CIS 771:
Software Specifications

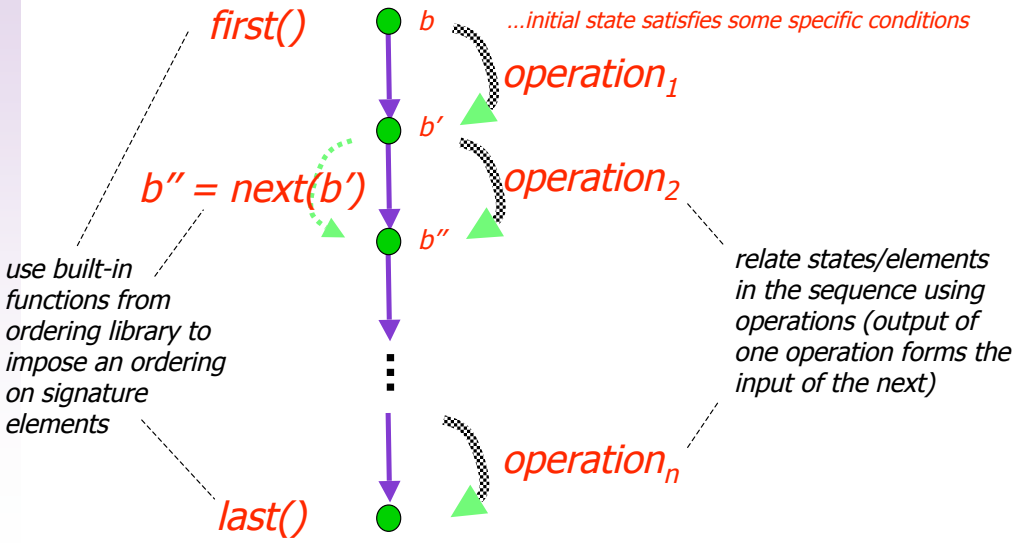
Lecture: Alloy Whirlwind Tour
(part E)

Outline

- Using the Alloy *ordering* library to establish an ordering on elements from a particular signature
- Using the order mechanism to reason about traces of system state changes realized by operations in our model

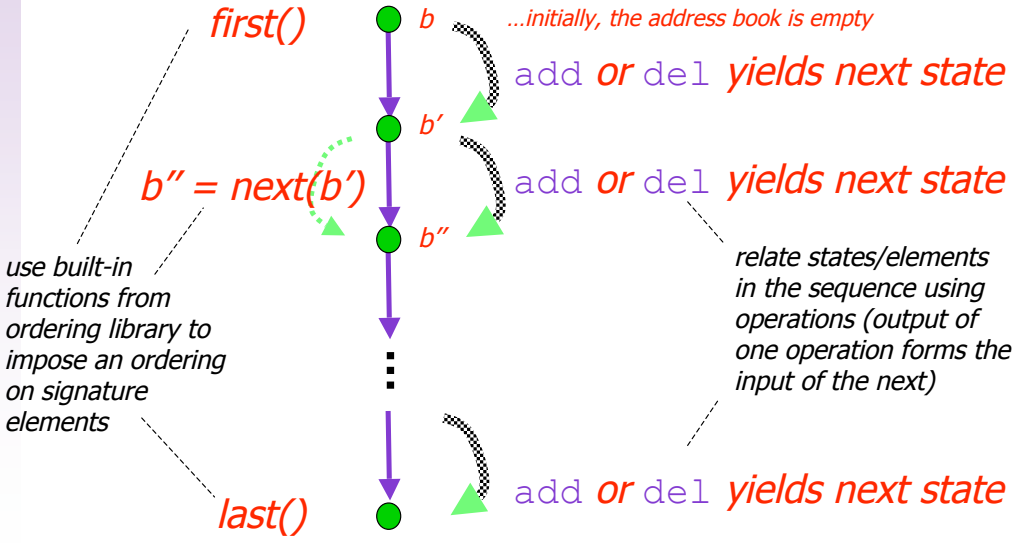
Modeling System Executions

Using different elements from a signature to represent different system states and state transitions…



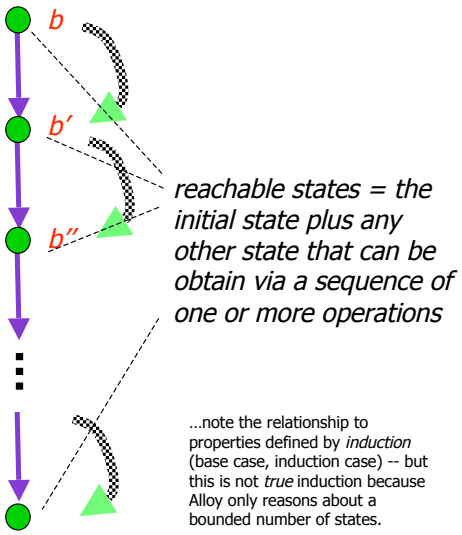
Modeling System Executions

Consider the address book example…



Modeling System Executions

How can we use this concept?



- What states can the system *reach* via the operations that we have defined?
  - Note: sometimes there are states that satisfy our model constraints that are actually not reachable via a sequence of operations -- we might want to avoid considering such states for some of our properties.
- Does every reachable state satisfy a certain property?
  - we refer to these as *invariant properties.*
- Is it possible for the system to reach a bad/error state?
- What are sequences of actions are possible?
  - i.e., must there always exist an add operation somewhere before a delete?
  - we refer to these as *temporal properties.*

Alloy Model with Traces

Setting up traces of address book states generated starting from an empty state and performing address book operations

```
module tour/addressBook3
open util/ordering [Book]as BookOrd
…
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]
```

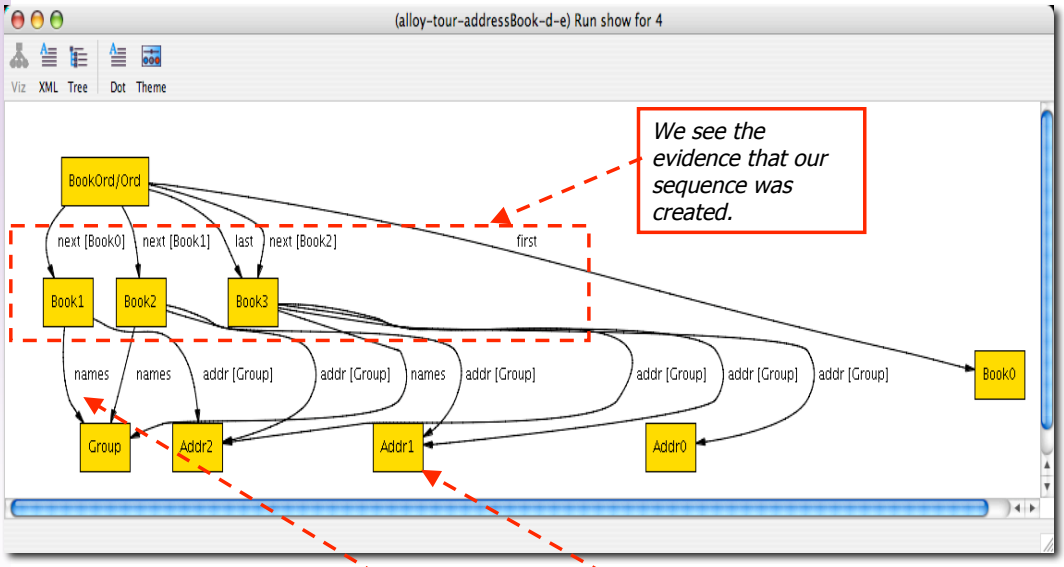
The initial book must be empty.

For all books except the last..

`b'` *is the successor of* `b`

`b'` *is the result of operating on* `b` *using* `add` *or* `del` *for some* `n` *and* `t`.

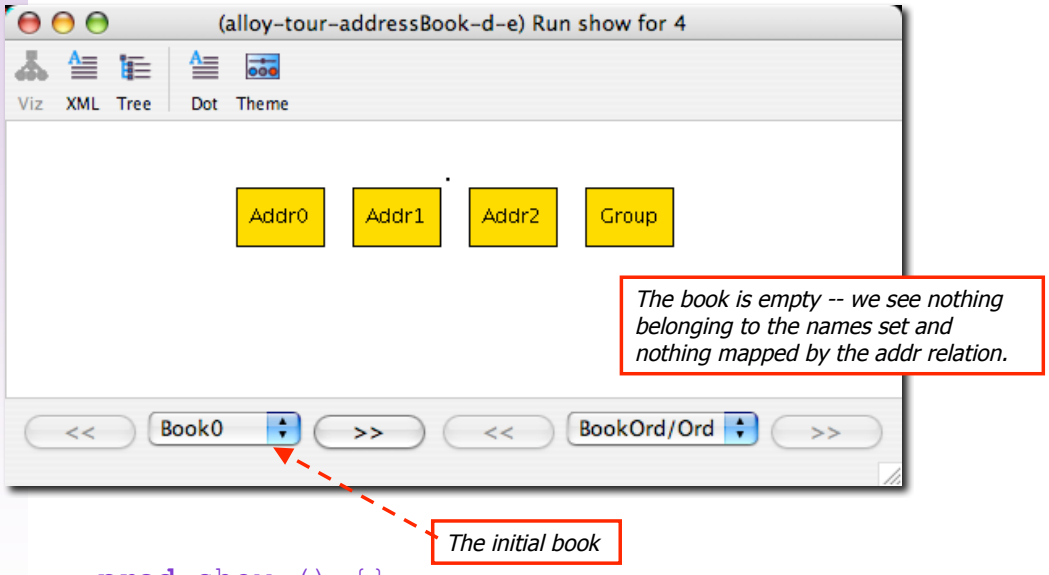
Visualize Instance




```
pred show () {}
run show for 4
```


Visualize Instance

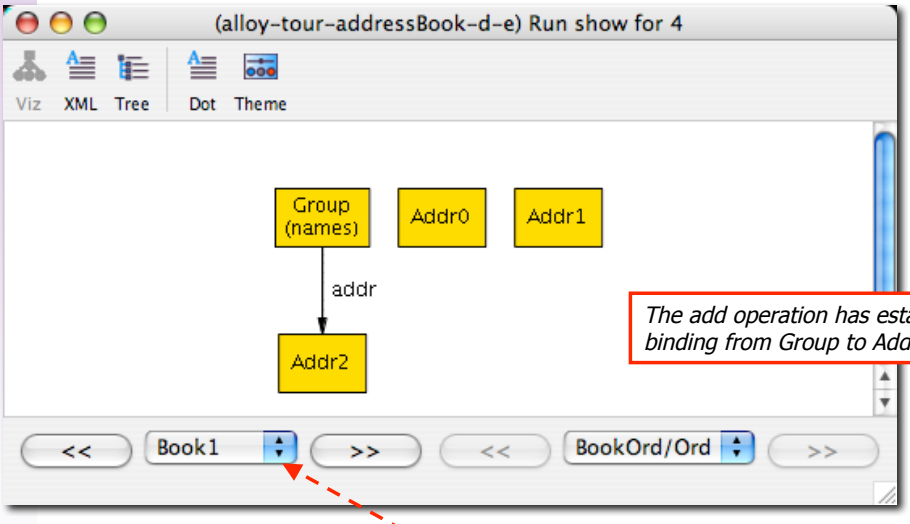
...project over Book and BookOrd/Ord




```
pred show () {}
run show for 4
```

Visualize Instance

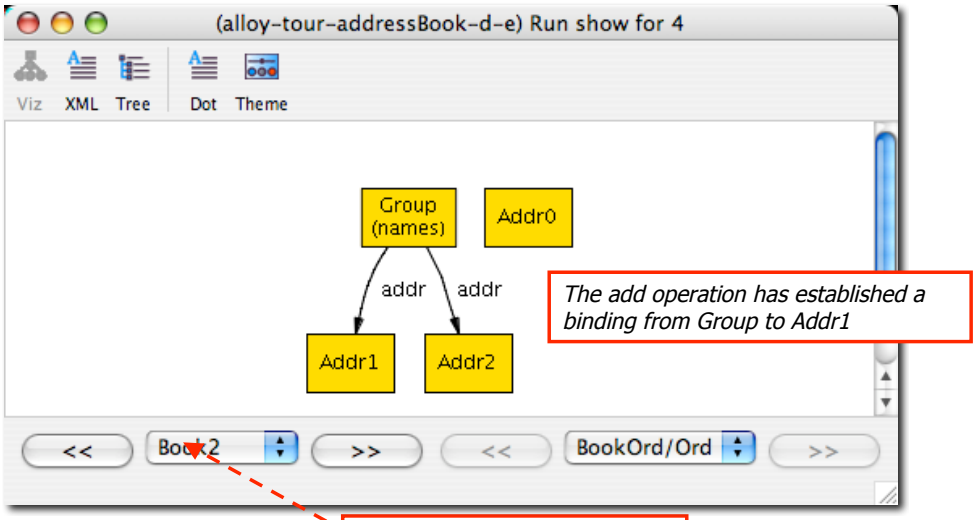
...project over Book and BookOrd/Ord




```
pred show () {}
run show for 4
```


Visualize Instance

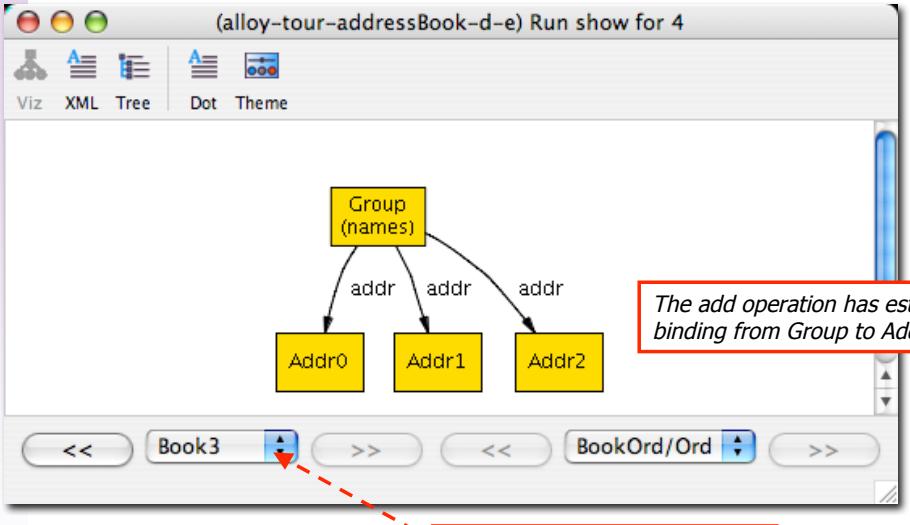
...project over Book and BookOrd/Ord




```
pred show () {}
run show for 4
```

Visualize Instance

...project over Book and BookOrd/Ord




```
pred show () {}
run show for 4
```


For You To Do

- Our previous run of the tool only produced traces that contained add operations.
- Define a new version of the traces predicate that explicitly describes a sequence of *four* books where (after the initial book) the second and third books are generated via an add operation and the fourth book is generated via a deletion operation.
- Hint: you will need to explicitly introduce variables such as b, b′, b″, and b‴, relate each of these via the next operation and also relate each using the add and delete operations.

Checking Assertions

Let us return the problem of last lecture -- we might have a situation where looking up the addresses associated with (reachable from!) a key yields nothing...

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

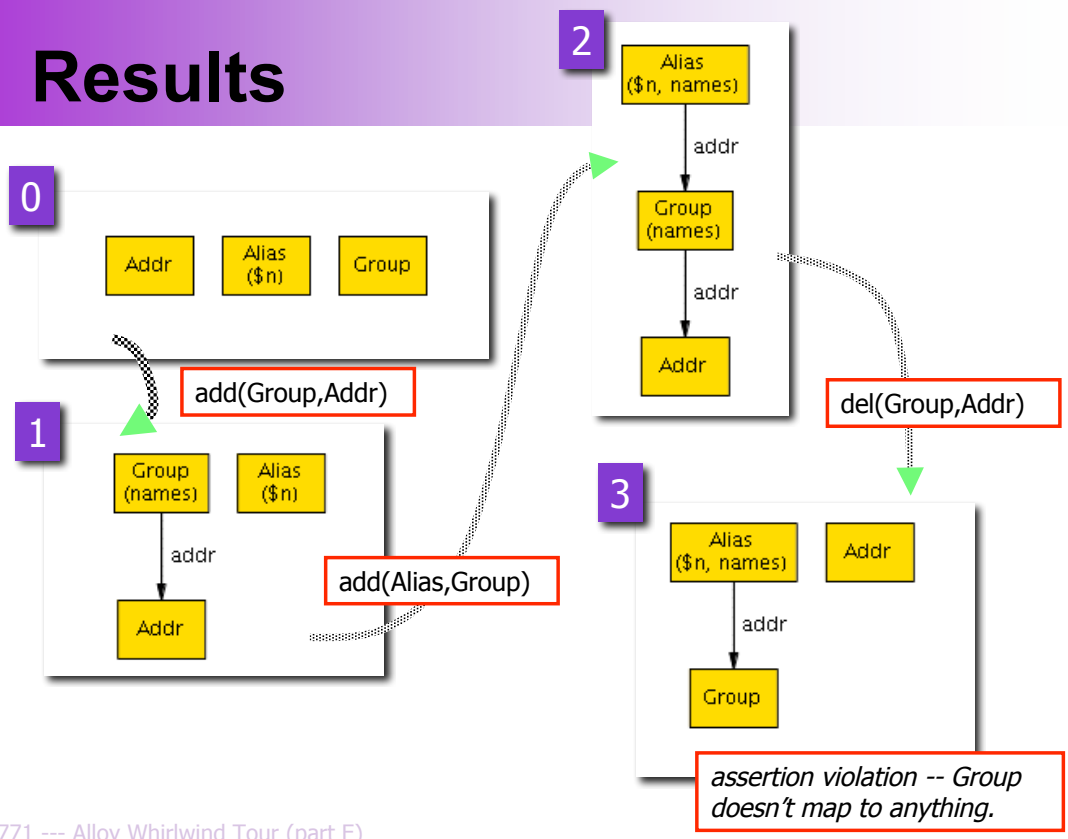
Recall definition of `lookup`

Assert and check that lookup always yields something

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

Simple *invariant property* -- a property that we want to hold for all reachable states

Results



Assessment

- It turns out the assertion can fail for multiple reasons…
  - an "improper" delete (as illustrated on the previous slide)
  - an add operation that adds a group as a target without first defining targets for the group (as illustrated in the book)
- We can modify our operations to add preconditions to try to rule out these situations.

Modified Add Operation

Add a precondition requiring the target to be an addr (a "leaf" in the lookup chain) or something that already has addresses associated with it via the lookup operation.

New precondition…

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

Modified Delete Operation

Add a precondition requiring that nothing maps to *n* (so removing won't result in a "dangling reference") and that there is still something left associated with *n* even after the target *t* is removed.

New precondition…

```
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}
```

…with changes in these two operations, no violations are found for lookupYields (even for scope of 6)

Summary

- Using Alloy's ordering library, we can arrange the instances of a particular signature (e.g., Book) in a sequence.
- Once we have a sequence, we can…
  - establish conditions on the initial element in the sequence (base case),
  - describe how each element in the sequence is related to its successor (induction case)…
    - here, we can relate a state to it's successor using operations to model an execution sequence that yields the *reachable states* of a system.
    - we need not use an inductive style, but also can also define an explicit sequence of certain operations.
- Once this framework is set up, we can check…
  - invariants -- properties that should hold for all reachable states
  - temporal ordering properties (we will see examples later) that impose constraints on the order in which operations can occur.

For You To Do

- Using the final model in which preconditions have been added for operations, modify the original traces structure so that the initial book has two entries in it.
- Write a constraint and use the solver to show that, via a sequence of operations, it is possible to get from the initial book with two entries in it to an empty book.

Acknowledgements

- The material in this lecture is based on Section 2.4 from…
  - *Software Abstractions: Logic, Language, and Analysis,* Daniel Jackson, MIT Press, 2006.