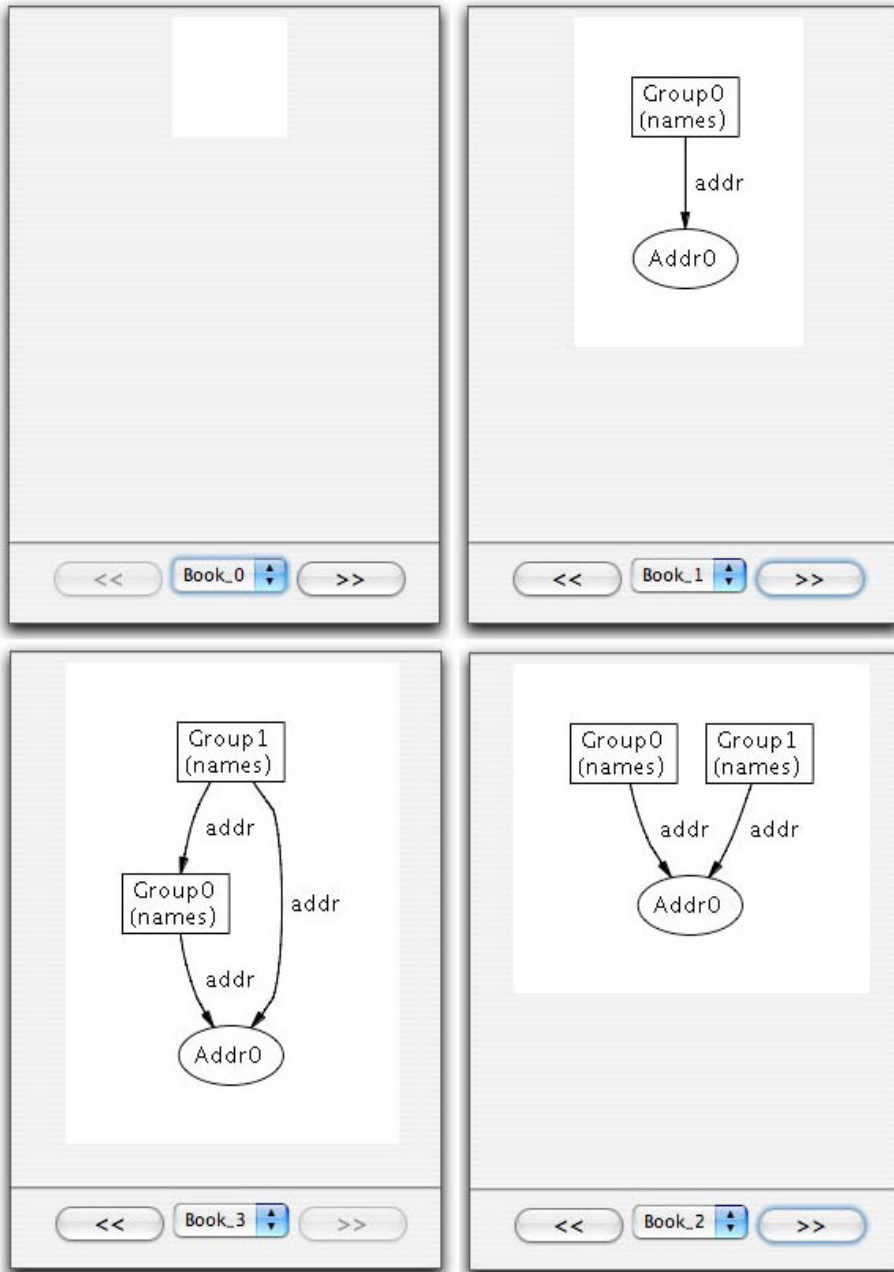
```
module tour/addressBook3
open util/ordering [Book]
...
pred init (b: Book) {no b.addr}

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

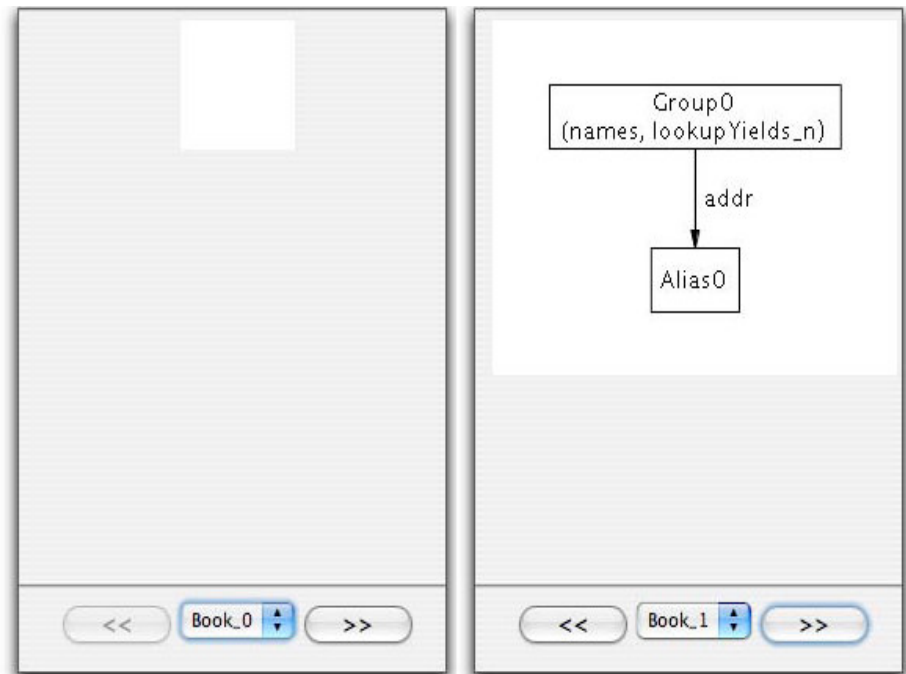


FIG. 2.16 Counterexample trace violating *lookupYields* with one step of *add*.

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

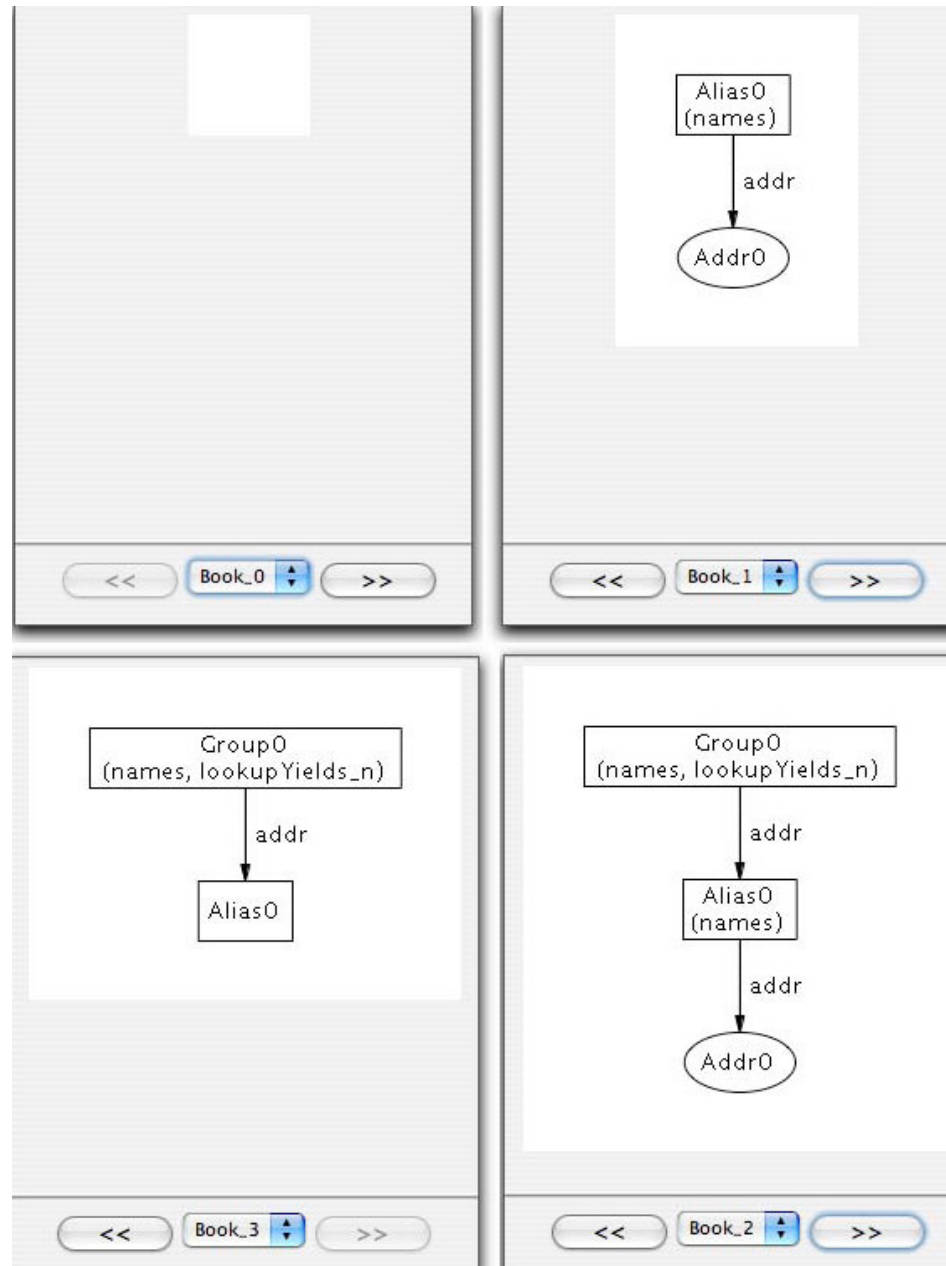
Solving meaningless adds

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
```

Summary

- Just modeled the domain and the correct states (predicates, facts, assertions).
- No code was written.
- Used only logic to describe valid states.

But:

- How about generating code???

pattern: operation preconditions

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

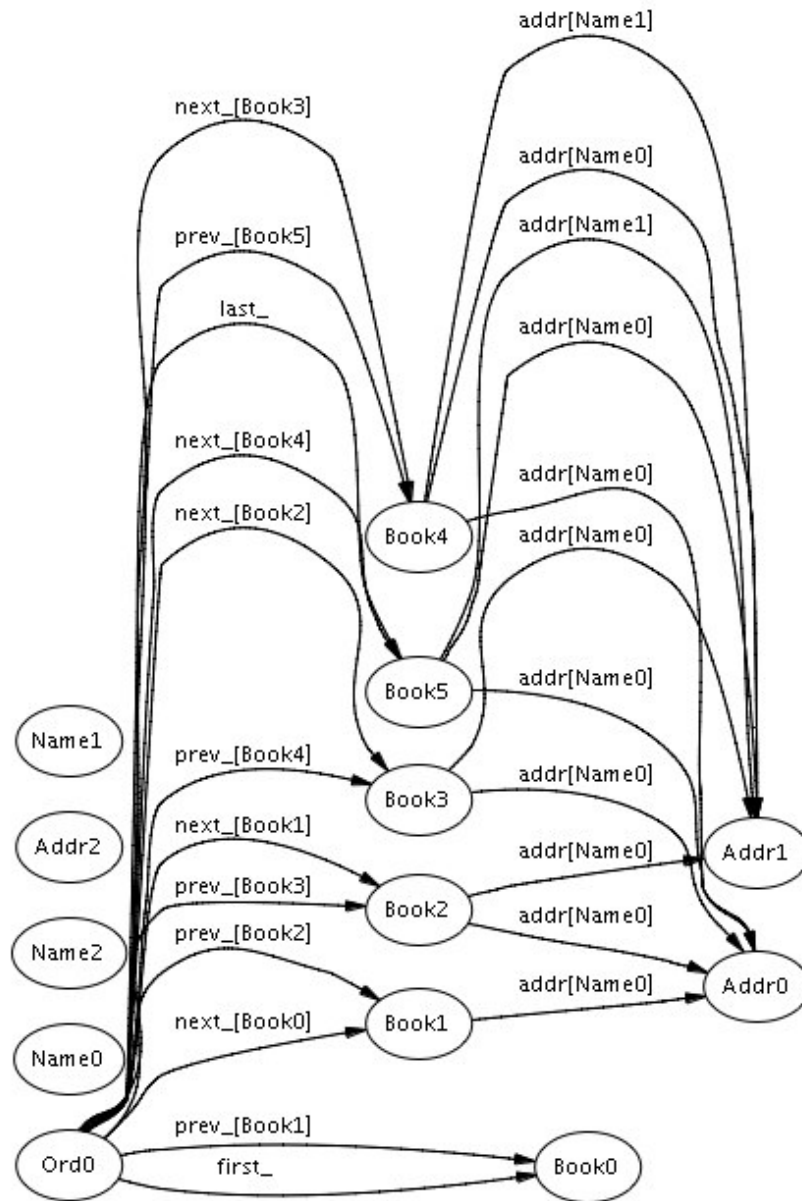
```
pred add(b, b': Book, n: Name, t: Target) {  
  // precondition  
  t in Name => (n !in t.*(b.addr) && some b.addr[t])  
  // postcondition  
  b'.addr = b.addr + n->t  
}
```

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

address book simulation

- simulate addressBook trace
 - write and run an empty predicate
- customize and cleanup visualization
 - remove all components of the Ord module
- but visualization is still complicated
- need to use projection . . .

still without projection



selecting projection

Views General util/ordering[examples/tutorial/addressBook/Book] alloy/lang/univ examples/tutorial/addressBook

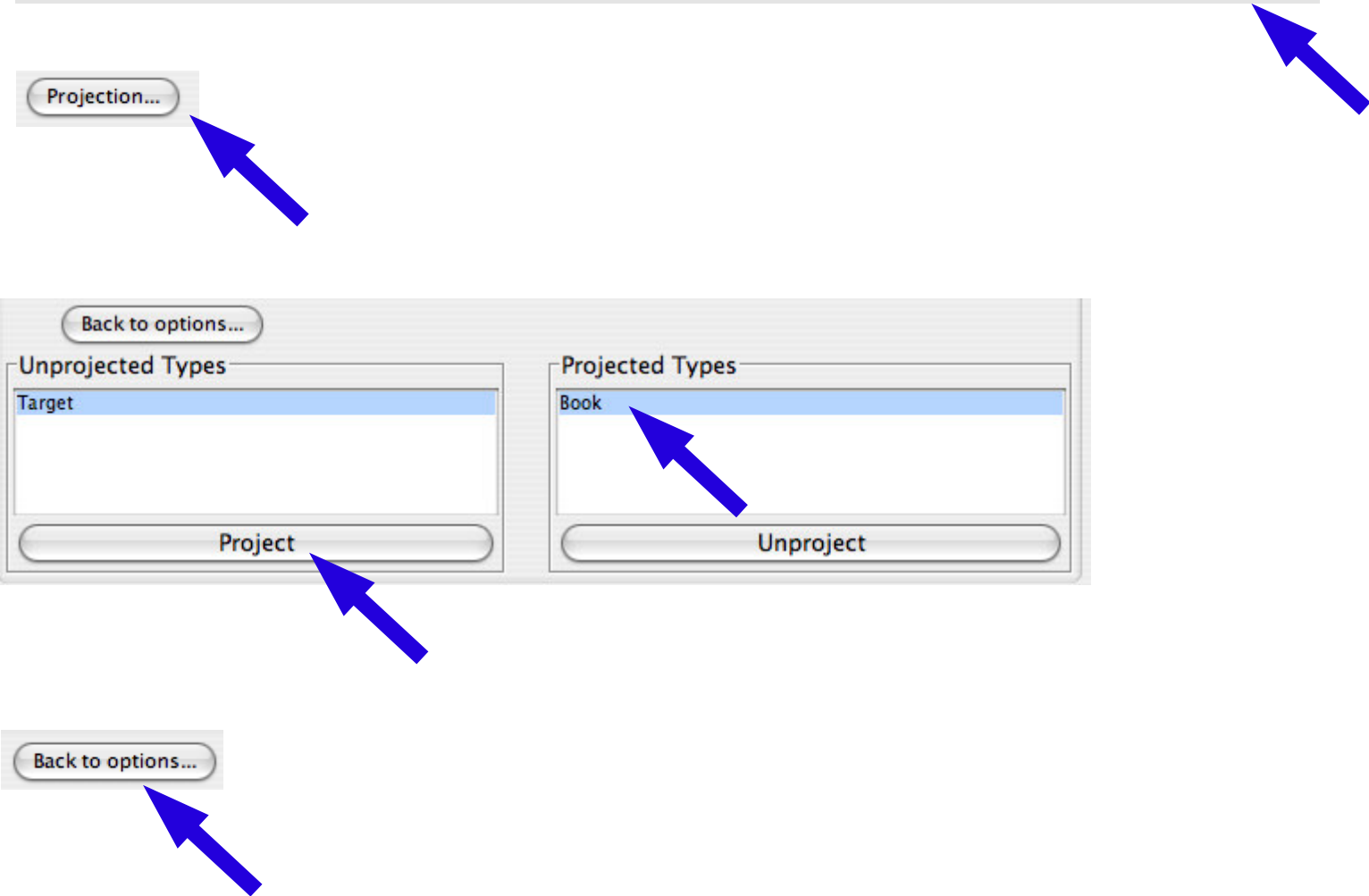
Projection...

Back to options...

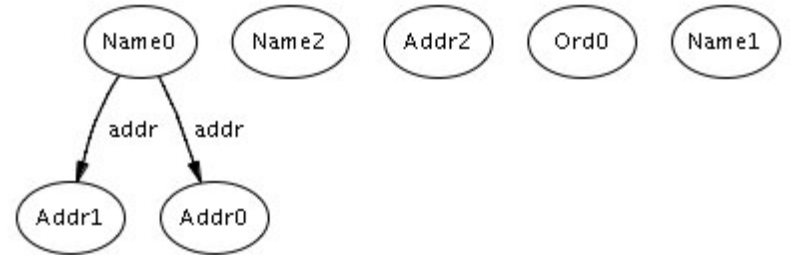
Unprojected Types	Projected Types
Target	Book

Project Unproject

Back to options...

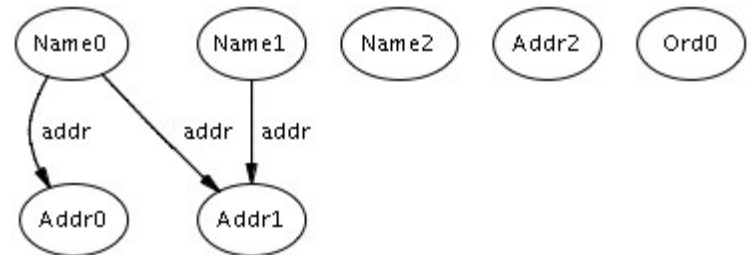
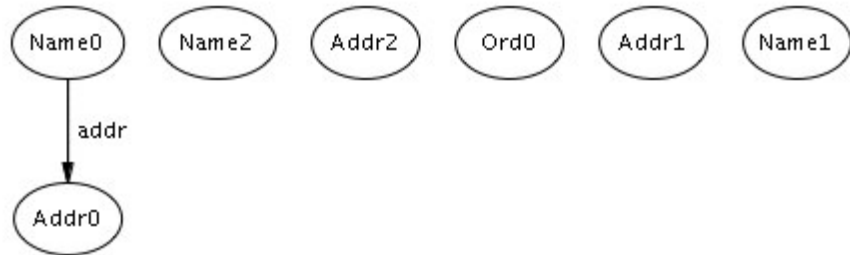


with projection



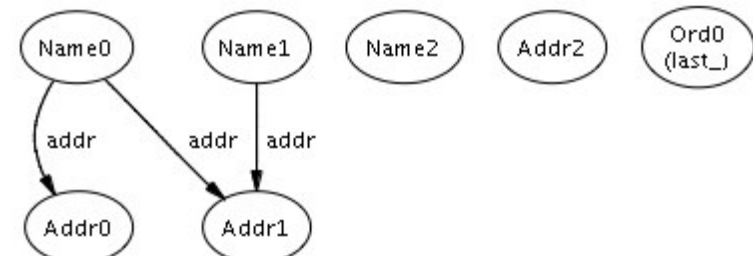
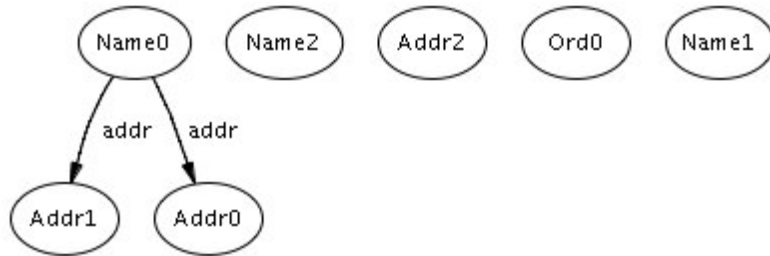
<< Book_0 >>

<< Book_3 >>



<< Book_1 >>

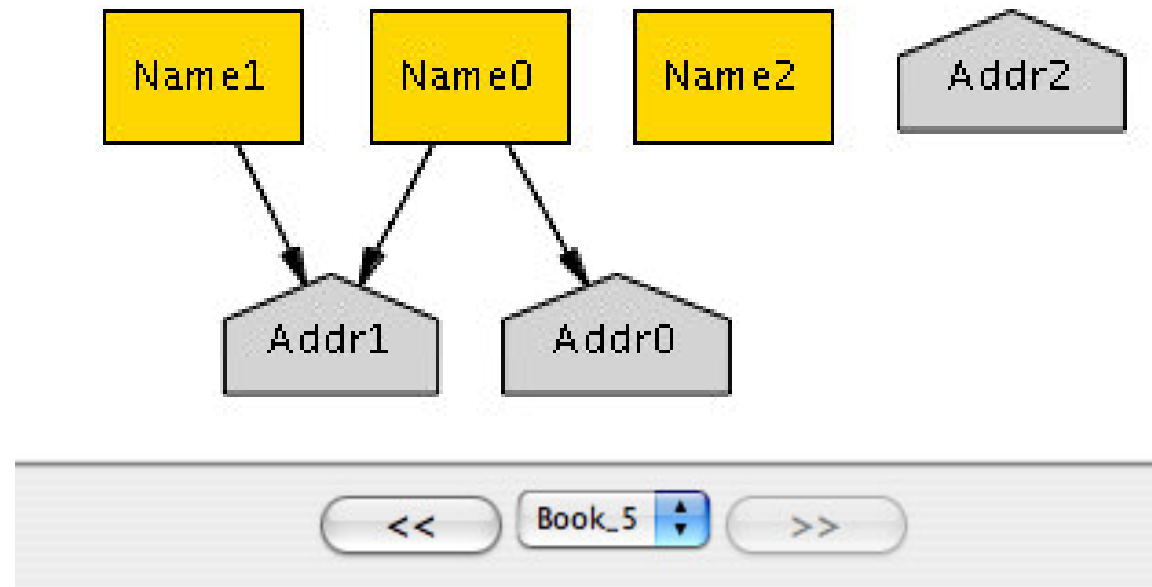
<< Book_4 >>



<< Book_2 >>

<< Book_5 >>

with projection and more



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  
}
```

thank you!

- websites
 - <http://alloy.mit.edu/>
 - <http://alloy.mit.edu/fm06/>
- provides . . .
 - online tutorial
 - reference manual
 - research papers
 - academic courses
 - sample case studies
 - alloy-discuss yahoo group

