CS 6610: Software Engineering II

Lecture 5: Introduction to Alloy - II
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 (\(#s \leq 1\) )
- **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 \( \{\} \) )
Alloy includes a rich collection of quantifiers:

- **all** $x : s \mid F$  
  F holds for every $x$ in $s$
- **some** $x : s \mid F$  
  F holds for some $x$ in $s$
- **no** $x : s \mid F$  
  F fails for every $x$ in $s$
- **lone** $x : s \mid F$  
  F holds for at most 1 $x$ in $s$
- **one** $x : s \mid F$  
  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.,
    \[ \text{matt} : \text{(one) Person} \]

- When we have quantification, e.g.,
  \[ \text{all } x : s \mid \ldots x \ldots \]
  \[ x = \{e\} \text{ for some element } e \text{ of } s \]
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

Also,
- `not` can be used for a unary negation operator
Yields the set of elements from $B$ reachable from all elements of $s : A$ navigating through $R : A \rightarrow B$. 

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

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.

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

- $\text{matt.}^{\text{parents}}$ // Matt’s ancestors
- $\text{matt.}^{\text{(~parents)}}$ // Matt’s descendents
How would you express the constraint “No person can be their own ancestor”
Example: Family Structure

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

```plaintext
no p : Person | p in p.+parents
```
For you to do

- 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
  
  \[ \text{matt.parents} \& \text{Man} \]
Logical Operators

- The usual operators are available
  - F && G : conjunction
  - F || G : disjunction
  - F=>G : implication
  - F<=>G : bi-directional implication
  - !F : negation
How would you express the constraint “No person can have more than one father or mother”

\[ \text{all } p : \text{Persons} | (\text{lone } (p.\text{parents } & \text{Man})) \land (\text{lone } (p.\text{parents } & \text{Woman})) \]

- This is an example of a negative constraint that is easier to state positively (to make use of the \text{lone} 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?

\[ \{ q : \text{People} \mid q.\text{parents} = \text{matt.\text{parents}} \} \]
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)”

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

- 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
  
  \[ \text{all } p \mid \text{some } p.\text{wife} \iff p \text{ in Man} \land \text{Married} \]

- A wife can’t be a sibling
  
  \[ \text{no } p \mid p.\text{wife} \land p.\text{siblings} \]
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
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.
Assertions

- No person has a parent that’s also a sibling.
  \[
  \text{all } p \mid \text{no } p.\text{parents} \land p.\text{siblings}
  \]

- Every person’s siblings are his/her siblings’ siblings.
  \[
  \text{all } p \mid p.\text{siblings} = p.\text{siblings}.\text{siblings}
  \]

- No person shares a common ancestor with his wife (i.e., wife isn’t related by blood).
  \[
  \text{no } p \mid \text{some } (p.\text{parents} \land p.\text{wife}.\text{parents})
  \]
Problems with Assertions

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