Alloy Analyzer 4 Tutorial

Session 4: Dynamic Modeling

Greg Dennis and Rob Seater
Software Design Group, MIT
model of an address book

```plaintext
abstract sig Target {}
sig Name extends Target {}
sig Addr extends Target {}

sig Book { addr: Name -> Target }

pred init [b: Book] { no b.addr }

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

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

assert namesResolve {
    all b: Book | inv[b] =>
        all n: Name | some b.addr[n] => some lookup[b, n]
}

check namesResolve for 4
```
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:

```haskell
pred add [b, b': Book, n: Name, t: Target] {
  b'.addr = b.addr + n->t
}
```
address book: operation simulation

• simulates an operation's executions
  ➢ download *addressBook.als* from the tutorial website
  ➢ execute run command to simulate the *add* operation
    – simulated execution can begin from invalid state!
  ➢ create and run the predicate *showAdd*
    – simulates the add method only from valid states
      
      *pred* showAdd [b, b': Book, n: Name, t: Target] {
        inv[b]
        add[b, b', n, t]
      }

  ➢ modify *showAdd* to force interesting executions
address book: delete operation

➢ write a predicate for a *delete* operation
  – removes a name-target pair from a book
  – simulate interesting executions

➢ assert and check that delete is the undo of add
  – adding a name-target pair and then deleting that pair yields a book equivalent to original
  – why does this fail?

➢ modify the assertion so that it only checks the case when the added pair is not in the pre-state book, and check
pattern: abstract machine

- treat actions as operations on global state

\[
\begin{align*}
\text{sig} & \text{ State } 
\{ ... \} \\
\text{pred} & \text{ init } [s: \text{ State}] \{ ... \} \\
\text{pred} & \text{ inv } [s: \text{ State}] \{ ... \} \\
\text{pred} & \text{ op1 } [s, s': \text{ State}] \{ ... \} \\
\text{...} \\
\text{pred} & \text{ opN } [s, s': \text{ State}] \{ ... \}
\end{align*}
\]

- in addressBook, \textit{State} is \textit{Book}
  - each \textit{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:

```java
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

```plaintext
open util/ordering[State] as ord
...

fact traces {
  init [ord/first]
  all s: State - ord/last |
  let s' = s.next |
    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]
```

\[ S = s_0 + s_1 + s_2 + s_3 + s_4 \]

- \( \text{first} = s_0 \)
- \( \text{last} = s_4 \)
- \( s_2 \text{.next} = s_3 \)
- \( s_2 \text{.prev} = s_1 \)
- \( s_2 \text{.nexts} = s_3 + s_4 \)
- \( s_2 \text{.prevs} = s_0 + s_1 \)
- \( \text{lt}[s_1, s_2] = \text{true} \)
- \( \text{lt}[s_1, s_1] = \text{false} \)
- \( \text{gt}[s_1, s_2] = \text{false} \)
- \( \text{lte}[s_0, s_3] = \text{true} \)
- \( \text{lte}[s_0, s_0] = \text{true} \)
- \( \text{gte}[s_2, s_4] = \text{false} \)
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 . . .
without projection
still without projection
with projection
with projection and more
checking safety properties

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

```plaintext
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

<table>
<thead>
<tr>
<th>global state</th>
<th>local state</th>
</tr>
</thead>
<tbody>
<tr>
<td>sig Color {}</td>
<td>sig Time {}</td>
</tr>
<tr>
<td>sig Light {}</td>
<td>sig Color {}</td>
</tr>
<tr>
<td>sig State {</td>
<td>sig Light {</td>
</tr>
<tr>
<td>color: Light -&gt;</td>
<td>color: Color one -&gt; Time</td>
</tr>
<tr>
<td>one Color }</td>
<td>}</td>
</tr>
</tbody>
</table>
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

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

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

- download `ringElection.als` from the tutorial website
- constrain the successor relation to form a ring
leader election: notes

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

```plaintext
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

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

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

➢ simulate interesting leader elections

➢ create intuitive visualization with projection

➢ check that at most one process is ever elected
  – no more than one process is deemed elected
  – no process is deemed elected more than once

➢ check that at least one process is elected
  – check for 3 processes and 7 time instances
  – write additional constraint to make this succeed
ordering module and exact scopes

- ordering module forces signature scopes to be exact

\[
\begin{align*}
3 \text{ Process, 7 Time} & \equiv \text{exactly 3 Process, exactly 7 Time}
\end{align*}
\]

- to analyze rings up to \( k \) processes in size:

\[
\begin{align*}
sig & \text{ Process } \\ sig & \text{ RingProcess extends Process } \\
& \text{ succ: RingProcess, } \\
& \text{ toSend: RingProcess } \rightarrow \text{ Time, } \\
& \text{ elected: set Time }
\end{align*}
\]

\[
\text{fact } \{ \text{all } p: \text{ RingProcess } | \text{ RingProcess in } p.\^\text{succ} \} \]
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

```plaintext
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?
thank you!

- website
  - http://alloy.mit.edu/

- provides . . .
  - online tutorial
  - reference manual
  - research papers
  - academic courses
  - sample case studies
  - alloy-discuss yahoo group