Alloy Logic

Excerpted (mostly), adapted (a bit) from:
SAIL Tutorial at
http://alloy.mit.edu/alloy/tutorials/day-course/
by Greg Dennis and Rob Seater
Alloy = logic + language + analysis

• logic
  – first order logic + relational calculus

• language
  – syntax for structuring specifications in the logic

• analysis
  – bounded exhaustive search for counterexample to a claimed property using SAT
logic: relations of atoms

- atoms are Alloy's primitive entities
  - indivisible, immutable, uninterpreted
- relations associate atoms with one another
  - set of tuples, tuples are sequences of atoms
- every value in Alloy logic is a relation!
  - relations, sets, scalars all the same thing
logic: everything is a relation

• sets are unary (1 column) relations
  \[ \text{Name} = \{(N0), \quad \text{Addr} = \{(A0), \quad \text{Book} = \{(B0),} \]
  \[ \quad \text{(N1), \quad \text{(A1), \quad \text{(B1)}} \]
  \[ \quad \text{(N2)} \quad \text{(A2)} \}\]

• scalars are singleton sets
  \[ \text{myName} = \{(N1)\} \]
  \[ \text{yourName} = \{(N2)\} \]
  \[ \text{myBook} = \{(B0)\} \]
logic: everything is a relation

• binary relation
  
  names = {(B0, N0),
            (B0, N1),
            (B1, N2)}

• ternary relation
  
  addr s = {(B0, N0, A0),
            (B0, N1, A1),
            (B1, N1, A2),
            (B1, N2, A2)}
logic: relations

\[ \text{addr} = \{ (B0, N0, A0), (B0, N1, A1), (B1, N1, A2), (B1, N2, A2) \} \]

- rows are unordered
- columns are ordered but unnamed
- all relations are first-order
- relations cannot contain relations, no sets of sets
logic: address book example

Name = \{(N0), (N1), (N2)\}
Addr = \{(A0), (A1), (A2)\}
Target = \{(N0), (N1), (N2), (A0), (A1), (A2)\}
address = \{(N0, A1), (N1, N2), (N2, A1), (N2, A0)\}
logic: constants

<table>
<thead>
<tr>
<th>none</th>
<th>empty set</th>
</tr>
</thead>
<tbody>
<tr>
<td>univ</td>
<td>universal set</td>
</tr>
<tr>
<td>iden</td>
<td>identity relation</td>
</tr>
</tbody>
</table>

Name = \{(N0), (N1), (N2)\}
Addr = \{(A0), (A1)\}

none
univ
iden
logic: set operators

\[\begin{align*}
+ & \quad \text{union} \\
\& & \quad \text{intersection} \\
- & \quad \text{difference} \\
in & \quad \text{subset} \\
equals & \quad \text{equality}
\end{align*}\]

Name = \{(N0), (N1), (N2)\}
Alias = \{(N1), (N2)\}
Group = \{(N0)\}
RecentlyUsed = \{(N0), (N2)\}

Alias + Group
Alias \& RecentlyUsed
Name - RecentlyUsed
RecentlyUsed in Alias
RecentlyUsed in Name
Name = Group + Alias

cacheAddr = \{(N0, A0), (N1, A1)\}
diskAddr = \{(N0, A0), (N1, A2)\}

cacheAddr + diskAddr
cacheAddr \& diskAddr
cacheAddr = diskAddr
logic: product operator

\[ p \rightarrow q \equiv \{(p_1, \ldots, p_n, q_1, \ldots, q_m) \mid (p_1, \ldots, p_n) \in p \land (q_1, \ldots, q_m) \in q\} \]

Name = \{(N0), (N1)\}       Name->Addr
Addr = \{(A0), (A1)\}
Book = \{(B0)\}             Book->Name->Addr =

\( b = \{(B0)\} \)
\( b' = \{(B1)\} \)
address = \{(N0, A0), (N1, A1)\}
address' = \{(N2, A2)\}

\( b \rightarrow b' \)

\( b \rightarrow \text{address} + b' \rightarrow \text{address}' \)
logic: dot join

\[ p \cdot q \equiv \{ (p_1, \ldots, p_{n-1}, q_2, \ldots, q_m) \mid (p_1, \ldots, p_n) \in p \land (q_1, q_2, \ldots, q_m) \in q \land p_n = q_1 \} \]

\[ p \cdot q = \begin{cases} (a, b) \\ (a, c) \\ (b, d) \end{cases} \]

\[ q = \begin{cases} (a, d, c) \\ (b, c, c) \\ (c, c, c) \\ (b, a, d) \end{cases} \]

\[ x \cdot f = \begin{cases} (c) \\ (b, d) \\ (c, a) \\ (d, a) \end{cases} \]
logic: box operators

\[ \text{exp}[q] \equiv p.\text{exp} \quad \text{(dot join takes precedence over box join)} \]

\[
\begin{align*}
\text{Book} & = \{(B0)\} & \text{myName} & = \{(N1)\} \\
\text{Name} & = \{(N0), (N1), (N2)\} & \text{myAddr} & = \{(A0)\} \\
\text{Addr} & = \{(A0), (A1), (A2)\} \\
\text{Host} & = \{(H0), (H1)\} \\
\end{align*}
\]

\[
\begin{align*}
\text{address} & = \{(B0, N0, A0), (B0, N1, A0), (B0, N2, A2)\} \\
\text{host} & = \{(A0, H0), (A1, H1), (A2, H1)\}
\end{align*}
\]

\[
\begin{align*}
\text{Book.address} \\
\text{Book.address}[\text{myName}] \\
\text{Book.address.myName}
\end{align*}
\]

\[
\begin{align*}
\text{host[myAddr]} \\
\text{address.host}
\end{align*}
\]
logic: unary operators

\[ \sim p \equiv \{ (p_n, \ldots, p_1) \mid (p_1, \ldots, p_n) \in p \} \]
\[ \hat{p} \equiv p + p.p + p.pp + \ldots \]
\[ *p \equiv \hat{p} + \text{idem} \]

Node = \{ (N0), (N1), (N2), (N3) \}
next = \{ (N0, N1), (N1, N2), (N2, N3) \}
first = \{ (N0) \}
rest = \{ (N1), (N2), (N3) \}

\sim\text{next}
\hat{\text{next}}

*\text{next}

first.\hat{\text{next}}
first.*\text{next}
logic: restriction and override

\[ q <: p \equiv \{ (p_1, \ldots, p_n) \mid (p_1) \in q \} \]  \( \text{domain restrict.} \)

\[ p :> q \equiv \{ (p_1, \ldots, p_n) \mid (p_n) \in q \} \]  \( \text{range restrict.} \)

\[ p ++ q \equiv p - (\text{domain}[q] <: p) + q \]  \( \text{override} \)

Name = \{(N0), (N1), (N2)\}  \quad \text{Alias} = \{(N0), (N1)\}
Addr = \{(A0)\}
address = \{(N0, N1), (N1, N2), (N2, A0)\}

address :> Addr
Alias <: address
address :> Alias

workAddress = \{(N0, N1), (N1, A0)\}
address ++ workAddress

\[ m' = m ++ (k \rightarrow v) \]  \( \text{update map } m \text{ with key-value pair } (k, v) \)
logic: boolean operators

| !   | & &  | ||  | =>  | <=> |
|-----|------|-----|-----|-----|
| not | and  | or  | implies | iff |
| negation | conjunction | disjunction | implication | bi-implication |

\[ F \implies G \text{ else } H \equiv (F \&\& G) \lor (!F \&\& H) \]
logic: quantifiers

<table>
<thead>
<tr>
<th>Quantifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>all x: e</td>
<td>F</td>
</tr>
<tr>
<td>some x: e</td>
<td>F</td>
</tr>
<tr>
<td>no x: e</td>
<td>F</td>
</tr>
<tr>
<td>lone x: e</td>
<td>F</td>
</tr>
<tr>
<td>one x: e</td>
<td>F</td>
</tr>
<tr>
<td>all disj x, y: e</td>
<td>F</td>
</tr>
</tbody>
</table>

some n: Name, a: Address | a in n.address
some name maps to some address — address book not empty

no n: Name | n in n.^address
no name can be reached by lookups from itself — address book acyclic

all n: Name | lone a: Address | a in n.address
every name maps to at most one address — address book is functional

all n: Name | no disj a,a': Address | (a + a') in n.address
no name maps to two or more distinct addresses — same as above
logic: set declarations

\[
\begin{array}{lll}
x & : & me \\
Q \ x & : & me \\
set & & any size \\
one & & size one (the default) \\
lone & & size 0 or 1 \\
some & & size 1 or more \\
\end{array}
\]

RecentlyUsed: set Name
RecentlyUsed is a subset of the set Name

senderAddress: Addr
senderAddress is a singleton subset of Addr

senderName: lone Name
senderName is either empty or a singleton subset of Name

receiverAddresses: some Addr
receiverAddresses is a nonempty subset of Addr
logic: relation declarations

- \( r: A \rightarrow n B \) set any number (the default)
- \( Q r: A \rightarrow n B \) one exactly one
- \( \text{lone} \) one or zero
- \( \text{some} \) one or more

workAddress: Name \( \rightarrow \) lone Addr
- each name refers to at most one work address

homeAddress: Name \( \rightarrow \) one Addr
- each name refers to exactly one home address

members: Name lone \( \rightarrow \) some Addr
- an address belongs to at most one group name and a group contains at least one address
logic: comprehensions

\{x_1: e_1, x_2: e_2, \ldots, x_n: e_n \mid F\}

\{n: \text{Name} \mid \text{no } n.^\text{address} \& \text{Addr}\}
set of names that don't resolve to any actual addresses

\{n: \text{Name}, a: \text{Address} \mid n \rightarrow a \text{ in } ^\text{address}\}
binary relation mapping names to reachable addresses
logic: if and let

three equivalent constraints:

all n: Name |
  (some n.workAddress
     implies n.address = n.workAddress
     else n.address = n.homeAddress)

all n: Name |
  let w = n.workAddress, a = n.address |
    (some w implies a = w else a = n.homeAddress)

all n: Name |
  n.address = (let w = n.workAddress |
    (some w implies w else n.homeAddress))
logic: cardinalities

#r number of tuples in r
0,1,... integer literal
+ plus
- minus

\[ \text{sum } x: e \mid e_i \]
sum of \( e_i \) for all singletons
\( x \) drawn from \( e \)

= equals
\(<\) less than
\(\geq\) greater than or equal

all \( b: \text{Bag} \mid #b.\text{marbles} \leq 3 \)
all bags have 3 or less marbles

\[ \#\text{Marble} = \text{sum } b: \text{Bag} \mid #b.\text{marbles} \]
the sum of the marbles across all bags equals the total number of marbles