Outline

- Operations -- modeling state transitions
  - pre-states
  - post-states
- Operation contracts
- Address Book operations
  - add name & address entry
  - remove entry
Dynamic Models

- Static models allow us to describe the legal states of the system
- We also want to be able to describe the legal transitions between states
  - A (name, addr) pair are added to the address book
  - An entry is removed from the address book
  - A new book is created

Modeling State Transitions

- In declarative modeling,
  - there is no “assignment operator”
  - we don’t do “destructive updates” or “state mutations”
- For every operation execution or state transition, there is a...
  - pre-state -- the state before the operation is executed
  - post-state -- the state after an operation is executed
Example Operations

...adding a new book

Operational view: Operation transforms the pre-state into the post-state

Declarative view: Operation relates the pre-state and the post-state

Example Operations

...removing an address book entry

Operational view: Operation transforms the pre-state into the post-state

Declarative view: Operation relates the pre-state and the post-state
Modeling State Transitions

- Alloy only has one “state”
  - e.g., there is really only one set of objects for each signature
- We will simulate having both a pre-state and post-state (and later on, a sequence of states) by using some conventions for variable names
  - b.addr -- represents b’s address field in the pre-state
  - b’.addr -- represents b’s address field in the post-state

In reality, there is only a single set Book in Alloy

We will represent a transformation on Book using two different book objects: b1 represents the original value of the book in the pre-state and b1’ represents the modified value of the book in the post-state.
Modeling State Transitions

- Specifications should generally focus on “what” instead of “how”
- Our specification of operations will focus on
  - Pre-conditions: what properties are expected/assumed of the pre-state if the operation is to work correctly
    - e.g., an address book delete of person N might assume that N already exists in the book
  - Post-conditions: what properties does the operation establish in the post-state
    - e.g., N and any associated addresses don’t exist in the address book after delete
  - Frame-conditions: what stayed the same
    - e.g., no other entries in the current book were removed

Software Contracts

- Service
  - service can rely on the fact that the pre-condition holds
  - frame condition guarantees that service only performs changes mentioned in the post-condition -- it doesn't interfere or mess with anything else.
  - service guarantees that the post-condition holds

- Client
  - client guarantees that it establishes the pre-condition of the service
  - client can rely on the fact that the service post-condition holds
Modeling \textit{Add Address}

\begin{verbatim}
pred add2(b, b': Book, n:Name, a:Addr) {
  // pre-condition
  // n is not currently in the book
  no n.(b.addr)

  // post-condition
  // invoking b' addr map on n yield’s a
  n.(b'.addr) = a

  // frame-condition
  // for all other names, the addr map should
  // yield the same value
  all n1: (Name - n) | n1.(b.addr) = n1.(b'.addr)
}
\end{verbatim}

\textit{Results}

\begin{verbatim}
run add2 for 2 Book, exactly 4 Name, exactly 4 Addr
\end{verbatim}
Results (alternate)  

Name3 not associated with Addr3 for pre-state book b

Frame conditions ensure that Name1, and Name2 have the same mapping (to Addr2) in both the pre- and post-state book.

Name3 is associated with Addr3 for post-state book b'

run add2 for 2 Book, exactly 4 Name, exactly 4 Addr
CIS 771 --- Alloy Whirlwind Tour (part C)

Assessment

- The tool support and methodology for modeling operations is not perfect
  - default ordering that Alloy assigns to generated names does not necessarily follow pre/post order
  - Naming convention using primes is just a convention -- Alloy really doesn’t understand that, e.g., b and b’ should be different versions of the same object. Sometimes you may need to force b != b’.
Assessment

Pre-state book is named Book1 and post-state book is named Book0

...therefore, in project over Book visualization, the post-state gets displayed before the pre-state

Consider the following definition of add...

```plaintext
pred add(b, b': Book, n: Name, a: Addr) {
  b'.addr = b.addr + n->a
}
```

run add for 3 but 2 Book
Assessment

Consider the following definition of add...

```plaintext
pred add(b,b': Book, n: Name, a: Addr) {  
  b != b'  
  b'.addr = b.addr + n->a  
}
run add for 3 but 2 Book
```

Add a constraint to force representation of the pre- and post-state books using two different objects.
**Assessment**

Now we have distinct pre- and post-state versions of the book object...

...and we see the new entry where Name2 is bound to Addr.

---

**Delete Operation**

Version of Delete without precise pre/post-conditions...

```
pred del(b, b': Book, n: Name) {
    b'.addr = b.addr - n->Addr
}
```

Intuition: take the entries in b.addr and throw away all tuples in which n is paired with any address.
For You To Do

- Visualize the behavior of the delete operation on the previous slide
- Add any constraints and/or make scope adjustments as we have discussed previously to create an interesting instance

Lookup Operation

Using the Alloy function construct

```alloy
fun lookup(b: Book, n: Name) {
  n.(b.addr)
}
```

- Since lookup returns a value, use an Alloy function construct instead of a predicate
- Note: lookup is “read only” -- it doesn’t cause a state transformation. Therefore, we have no “primed” variables.
Checking Operation Composition

Using the Alloy assertion construct

```alloy
assert delUndoesAdd {
    all b,b',b'': Book, n:Name, a:Addr |
    add[b,b',n,a] and del[b',b'',n]
    implies b.addr = b''.addr
}
```

```alloy
check delUndoesAdd for 3
```

- Intuition: we claim (assert) that if we add \((n,a)\) to \(b\) to get \(b'\), then remove from \(b'\) any associations with \(n\) to get \(b''\), then the addr mapping of \(b\) and \(b''\) should be equal.
- run/pred tells the analyzer to look for an example that satisfies pred
- check/assert tells the analyzer to look for a counterexample that falsifies assert

---

Result

A counterexample (assertion violation) is found...

```
If \(b\) and \(b'\) are the same (i.e., if \((n,a)\) already existed in \(b\).addr before the add operation)...
```

...then removing all bindings from \(n\) to obtain \(b''\) gives us a \(b''\) that is different than \(b'\)

This outcome is tied to our previous observation: the definition of add did not include a pre-condition requiring \(n\) to be absent from the book.
Checking Operation Composition

**Modified version...**

```plaintext
assert delUndoesAdd {
  all b,b',b'': Book, n:Name, a:Addr |
  no n.(b.addr) and 
  add[b,b',n,a] and del[b',b'',n] 
  implies b.addr = b''.addr
}

check delUndoesAdd for 3

- ...now the analyzer finds no counter-examples
```

**Assertion checking idempotency of add**

```plaintext
assert addIdempotent {
  all b,b',b'': Book, n:Name, a:Addr |
  add[b,b',n,a] and add[b',b'',n,a] 
  implies b''.addr = b''.addr
}

- Intuition: repeating the add operation yields no additional effect
- Outcome: no counterexamples found
```
Checking Operation Composition

Assertion checking non-interference of add

```plaintext
assert addLocal {
    all b,b': Book, n,n':Name, a:Addr |
    add[b,b',n,a] and n != n'
    implies lookup[b,n'] = lookup[b',n']
}
```

- **Intuition:**
  - checks the frame condition for add that we discussed earlier
  - adding an address for n doesn’t change the bindings for any of
    the other names in the book
- **Outcome:** no counterexamples found

Conclusions

- In addition to specifying the legal states of a system, Alloy enables
  us to describe the legal transitions between states.
  - We described such transitions as operations that take as input a pre-
    state and return a post-state.
- In high-level specifications, we aim to specify what abstraction
  properties/constraints a state transition (operation) must satisfy --
  not how the operation is actually carried out.
- Declarative specifications of operations are often broken down into
  three parts
  - pre-conditions (properties that we assume about the pre-state)
  - post-conditions (properties that the operation guarantees of the post-
    state)
  - frame-conditions (constraints that identify explicitly which portions of
    the state are allowed to modified by the operation).
For You To Do

- Define an alternative version of the `del` operation that includes pre-, post-, and frame-conditions similar to the definition `add2` given in this lecture.
- Analyze your definition using several scope settings to assess its validity.
- Create an assertion `delidempotent` (that applies to the version of delete that you defined above) similar to `addidempotent` assertion defined in this lecture and check the assertion using several different scope settings. What can you conclude about the validity of the assertion?

Acknowledgements

- The material in this lecture is based on Section 2.2 from...