CIS 771: Software Specifications

Lecture 3: Introduction to Alloy

Copyright 2001-2002, Matt Dwyer, John Hatcliff, and Rod Howell. The syllabus and all lectures for this course are copyrighted materials and may not be used in other course settings outside of Kansas State University in their current form or modified form without the express written permission of one of the copyright holders. During this course, students are prohibited from selling notes to or being paid for taking notes by any person or commercial firm without the express written permission of one of the copyright holders.

Outline

- Introduction to basic Alloy constructs using a simple example of a static model
  - How to define domains, subsets, relations with multiplicity constraints
  - How to use Alloy’s quantifiers and predicate forms
- Basic use of the Alloy Constraint Analyzer (ACA)
  - Loading, compiling, and analyzing a simple Alloy specification
  - Adjusting basic tool parameters
  - Using the visualization tool to view instances of models
Next Lecture

- Alloy: Rationale and Use Strategies
  - What types of systems have been modeled with Alloy
  - What types of questions can ACA answer
  - What is the purpose of each of the sections in an Alloy specification
- Alloy Specifications
  - Parameterized conditionals
  - Indexed relations
  - Graphical representations of Alloy models
  - More complex examples

Example: Family Structure

- We want to...
  - Model parent/child relationships as primitive relations
  - Model spousal relationships as primitive relations
  - Model relationships such as “sibling” as derived relations
  - Enforce certain biological constraints via 1st-order predicates
    (e.g., only one mother)
  - Enforce certain social constraints via 1st-order predicates
    (e.g., a wife isn’t a sibling)
  - Confirm or refute the existence of certain derived relationships
    (e.g., no one has a wife with whom he shares a parent)
Domains

- Domains are basic sets
  - they are disjoint
  - represent the individual objects that are modeled
- Typical Alloy modeling strategy
  - Identify basic domains, then declare other sets of interest as subsets
- Simple example with three domains:

  ```
  domain {A, B, C}
  ```

State Components: Sets

- Sets are introduced as subsets
  - defined over domains
  - either directly
    ```
    S1 : A // S1 is a subset of domain A
    ```
  - or indirectly
    ```
    S2 : S1 // S2 is a subset of S1
    ```
- Sets can be introduced in groups

  ```
  S3, S4, S5 : B // S3, S4, S5 subsets of B
  ```
**State Components: Sets**

- Groups of sets can have additional constraints
  - Using the **disjoint** keyword
    
    ```
    disjoint S3, S4, S5 : B
    ```
    ...guarantees that $S_3$, $S_4$, $S_5$ are disjoint

  - Using the **partition** keyword
    
    ```
    partition S3, S4, S5 : B
    ```
    ...guarantees that $S_3$, $S_4$, $S_5$ partition $B$. That is, $S_3$, $S_4$, $S_5$ are disjoint and they cover $B$ --- every element from $B$ must appear in either $S_3$, $S_4$, or $S_5$.

**Example: Family Structure**

**Alloy Model**

```alloy
model Family {
  domain {Person}
  state {
    partition Man, Woman : Person
    Married : Person
  }
}
```

**Graphical Representation**

![Family Structure Diagram]
Model Instances

The Alloy Constraint Analyzer will generate instances of models so that we can see if they match our intentions. Which of the following are instances of our current model?

```
model Family {
    domain {Person}
    state {
        partition
        Man, Woman : Person
        Married : Person
    }
}
```

A. Domains: Person = {P0,P1,P2}, Sets: Man = {P1,P2}, Married = {}, Woman = {P0}

B. Domains: Person = {P0,P1,P2}, Sets: Man = {P1,P2}, Married = {}, Woman = {P0,P1}

C. Domains: Person = {P0,P1,P2,P3}, Sets: Man = {P0,P1,P2,P3}, Married = {P2,P3}, Woman = {}

D. Domains: Person = {P0,P1}, Sets: Man = {P0}, Married = {P1}, Woman = {}

E. Domains: Person = {P0,P1}, Sets: Man = {P0}, Married = {P1,P0}, Woman = {P1}

---

State Components: Relations

- Declaring a relation between two sets A and B
  
  \[ r : A \rightarrow B \]
  
  \[ r_1, r_2 : A \rightarrow B \]

- Comments on notation...
  
  - **denotes** \( r \) is a subset of \( A \times B \)
  
  - Written with \( \rightarrow \) because we will often think of \( r \) as a “mapping” from \( A \) to \( B \)... but \( r \) is not necessarily a function
Example: Family Structure

Alloy Model with siblings

model Family {
    domain {Person}
    state {
        // Sets
        partition
        Man, Woman : Person
        Married : Person
        // Relations
        siblings : Person -> Person
    }
}

Example instance

Domains:
Person = {P0,P1,P2,P3}
Sets:
Man = {P1,P2}
Married = {}
Woman = {P0,P3}
Relations:
siblings = {P0 -> {P1,P2},
P1 -> {P0,P2},
P2 -> {P0,P1}}

(P0,P1)
(P0,P2)
(P1,P0)
(P1,P2)
(P2,P0)
(P2,P1)

Intuition: P0,P1,P2 are siblings

Relation Operators

- \(\sim\): transpose of relation
  - runs the relational mapping backward (image to domain)
  - we can introduce a named transpose at declaration
    \(R (\sim T) : A \rightarrow B\)

- What’s a good use of \(\sim\) in our Family Structure example?

  children (\sim parents) : Person \rightarrow Person
Multiplicities

- Allow us to constrain the sizes of sets, including the definition domain and the image of a relation.

- There are three multiplicities:
  - `+` : one or more
  - `?` : zero or one
  - `!` : exactly one

- Examples:
  - `red : Color!` // set red contains exactly one color
  - `favorite : Person?` // at most one favorite person

Multiplicities and Relations

- Multiplicities can be applied to the domain, range or both of a relation.

- `f : S -> T?`:
  - says that, for each element `s` of `S`, `f` maps `s` to at most a single value in `T`

- Potential instances:

```plaintext
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Conventional name:** partial function
Multiplicities and Relations

Multiplicities can be applied to the domain, range or both of a relation.

- \( f : S \rightarrow T! \)
  - says that, for each element \( s \) of \( S \), \( f \) maps \( s \) to exactly one value in \( T \)

Potential instances:

- Conventional name: total function

\[
\begin{array}{cccc}
  s1 & t1 & x & x \\
  s2 & t2 & x & x \\
  s3 & t3 & & \\
  s4 & t4 & & \\
\end{array}
\]

- Conventional name: bijection

\[
\begin{array}{cccc}
  x & x & x & x \\
  s1 & t1 & t1 & t1 \\
  s2 & t2 & t2 & t2 \\
  s3 & t3 & t3 & t3 \\
  s4 & t4 & t4 & t4 \\
\end{array}
\]
Multiplicities and Relations

Multiplicities can be applied to the domain, range or both of a relation.

\[ f : S? \rightarrow T? \]
- says that, for each element \( t \) of \( T \), at most one element of \( S \) is mapped to \( t \) by \( f \) (plus earlier constraints)

Potential instances:

\[
\begin{align*}
& s_1 \rightarrow t_1 \\
& s_2 \rightarrow t_2 \\
& s_3 \rightarrow t_3 \\
& s_4 \rightarrow t_4 \\
\end{align*}
\]

\[
\begin{align*}
& s_2 \rightarrow t_2 \\
& s_3 \rightarrow t_3 \\
& s_4 \rightarrow t_4 \\
\end{align*}
\]

\[
\begin{align*}
& s_1 \rightarrow t_1 \\
& s_2 \rightarrow t_2 \\
& s_3 \rightarrow t_3 \\
& s_4 \rightarrow t_4 \\
\end{align*}
\]

Conventional name: partial injective function

Other kinds of relational structures can be specified using multiplicities

Examples:
- \( f : S? \rightarrow T \) .partial injective relation
- \( f : S \rightarrow T+ \) .total relation
- \( f : S+ \rightarrow T \) .surjective relation
Example: Family Structure

- How would you use multiplicities to define the wife relation?

\[ \text{wife (} \sim\text{husband)} \quad : \quad \text{Man } ? \rightarrow \text{Woman } ? \]

- Intuition: partial injective function
  - Each man has zero or one woman as a wife
  - Each wife has zero or one man as a husband

Summarizing

Alloy Model

```alloy
def model Family {
    domain {Person}
    state {
        // Sets
        partition Man, Woman : Person
        Married : Person

        // Relations
        siblings : Person -> Person
        children (~parents) : Person -> Person
        wife (~husband) : Man ? -> Woman ?
    }
}
```

CIS 771 --- Intro to Alloy
Demo: Alloy Constraint Analyzer

- Load, compile, analyze cycle
- Selecting schema
- Visualization of model instances
- Increase/decrease scopes

For you to do (pause here)

- Now you should startup Alloy
- Load family-1.all
- Compile it
- Analyze the state schema
- Look at the generated instance
- Does it look correct?
- What if anything would you change about it?
Model Instance

Instance found:
Domains:
    Person = \{P0, P1, P2\}
Sets:
    Man = \{P1, P2\}
    Married = \{P0, P1, P2\}
    Woman = \{P0\}
Relations:
    children = \{P0 \rightarrow \{P0, P2\}, P1 \rightarrow \{P0, P1, P2\}, P2 \rightarrow \{P2\}\}
    husband = \{P0 \rightarrow \{P2\}\}
    parents = \{P0 \rightarrow \{P0, P1\}, P1 \rightarrow \{P1\}, P2 \rightarrow \{P0, P1, P2\}\}
    siblings = \{P0 \rightarrow \{P0, P2\}, P1 \rightarrow \{P2\}, P2 \rightarrow \{P0, P1, P2\}\}
    wife = \{P2 \rightarrow \{P0\}\}

Person can be their own parent?

Instance found:
Domains:
    Person = \{P0, P1, P2\}
Sets:
    Man = \{P1, P2\}
    Married = \{P0, P1, P2\}
    Woman = \{P0\}
Relations:
    children = \{P0 \rightarrow \{P0, P2\}, P1 \rightarrow \{P0, P1, P2\}, P2 \rightarrow \{P2\}\}
    husband = \{P0 \rightarrow \{P2\}\}
    parents = \{P0 \rightarrow \{P0, P1\}, P1 \rightarrow \{P1\}, P2 \rightarrow \{P0, P1, P2\}\}
    siblings = \{P0 \rightarrow \{P0, P2\}, P1 \rightarrow \{P2\}, P2 \rightarrow \{P0, P1, P2\}\}
    wife = \{P2 \rightarrow \{P0\}\}
Multiple Fathers?

Instance found:
Domains:
  Person = {P0,P1,P2}
Sets:
  Man = {P1,P2}
  Married = {P0,P1,P2}
  Woman = {P0}
Relations:
  children = {P0 -> {P0,P2}, P1 -> {P0,P1,P2}, P2 -> {P2}}
  husband = {P0 -> {P2}}
  parents = {P0 -> {P0,P1}, P1 -> {P1}, P2 -> {P0,P1,P2}}
  siblings = {P0 -> {P0,P2}, P1 -> {P2}, P2 -> {P0,P1,P2}}
  wife = {P2 -> {P0}}

Self-Siblings, Child-Siblings?

Instance found:
Domains:
  Person = {P0,P1,P2}
Sets:
  Man = {P1,P2}
  Married = {P0,P1,P2}
  Woman = {P0}
Relations:
  children = {P0 -> {P0,P2}, P1 -> {P0,P1,P2}, P2 -> {P2}}
  husband = {P0 -> {P2}}
  parents = {P0 -> {P0,P1}, P1 -> {P1}, P2 -> {P0,P1,P2}}
  siblings = {P0 -> {P0,P2}, P1 -> {P2}, P2 -> {P0,P1,P2}}
  wife = {P2 -> {P0}}
Married w/out husband/wife?

Instance found:
Domains:
  Person = {P0,P1,P2}
Sets:
  Man = {P1,P2}
  Married = {P0,P1,P2}
  Woman = {P0}
Relations:
  children = {P0 -> {P0,P2}, P1 -> {P0,P1,P2}, P2 -> {P2}}
  husband = {P0 -> {P2}}
  parents = {P0 -> {P0,P1}, P1 -> {P1}, P2 -> {P0,P1,P2}}
  siblings = {P0 -> {P0,P2}, P1 -> {P2}, P2 -> {P0,P1,P2}}
  wife = {P2 -> {P0}}
Adding Constraints

- We’d like to enforce the following constraints which are simply matters of biology
  - No person can be their own parent (or more generally, their own ancestor)
  - No person can have more than one father or mother
  - A person’s siblings are those with the same parents

Adding Constraints

- We’d like to enforce the following social constraints
  - Any married man has a wife
  - Any married woman has a husband
  - A man’s wife cannot be one of his siblings
Quantifiers

- Alloy includes a rich collection of quantifiers
  - all x : s | F  F holds for every x in s
  - some x : s | F  F holds for some x in s
  - no x : s | F    F fails for every x in s
  - sole x : s | F  F holds for at most 1 x in s
  - one x : s | F   F holds for exactly 1 x in s

Everything is a Set in Alloy

- There are no scalars
  - We never speak directly about elements of sets
  - Instead, we always use singleton sets, e.g.,
    matt : Person!
- When we have quantification, e.g.,
  all x : s | ..x...
  x = {e} for some element e of s
### Set Comparison

- **s = t**: s and t have same elements
- **s in t**: s is a subset of t
- **s != t**: negation of equality
- **s !in t**: negation of subset
- **s /= t**: s = t and s is non-empty
- **s /in t**: s in t and s is non-empty

Also,
- **not** can be used for a unary negation operator

### Navigation

**s. R**

Yields the set of elements from B reachable from all elements of s : A navigating through R : A -> B

![Diagram](diagram.png)

- **Examples**
  - `matt.parents` // Matt’s parents
  - `matt.parents.parents` // Matt’s grandparents

- What if we want to find Matt’s ancestors or descendents?
### Transitive Closure

+\[ r \]

- Intuitively, the transitive closure of a relation \( r: S \times S \) is what you get when you keep navigating through \( r \) until you can't go any farther.

\[
(S0,S1) \quad (S1,S2) \quad (S2,S3) \quad (S4,S7) \quad +r
\]

\[
(S0,S1) \quad (S1,S2) \quad (S2,S3) \quad (S4,S7) \quad (S0,S2) \quad (S0,S3) \quad (S1,S3) \quad +r
\]

- What if we want to find Matt's ancestors or descendents?

  - \texttt{matt.+parents} // Matt's ancestors
  - \texttt{matt.+(~parents)} // Matt's descendents

### Example: Family Structure

- How would you express the constraint “No person can be their own ancestor”

  \[
  \text{no } p : \text{Person} \mid p \in p.+parents
  \]
For you to do (pause here)

- Do we need to add the constraint “No person can be their own descendent”?

Set Operators

- Set operators
  - + : union
  - & : intersection
  - - : difference

- Build the set containing Matt’s father
  
  \[ \text{matt.parents} \& \text{Man} \]
Set Predicates

- **some s**: s is non-empty (#s > 0)
- **no s**: s is empty (#s = 0)
- **sole s**: s has at most one element (#s <= 1)
- **one s**: s has exactly one element (#s = 1)
- There is no general way to find out the size of a set (the # operator is not available)
- There are no set constants (e.g., empty set {})
Example: Family Structure

- How would you express the constraint “No person can have more than one father or mother”

\[
\text{all } p : \text{Persons} \mid (\text{sole}(p.\text{parents} \land \text{Man})) \land\ (\text{sole}(p.\text{parents} \land \text{Woman}))
\]

- This is an example of a negative constraint that is easier to state positively (to make use of the `sole` operator).

Set Comprehension

\[
\{ x : S \mid F \}
\]

- the set of values drawn from set $S$ for which $F$ holds

- How would use the comprehension notation to specify the set of people that have the same parents as Matt?

\[
\{ q : \text{People} \mid q.\text{parents} = \text{matt.parents} \}
\]
Example: Family Structure

- How would you express the constraint “A person P’s siblings are those people with the same parents as P (excluding P)”

\[
\text{all } p \mid \text{p.siblings} = (\{ q \mid \text{p.parents} = q.\text{parents} \} - p)
\]

- Note: you can omit type declarations in quantification and comprehensions because they can be inferred from the context.

---

Example: Family Structure

- Each married man has a wife and everyone with a wife is a married man

\[
\text{all } p \mid \text{some } p.\text{wife} \leftrightarrow p \in \text{Man & Married}
\]

- A wife can’t be a sibling

\[
\text{no } p \mid p.\text{wife} /\in p.\text{siblings}
\]

- Why /\in instead of \in?

---
For you to do (pause here)

- Load family-2.all
- Compile it
- Analyze the state schema
- Look at the generated instance
- Does it look correct?
- What if anything would you change about it?

Empty Instances

- The analyzer’s algorithms prefer smaller instances
  - Often it produces empty or otherwise trivial instances
  - It is useful to know that these instances satisfy the constraints (since you may not want them)
- Usually, they do not illustrate the interesting behaviors that are possible
Conditions

- We can use condition schemas to encode “realism constraints” to e.g.,
  - Force generated models to include at least one married person, or one married man, etc.
- Later on we’ll see that condition schemas can be used to implement “constraint macros” – parameterized macros that can be called from other schemas.
  - This allows common constraints to be shared.

For you to do (pause here)

- Load family-3.all
- Compile it
- Analyze the state schema
- Look at the generated instance
- Does it look correct?
- How can you produce two married couples?
  - Modify conditions or scopes?
Assertions

- Often we believe that our model enforces certain constraints that are not directly expressed.
- We can express these additional constraints as assertions and use the analyzer to check if they hold.
- If an assertion does not hold, the analyzer will produce a counterexample instance.
- If a desired property expressed as an assertion does not hold, typically you want to move that constraint into an invariant or otherwise refine your specification until the assertion holds.

No person has a parent that’s also a sibling.

\[ \forall p \mid \neg p.\text{parents} \land p.\text{siblings} \]

Every person’s siblings are his/her siblings’ siblings.

\[ \forall p \mid p.\text{siblings} = p.\text{siblings}.\text{siblings} \]

No person shares a common ancestor with his wife (i.e., wife isn’t related by blood).

\[ \neg p \mid \exists (p.\text{parents} \land p.\text{wife}.\text{parents}) \]
For you to do (pause here)

- Load family-4.all
- Compile it
- Analyze the state schema
- Look at the generated counter-examples
- Why is SiblingsSiblings false?
- Why is NoIncest false?

Problems with Assertions

Analyzing SiblingsSiblings ...
Scopes: Person(3)
Counterexample found:
Domains:
  Person = \{P0,P1,P2\}
Sets:
  Man = \{P1,P2\}
  Married = \{P0,P1\}
  Single = \{P2\}
  Woman = \{P0\}
Relations:
  children = \{P1 -> \{P0,P2\}\}
  husband = \{P0 -> \{P1\}\}
  parents = \{P0 -> \{P1\}, P2 -> \{P1\}\}
  siblings = \{P0 -> \{P2\}, P2 -> \{P0\}\}
  wife = \{P1 -> \{P0\}\}

P0.siblings = \{P2\}
P0.siblings.siblings = \{P0\}
**Problems with Assertions**

Analyzing NoIncest ...
Scopes: Person(3)
Counterexample found:
Domains:
   Person = \{P1,P2\}
Sets:
   Man = \{P2\}
   Married = \{P1,P2\}
   Single = {}  
   Woman = \{P1\}
Relations:
   children = \{P2 -> \{P1\}\}
   **husband** = \{P1 -> \{P2\}\}
   **parents** = \{P1 -> \{P2\}\}
   siblings = {}
   wife = \{P2 -> \{P1\}\}

**For you to do**

- Fix the specification
  - If the model is underconstrained, add appropriate constraints
  - If the assertion is not correct, modify it
- Demonstrate that your fixes yield no counter-examples
  - Does varying the scope make a difference?
  - Does this mean that the assertions hold for all models?
### For you to do

- Express the notion of “blood relative” (share common ancestor) as a condition parameterized on two singleton sets p and q that holds when p and q have a common ancestor.
- Add an extra group of invariants that add common social constraints on the husband/wife and parent relations
  - A person can’t have children with a blood relative
  - A person can’t be married to a blood relative.

### Acknowledgements

- Portions of these slides are adapted from a previous Alloy introductory lecture developed by Matt Dwyer.
- The family structure example is based on an example from Daniel Jackson distributed with the Alloy Constraint Analyzer.