22c:181 / 55:181
Formal Methods in Software Engineering

Introduction to Alloy
Outline

- Introduction to basic Alloy constructs using a simple example of a static model
  - How to define sets, subsets, relations with multiplicity constraints
  - How to use Alloy’s quantifiers and predicate forms
- Basic use of the Alloy Analyzer 4 (AA)
  - Loading, compiling, and analyzing a simple Alloy specification
  - Adjusting basic tool parameters
  - Using the visualization tool to view instances of models
Roadmap

- Alloy: Rationale and Use Strategies
  - What types of systems have been modeled with Alloy
  - What types of questions can AA 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
Alloy --- Why was it created?

- **Lightweight**
  - small and easy to use, and capable of expressing common properties tersely and naturally

- **Precise**
  - having a simple and uniform mathematical semantics

- **Tractable**
  - amenable to efficient and fully automated semantic analysis (within scope limits)
Alloy --- Comparison

- **UML**
  - Has similarities (graphical notation, OCL constraints) but it is neither lightweight, nor precise
  - UML includes many modeling notions omitted from Alloy (use-cases, state-charts, code architecture specs)
  - Alloy’s diagrams and relational navigation are inspired by UML

- **Z**
  - Precise, but intractable. Stylized typography makes it harder to work with.
  - Z is more expressive than Alloy, but more complicated
  - Alloy’s set-based semantics is inspired by Z
Alloy --- What is it used for?

- Alloy is not meant for modeling code architecture (ala class diagrams in UML)

- But there might be a close relationship between the Alloy specification and an implementation in an OO language
Alloy --- Example Applications

- The Alloy 4 distribution comes with over dozens of example models that together illustrate the use of Alloy’s constructs.

  Examples

  - Specification of a distributed spanning tree
  - Model of a generic file system
  - Model of a generic file synchronizer
  - Tower of Hanoi model
Example: Family Structure

We want to...

- Model parent/child relationships as primitive relations
- Model spousal relationships as primitive relations
- Model relationships such as “siblings” 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)
Example: addressBook

- An **address book** for an email client that maintains a mapping from **names** to **addresses**.

<table>
<thead>
<tr>
<th>FriendBook</th>
<th>WorkBook</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ted -&gt; <a href="mailto:ted@gmail.com">ted@gmail.com</a></td>
<td>Mrs Pilard -&gt; <a href="mailto:lpilard@uiowa.edu">lpilard@uiowa.edu</a></td>
</tr>
<tr>
<td>Ryan -&gt; <a href="mailto:ryan@hotmail.com">ryan@hotmail.com</a></td>
<td>Ryan -&gt; <a href="mailto:ryan@uiowa.edu">ryan@uiowa.edu</a></td>
</tr>
</tbody>
</table>
Atoms and Relations

- In Alloy, everything is built from atoms and relations.

- An atom is a primitive entity that is:
  - *Indivisible*: it can’t be broken down into smaller parts
  - *Immutable*: its properties don’t change over time
  - *Uninterpreted*: it does not have any built in property (the way numbers do for example)

- A relation is a structure that relates atoms. It is a set of tuples, each tuple being a sequence of atoms.
Atoms and Relations: Examples

- **Unary relations**: a set of names, a set of addresses and a set of books
  
  \[
  \begin{align*}
  \text{Name} &= \{(N0),(N1),(N2)\} \\
  \text{Addr} &= \{(D0),(D1)\} \\
  \text{Book} &= \{(B0),(B1)\}
  \end{align*}
  \]

- **A binary relation** from names to addresses
  
  \[
  \text{address} = \{(N0,D0),(N1,D1)\}
  \]

- **A ternary relation** from books to name to addresses
  
  \[
  \text{addr} = \{(B0,N0,D0), (B0,N1,D1), (B1,N1,D2)\}
  \]
Relations

- **Size of a relation**: the number of tuples in the relation
- **Arity of a relation**: the number of atoms in each tuple of the relation
  - relations with arity 1, 2, and 3 are said to be *unary*, *binary*, and *ternary* relations
- **Examples**:
  - relation of arity 1 and size 1: myName = {(N0)}
  - relation of arity 2 and size 3:
    - address = {(N0,D0),(N1,D1),(N2,D1)}
Alloy Specifications

- Signatures and Fields
- Predicates and Functions
- Facts
- Assertions
- Commands and scopes
Signatures and Fields

- **Signatures**
  - Describes the entities that you want to reason about
  - Sets defined in signatures are fixed (concept related to operations and dynamic models)

- **Fields**
  - Define relations between signatures

- **Simple constraints**
  - Multiplicities on signatures
  - Multiplicities on relations
Signatures

- A signature introduces a set of atoms

- The declaration
  
  ```
  sig A {}
  ```

  introduces a set named A.

- A set can be introduced as an extension of another; thus
  
  ```
  sig A1 extends A {}
  ```

  introduces a set A1 that is a subset of A
Example Signatures and Fields

Family Structure:

abstract sig Person {
  children: set Person,
  siblings: set Person
}

sig Man, Woman extends Person {}

sig Married in Person {
  marriedWith: one Married
}
Signatures

- A1 and A2 are extensions of A
- A signature that is declared independently of any other is a top-level signature, e.g. A and B.

- The extension of a signature are mutually disjoint, as are top-level signatures.

\begin{verbatim}
  sig A {}
  sig B {}
  sig A1 extends A {}
  sig A2 extends A {}
\end{verbatim}
Signatures

- An abstract signature has no element except those belonging to its extensions.
- A set can be introduced as a subset of another.

```
abstract sig A {}
sig B {}
sig A1 extends A {}
sig A2 extends A {}
```

```
sig A3 in A {}
```
Example: Family Structure

**Alloy Model**

abstract sig Person {}

sig Man extends Person {}

sig Woman extends Person {}

sig Married in Person {}

**Graphical Representation**

- Person
  - Man
  - Woman
  - Married

- Married
  - in
  - extends
  - extends
The Alloy 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 Instances

**abstract sig Person {}**

**sig Man extends Person {}**

**sig Woman extends Person {}**

**sig Married in Person {}**

- Person = {(P0),(P1),(P2)}
  - Man = {(P1),(P2)}
  - Married = {}
  - Woman = {(P0),(P1)}

- Person = {(P0),(P1)}
  - Man = {(P0)}
  - Married = {(P1)}
  - Woman = {}

- Person = {(P0),(P1),(P2),(P3)}
  - Man = {(P0),(P1),(P2),(P3)}
  - Married = {(P2),(P3)}
  - Woman = {}

- Person = {(P0),(P1)}
  - Man = {(P0)}
  - Married = {(P1),(P0)}
  - Woman = {(P1)}

- Person = {(P0),(P1),(P2)}
  - Man = {(P1),(P2)}
  - Married = {}
  - Woman = {(P0),(P1)}
Fields

- **Relations** are declared as **fields of signatures**
  - Writing
    
    \[ \text{sig } A \{ f: e \} \]
  
    introduces a relation \( f \) whose **domain is** \( A \) and whose image is given by the expression \( e \).

- **Examples:**
  - **Binary Relation:**
    \[ \text{sig } A \{ f1: A \} \]  
    // \( f1 \) is a subset of \( A \times A \)
  
  - **Ternary Relation:**
    \[ \text{sig } B \{ f2: A \rightarrow A \} \]  
    // \( f2 \) is a subset of \( B \times A \times A \)
**Example: Family Structure**

**Alloy Model with siblings**

```alloy
abstract sig Person {
    siblings: Person
}
sig Man extends Person {}
sig Woman extends Person {}
sig Married in Person {}
```

**Example instance**

- Person = {((P0), (P1))}
- Man = {((P0), (P1))}
- Married = {}
- Woman = {}
- siblings = {((P0,P1), (P1,P0))}

**Intuition:** P0 and P1 are siblings

`siblings is a binary relation it is a subset of Person x Person`
Multiplicities

- Allow us to constrain the sizes of sets
  - A multiplicity keyword placed before a signature declaration constraints the number of element in the signature’s set
    \[
    \text{m \ sig \ A \ {}}
    \]
  - We can also make multiplicities constraints on fields:
    \[
    \text{sig \ A \ \{f: \ m \ e\}}
    \]
    \[
    \text{sig \ A \ \{f: \ e1 \ m -> n \ e2\}}
    \]

- There are four multiplicities
  - \text{set} : any number
  - \text{some} : one or more
  - \text{lone} : zero or one
  - \text{one} : exactly one
Multiplicities: examples

- **Without multiplicity:**
  - A set of pixels, each of which is red, green or blue

  ```
  abstract sig Pixel {} 
  sig Red, Yellow, Green extends Pixel {} 
  ```

- **With multiplicity:**
  - An enumeration of color

  ```
  abstract sig Color {} 
  one sig Red, Yellow, Green extends Color {} 
  ```
Multiplicities: examples

A file system in which each directory contains any number of objects, and each alias points to exactly one object.

```
abstract sig Object {}
sig Directory extends Object {contents: set Object}
sig File extends Object {}
sig Alias extends File {to: one Object}
```

- The default keyword, if omitted, is `one`, so:
  
  ```
sig A {f: e} and sig A {f: one e}
  ```

are equivalent.
Multiplicities: examples

- A book maps names to addresses
  - There is at most one address per Name
  - An address is associated to at least one name

```
sig Name, Addr {}
sig Book {
    addr: Name some -> lone Addr
}
```
Multiplicities: examples

- A collection of weather forecasts, each of which has a field weather associating every city with exactly one weather condition

```
sig Forecast {weather: City -> one Weather}
sig City {}
abstract sig Weather {}
one sig Rainy, Sunny, Cloudy extends Weather {}
```

- Instance:

  City = {(Iowa City), (Chicago)}
  Rainy = {(rainy)}
  Sunny = {(sunny)}
  Cloudy = {(cloudy)}
  Forecast = {(ch1), (ch2)}
  weather = { (ch1, Iowa City, rainy), (ch1, Chicago, rainy),
               (ch2, Iowa City, rainy), (ch2, Chicago, sunny) }
```
Multiplicities and Binary Relations

**sig S {f: lone T}**
- says that, for each element \( s \) of \( S \), \( f \) maps \( s \) to at most a single value in \( T \)

*Conventional name:* partial function

**Potential instances:**

- \( s_1 \rightarrow t_1 \)
- \( s_2 \rightarrow t_2 \)
- \( s_3 \rightarrow t_3 \)
- \( s_4 \rightarrow t_4 \)
- \( s_1 \rightarrow t_1 \)
- \( s_2 \rightarrow t_2 \)
- \( s_3 \rightarrow t_3 \)
- \( s_4 \rightarrow t_4 \)
- \( s_1 \rightarrow t_1 \)
- \( s_2 \rightarrow t_2 \)
- \( s_3 \rightarrow t_3 \)
- \( s_4 \rightarrow t_4 \)
Multiplicities and Binary Relations

- **sig S \{ f: one T \}**
  - says that, for each element \( s \) of \( S \), \( f \) maps \( s \) to **exactly one** value in \( T \)

- **Potential instances:**

Conventional name: total function
Multiplicities and Ternary Relations

- **sig S {f: T -> one V}**
  - Says that, for each element \( s \) of \( S \), the field \( f \) satisfies:
    - for each element \( t \) of \( T \), \( f \) maps \( t \) to exactly one value in \( V \)

- Potential instances:
Multiplicities and Ternary Relations

- **sig S \{ f: T \rightarrow V \}**
  - Says that, for each element s of S, the field f satisfied:
    - for each element v of V, at most one element of T is mapped to v by f.

- **Potential instances:**

```
> s1   s2
   t1  t2  v1  v2
   t3  t4  v3  v4

X s1
   t1  t2  v1  v2
   t3  t4  v3  v4

> s1
   t1  t2  v1  v2
   t3  t4  v3  v4

X s1
   t1  t2  v1  v2
   t3  t4  v3  v4
```
Multiplicities and Relations

- Other kinds of relational structures can be specified using multiplicities

- Examples:
  - `sig S {f: some T}` ...total relation
  - `sig S {f: set T}` ...partial relation
  - `sig S {f: T set -> set V}`
  - `sig S {f: T one -> V}`
  - ...

Example: Family Structure

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

  \[\text{sig Person \{children: set Person\}}\]

  - Intuition: each person has zero or more children

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

  \[\text{sig Married \{marriedWith: one Married\}}\]

  - Intuition: each married person has exactly one spouse
Summarizing

**Alloy Model**

```alloy
abstract sig Person {
    children: set Person,
    siblings: set Person
}

sig Man, Woman extends Person {}

sig Married in Person {
    marriedWith: one Married
}
```
Exercises

- Start Alloy:
- Load file family-1.als
- Execute it
- Analyze the metamodel
- Look at the generated instance
- Does it look correct?
- What, if anything, would you change about it?
Model Instance

Instance found:

Person = {Man0, Man1, Man2}
Man = {Man0, Man1, Man2}
Woman = {}
Married = {Man0, Man1, Man2}

children = { (Man0, Man0), (Man0, Man1),
            (Man1, Man0),
            (Man2, Man1), (Man2, Man2) }
siblings = { (Man0, Man0), (Man0, Man1),
            (Man1, Man0), (Man1, Man2),
            (Man2, Man2) }
marridWith = { (Man1, Man0), (Man0, Man2), (Man2, Man0) }
Person can be their own child?

Instance found:

Person = \{\text{Man0}, \text{Man1}, \text{Man2}\}
Man = \{\text{Man0}, \text{Man1}, \text{Man2}\}
Woman = \{\}
Married = \{\text{Man0}, \text{Man1}, \text{Man2}\}

children = \{(\text{Man0}, \text{Man0}), (\text{Man0}, \text{Man1}),
            (\text{Man1}, \text{Man0}),
            (\text{Man2}, \text{Man1}), (\text{Man2}, \text{Man2})\}

siblings = \{(\text{Man0}, \text{Man0}), (\text{Man0}, \text{Man1}),
            (\text{Man1}, \text{Man0}), (\text{Man1}, \text{Man2}),
            (\text{Man2}, \text{Man2})\}

marriedWith = \{(\text{Man1}, \text{Man0}), (\text{Man0}, \text{Man2}), (\text{Man2}, \text{Man0})\}
Multiple Fathers?

Instance found:

Person = \{Man0, Man1, Man2\}
Man = \{Man0, Man1, Man2\}
Woman = {} 
Married = \{Man0, Man1, Man2\}

children = \{(Man0, Man0), (\textbf{Man0}, Man1), \\
(Man1, Man0), \\
(\textbf{Man2}, Man1), (Man2, Man2)\}

siblings = \{(Man0, Man0), (Man0, Man1), \\
(Man1, Man0), (Man1, Man2), \\
(Man2, Man2)\}

marriedWith = \{(Man1, Man0), (Man0, Man2), (Man2, Man0)\}
Self-Siblings?

Instance found:

Person = \{\text{Man0, Man1, Man2}\}
Man = \{\text{Man0, Man1, Man2}\}
Woman = \{
Married = \{\text{Man0, Man1, Man2}\}

children = \{
    (\text{Man0, Man0}),
    (\text{Man0, Man1}),
    (\text{Man1, Man0}),
    (\text{Man2, Man1}),
    (\text{Man2, Man2})
\}
siblings = \{
    (\text{Man0, Man0}),
    (\text{Man0, Man1}),
    (\text{Man1, Man0}),
    (\text{Man1, Man2}),
    (\text{Man2, Man2})
\}
marrriedWith = \{(\text{Man1, Man0}), (\text{Man0, Man2}), (\text{Man2, Man0})\}
Asymmetric Siblings ?

Instance found:

Person = {Man0,Man1,Man2}
Man = {Man0,Man1,Man2}
Woman = {}
Married = {Man0,Man1,Man2}

children = { (Man0,Man0), (Man0,Man1),
            (Man1,Man0),
            (Man2,Man1), (Man2,Man2) }

siblings = { (Man0,Man0), (Man0,Man1),
             (Man1,Man0), (Man1,Man2),
             (Man2,Man1), (Man2,Man2)  "where is (Man2,Man1) ?"
             }

marriedWith = {(Man1,Man0), (Man0,Man2), (Man2,Man0)}
Instance found:

Person = \{\text{Man0}, \text{Man1}, \text{Man2}\}
Man = \{\text{Man0}, \text{Man1}, \text{Man2}\}
Woman = \{
Married = \{\text{Man0}, \text{Man1}, \text{Man2}\}

children = \{(\text{Man0, Man0}), (\text{Man0, Man1}), (\text{Man1, Man0}), (\text{Man2, Man1}), (\text{Man2, Man2})\}

siblings = \{(\text{Man0, Man0}), (\text{Man0, Man1}), (\text{Man1, Man0}), (\text{Man1, Man2}), (\text{Man2, Man2})\}

marriedWith = \{(\text{Man1, Man0}), (\text{Man0, Man2}), (\text{Man2, Man0})\}
Asymmetric marriage?

Instance found:

\[
\text{Person} = \{\text{Man0, Man1, Man2}\} \\
\text{Man} = \{\text{Man0, Man1, Man2}\} \\
\text{Woman} = \{} \\
\text{Married} = \{\text{Man0, Man1, Man2}\}
\]

\[
\text{children} = \{ (\text{Man0, Man0}), (\text{Man0, Man1}), \\
(\text{Man1, Man0}), \\
(\text{Man2, Man1}), (\text{Man2, Man2}) \}
\]

\[
\text{siblings} = \{ (\text{Man0, Man0}), (\text{Man0, Man1}), \\
(\text{Man1, Man0}), (\text{Man1, Man2}), \\
(\text{Man2, Man2}) \}
\]

\[
\text{marriedWith} = \{(\text{Man1, Man0}), (\text{Man0, Man2}), (\text{Man2, Man0})\}
\]

“where is (Man0, Man1)?”
Model Weaknesses

- The model is underconstrained
  - It doesn’t match our domain knowledge
  - We can add constraints to enrich the model
- Underconstrained models are common early in the development process
  - AA gives us quick feedback on weaknesses in our model
  - We can incrementally add constraints until we are satisfied with it
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
  - The marriedWith relation is symmetric
  - A man’s wife cannot be one of his siblings
Predefined Sets

- There are three constants:
  - **none**: empty set
  - **univ**: universal set
  - **ident**: identity

- Example. For a model with just the two sets:
  
  \[
  \begin{align*}
  \text{Man} &= \{(M0),(M1),(M2)\} \\
  \text{Woman} &= \{(W0),(W1)\}
  \end{align*}
  \]

  the constants have the values

  \[
  \begin{align*}
  \text{none} &= \{} \\
  \text{univ} &= \{(M0),(M1),(M2),(W0),(W1)\} \\
  \text{ident} &= \{(M0,M0),(M1,M1),(M2,M2),(W0,W0),(W1,W1)\}
  \end{align*}
  \]
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
  - `lone x: S | F` F holds for at most 1 x in S
  - `one x: S | F`  F holds for exactly 1 x in S
Quantifiers

- Quantifiers can be also applied to expressions denoting relations

  - `some e` - $e$ is non-empty
  - `no e` - $e$ is empty
  - `lone e` - $e$ has at most one tuple
  - `one e` - $e$ has exactly one tuple
Everything is a Set in Alloy

- There are no scalars
  - We never speak directly about elements (or tuples) of relations
  - Instead, we always use singleton relations:
    ```
    let matt = one Person
    ```

- When we have quantification:
  ```
  all x : S | ... x ... 
  x = {t} for some element t of S
  ```
Set Operators

- Set operators
  - + : union
  - & : intersection
  - - : difference
  - in : subset
  - = : equality

- Build the set containing Matt’s sons

  \[ \text{matt.children} \quad \& \quad \text{Man} \]
Relational Operators

- `->` arrow (product)
- `~` transpose
- `.` dot join
- `[]` box join
- `^` transitive closure
- `*` reflexive-transitive closure
- `<:` domain restriction
- `:>` image restriction
- `++` override
Arrow Product

- **p -> q**
  - p and q are two relations
  - p->q is the relation you get by taking every combination of a tuple from p and a tuple from q and concatenating them.

- **Examples:**
  
  Name = {((N0),(N1))}  
  Addr = {((D0),(D1))}  
  Book = {((B0))}  

  Name -> Addr = {((N0,D0),(N0,D1),(N1,D0),(N1,D1))}  
  Book -> Name -> Addr =  
  {((B0,N0,D0),(B0,N0,D1),(B0,N1,D0),(B0,N1,D1))}
**Transpose**

- \( \sim p \)
  - take the mirror image of the relation \( p \), i.e. reverse the order of atoms in each tuple.

- **Example:**
  - \( \text{example} = \{(a0, a1, a2, a3), (b0, b1, b2, b3)\} \)
  - \( \sim\text{example} = \{(a3, a2, a1, a0), (b3, b2, b1, b0)\} \)

- **How would you use \( \sim \) to express the parent relation?**
  - \( \sim\text{children} \)
How to join tuples?

- **p.q**: What is the join of these two tuples?
  - \((s_1, \ldots, s_m)\)
  - \((t_1, \ldots, t_m)\)

  If \(s_m \neq t_1\) then the result is empty.
  If \(s_m = t_1\) then the result is: \((s_1, \ldots, s_{m-1}, t_2, \ldots, t_m)\)

- **Examples**:
  - \(\{(a,b)\} . \{(a,c)\} = \{\}\)
  - \(\{(a,b)\} . \{(b,c)\} = \{(a,c)\}\)

- What about \(\{(a)\} . \{(a)\}\)? Not defined!
  - p.s is defined iff p and s are not both unary relation
Exercises

What’s the result of these join applications?

- \{(a,b)\} \cdot \{(c)\}
- \{(a)\} \cdot \{(a,b)\}
- \{(a,b)\} \cdot \{(b)\}
- \{(a)\} \cdot \{(a,b,c)\}
- \{(a,b,c)\} \cdot \{(c)\}
- \{(a,b)\} \cdot \{(a,b,c)\}
- \{(a,b,c,d)\} \cdot \{(d,e,f)\}
- \{(a)\} \cdot \{(b)\}
How to join relations?

- \( p \cdot q \)
  - \( p \) and \( q \) are two relations that are **not both unary**
  - \( p \cdot q \) is the relation you get by taking every combination of a tuple from \( p \) and a tuple from \( q \) and adding their join, if it exists.
Examples:

to maps a message to the name it’s intended to be send to

address maps names to addresses

- **to** = {\((M0, N0), (M0, N2), (M1, N2), (M2, N3)\)}
- **address** = {\((N0, D0), (N0, D1), (N1, D1), (N2, D3)\)}

**to.address** maps a message to the addresses it should be sent to

- **to.address** = {\((M0, D0), (M0, D1), (M0, D3), (M1, D3)\)}
Exercises

- Given a relation $addr$ of arity four that contains the tuple $b \rightarrow n \rightarrow a \rightarrow t$ when book $b$ maps name $n$ to address $a$ at time $t$, and a book $b$ and a time $t$:
  - $addr = \{(B0,N0,D0,T0),(B0,N0,D1,T1),\ (B0,N1,D2,T0),(B0,N1,D2,T1),(B1,N2,D3,T0),\ (B1,N2,D4,T1)\}$
  - $t = \{(T1)\}$
  - $b = \{(B0)\}$

  The expression $b \cdot addr \cdot t$ is the name-address mapping of book $b$ at time $t$. What is the value of $b \cdot addr \cdot t$?

- When $p$ is a binary relation and $q$ is a ternary relation, what is the arity of the relation $p \cdot q$?

- Join is not associative, why?
  (i.e. $(a \cdot b) \cdot c$ and $a \cdot (b \cdot c)$ are not always equivalent)
Example: Family Structure

- How would you use join to find Matt’s children or grandchildren?
  - `matt.children` // Matt’s children
  - `matt.children.children` // Matt’s grandchildren

- What if we want to find Matt’s descendants?
Box Join

- $p[q]
  - Semantically identical to dot join, but takes its arguments in different order
    \[ p[q] \equiv q.p \]
  - Example: Matt’s children or grandchildren?
    - `children[matt]` // Matt’s children
    - `children.children[matt]` // Matt’s grandchildren
    - `children[children[matt]]` // Matt’s grandchildren
Transitive Closure

- $\forall r$
  - 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.

$\forall r = r + r \cdot r + r \cdot r \cdot r + ...$

$\forall r$

(S0, S1)
(S1, S2)
(S2, S3)
(S4, S7)
(S0, S2)
(S0, S3)
(S1, S3)
Example: Family Structure

- What if we want to find Matt’s ancestors or descendants?
  - `matt.^children` // Matt’s descendants
  - `matt.^(~children)` // Matt’s ancestors

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

  ```
  no p: Person | p in p.^(~children)
  ```
Reflexive-transitive closure

\[ *r = \wedge r + \text{idem} \]

(\(S0, S1\))
(\(S1, S2\))
(\(S2, S3\))
(\(S4, S7\))

\(r\)
Domain and image Restrictions

- The restriction operators are used to filter relations to a given domain or image.

- If $s$ is a set and $r$ is a relation then
  - $s <: r$ contains tuples of $r$ starting with an element in $s$
  - $r :> s$ contains tuples of $r$ ending with an element in $s$

- Example:
  - $\text{Man} = \{(M0),(M1),(M2),(M3)\}$
  - $\text{Woman} = \{(W0),(W1)\}$
  - $\text{children} = \{(M0,M1),(M0,M2),(M3,W0),(W1,M1)\}$
  - $\text{Man} <: \text{children} = \{(M0,M1),(M0,M2),(M3,W0)\}$  // father-child
  - $\text{children} :> \text{Man} = \{(M0,M1),(M0,M2),(W1,M1)\}$  // parent-son
Override

- **p ++ q**
  - p and q are two relations of *arity two or more*
  - the result is like the union between p and q except that tuples of q can replace tuples of p. Any tuple in p that matches a tuple in q starting with the same element is dropped.
  - **p ++ q = p − (domain(q) <: p) + q**

- **Example**
  - oldAddr = {(N0,D0),(N1,D1),(N1,D2)}
  - newAddr = {(N1,D4),(N3,D3)}
  - oldAddr ++ newAddr = {(N0,D0),(N1,D4),(N3,D3)}
Logical Operators

- The usual logical operators are available
  - not
  - and
  - or
  - implies
  - else
  - <=>
    - negation
    - conjunction
    - disjunction
    - implication
    - alternative
    - iff

- Example:
  - a != b is equivalent to not a = b
Operator Precedence

- `||`
- `<>`
- `=>`
- `=>`
- `&&`
- `!`
- `= !!= in`
- `+ -`
- `++`
- `&`
- `->`
- `<<`
- `::`
- `::`
- `[[]`
- `..`
- `~ * ^`
Example: Family Structure

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

all p: Person |
(lone (p.parent & Man)) and
(lone (p.parent & Woman))

This is an example of a negative constraint that is easier to state positively (to make use of the lone operator).
Set Comprehension

\{ x : S | 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: Person | q.parent = matt.parent \}
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)”

\[
\text{all } p: \text{Person} \mid p.\text{siblings} = \\
\{q: \text{Person} \mid p.\text{parent} = q.\text{parent}\} - p
\]
Example: Family Structure

- Each married man (woman) has a wife (husband)

\[
\begin{align*}
\text{all } p: \text{Married} & | \\
(p \text{ in } \text{Man} & \Rightarrow p.\text{marriedWith in } \text{Woman}) \\
\text{and} & \\
(p \text{ in } \text{Woman} & \Rightarrow p.\text{marriedWith in } \text{Man})
\end{align*}
\]

- A spouse can’t be a sibling

\[
\begin{align*}
\text{no } p: \text{Married} & | \\
\phantom{p} & p.\text{marriedWith in } p.\text{siblings}
\end{align*}
\]
Let

- You can factor expressions out:
  \[
  \text{let } x = e \mid A
  \]
  Each occurrence of the variable \( x \) will be replaced by the expression \( e \) in \( A \)

- Example: *Each married man (woman) has a wife (husband)*

```ml
all p: Married | let spouse = p.marriedWith | (p in Man \Rightarrow spouse in Woman) and (p in Woman \Rightarrow spouse in Man)
```
**Facts**

- Additional constraints on signatures and fields are expressed in Alloy as **facts**

- AA looks for instances of a model that also satisfy all its fact constraints
Example Facts

Family Structure:

-- No person can be their own ancestor
\begin{verbatim}
fact selfAncestor {
    no p: Person | p in p.^parent
}
\end{verbatim}

-- At most one father and mother
\begin{verbatim}
fact loneParents {
    all p: Person | lone (p.parent & Man) and lone (p.parent & Woman)
}
\end{verbatim}

-- P's siblings are persons with same parents excluding P
\begin{verbatim}
fact siblingDefinition {
    all p: Person |
    p.siblings = {q: Person | p.parent = q.parent} - p
}
\end{verbatim}
Example Facts

**Family Structure:**

```plaintext
fact social {
   -- Each married man (woman) has a wife (husband)
   all p: Married |
   let spouse = p.marriedWith |
   (p in Man => spouse in Woman) and
   (p in Woman => spouse in Man)

   -- A spouse can't be a sibling
   no p: Married | p.marriedWith in p.siblings

   -- A person can't be married to a blood relative
   no p: Married |
   some (p.*parent & (p.marriedWith).*parent)
}
```
Run command

- Used to ask AA to generate an instance of the model
- May include conditions
  - Used to guide AA to pick model instances with certain characteristics
  - E.g., force certain sets and relations to be non-empty
  - In this case, not part of the “true” specification
- AA only executes the first run command in a file
Run Command

- To analyse a model, you add a run command and instruct AA to execute it.
  - run command
develops the tool to search for an instance of the model
  - you may also give a scope
  bounds the size of instances that will be considered
Scope

- **Limits the size of instances** considered to make instance finding feasible

- Represents the maximum number of tuples in each **top-level signature**

- Default value = 3
Run 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.
Run Example

Family Structure:

-- The simplest run command
-- The scope is 3
run {}

-- With conditions, forcing each set to be populated
-- Set the scope to 2
run {some Man && some Woman && Some Married} for 2

-- Other scenarios
run {some Woman && no Man} for 7
run {some Man && some Married && no Woman}
Exercises

- Load family-2.als
- Execute it
- Analyze the metamodel
- 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
Exercises

- Load family-3.als
- Execute it
- Look at the generated instance
- Does it look correct?
- How can you produce
  - two married couples?
  - a non-empty married relation and a non-empty siblings relation?
Assertions

- Often we believe that our model entails 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 is also a sibling.
  \[
  \text{all } p : \text{Person } | \text{ no } p \cdot \text{parent } \& p \cdot \text{siblings}
  \]

- A person’s siblings are his/her siblings’ siblings.
  \[
  \text{all } p : \text{Person } | \text{ p.siblings } = \text{ p.siblings.siblings}
  \]

- No person shares a common ancestor with his/her spouse (i.e., spouse isn’t related by blood).
  \[
  \text{no } p : \text{Married } | \text{ some } (p \cdot \text{parent } \&
  \quad \text{ p.marriedWith} \cdot \text{parent})
  \]
Assertion Scopes

- You can specify a scope explicitly for any signature, but:
  - If a signature has been given a bound
  - Then the bound of its parent or any other extension of the same parent can be determined
Example Scope

abstract sig Object {}
sig Directory extends Object {}
sig File extend Object {}
sig Alias extend File {}

We consider an assertion A.

- **well-formed:**
  - check A for 5 Object
  - check A for 4 Directory, 3 File
  - check A for 5 Object, 3 Directory
  - check A for 3 Directory, 3 Alias, 5 File

- **ill-formed** because it leaves the bound of File unspecified
  - check A for 3 Directory, 3 Alias
Example Scope

```plaintext
abstract sig Object {}

sig Directory extends Object {}

sig File extends Object {}

sig Alias extends File {}
```

- **check A for 5** [or] **run {} for 5**
  - places a bound of 5 on each top-level signature (in this case just Object)
- **check A for 5 but 3 Directory**
  - additionally places a bound of 3 on Directory, and a bound of 2 on File by implication
- **check A for exactly 3 Directory, exactly 3 Alias, 5 File**
  - limits File to at most 5 tuples, but requires that Directory and Alias have exactly 3 tuples each
Scope

- Size determined in a signature declaration has priority on size determined in scope

Example:

```plaintext
abstract sig Color {}
one sig red, yellow, green extends color {}
sig Pixel {color: one Color}

check A for 2
  limits the signature Pixel to 2 tuples, but assigns a size of exactly 3 to color
```
Exercises

- Load family-4.als
- Execute it
- Look at the generated counter-examples
- Why is SiblingSibling false?
- Why is NoIncest false?
Problems with Assertions

Analyzing Sibling ... Scopes: Person(3)
Counterexample found:

Person = {M,W0,W1}
Man = {M}
Woman = {W0,W1}
Married = {M,W1}

children = {(W0,W1)}
siblings = {(M,W0),(W0,M)}
marrriedWith = {(M,W1),(W1,M)}

M.siblings = {W0}
M.siblings.siblings = {M}
Problems with Assertions

Analyzing NoIncest ...
Scopes: Person(3)
Counterexample found:

Person = \{M0,M1,W\}
Man = \{M0,M1\}
Woman = \{W\}
Married = \{M1,W\}

children = \{(M0,W),(W,M1)\}
siblings = \{\}
marrriedWith = \{(M1,W),(W,M1)\}

(M0 is an Ancestor of M1 and
M0 is an ancestor of W )
and
M1 and W are married
Exercises

- 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?
Exercises

- 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.
Predicates and Functions

- Can be used as “macros”
  - Can be named and reused in different contexts (facts, assertions, and conditions of run)
  - Can be parameterized
  - Used to factor out common patterns

- Predicates are good for:
  - Constraints that you don’t want to record as fact
  - Constraints that you want to reuse in different context

- Functions are good for
  - Expressions that you want to reuse in different context
Functions

- A named expression, with zero or more arguments and an expression for the result

Examples:

- The parent relation
  
  ```
  fun parent [] : Person->Person {~children}
  ```

- Sisters
  
  ```
  fun sisters [p: Person] {
    {w: Woman | w in p.siblings} }
  ```

- No person can be their own ancestors or sisters
  
  ```
  all p: Person | not (p in p.^parent or p in sisters[p])
  ```
Predicates

- A named constraint, with zero or more arguments
- Predicates are NOT included when analyzing other schemas unless they are referenced by name in the schemas being analyzed
- Example:
  - Two persons are blood relatives iff they have a common ancestor
    
    ```
    pred BloodRelated [p: Person, q: Person] { 
      some (p.*parent & q.*parent) 
    }
    ```
  
  - A person can't be married to a blood relative
    ```
    no p: Married | BloodRelated[p, p.marriedWith]
    ```
Predicate or Fact?

- A predicate only holds when invoked while a fact always holds.
- General rule:
  - assumptions that always hold go into facts
  - other constraints go in predicates
- But
  - You can package constraints as predicates and then include the predicates in facts
- Thus predicates are more flexible than facts
Exercises

- Define a *predicate* that characterizes the notion of “in-law” for the family example
- Write an *fact* stating that a person is an in-law of their in-laws
- Add these to the family example and *run* it through AA
- Can you express this same notion in another way in the Alloy model?
  - Do so and run it through AA
  - Which approach is better? Why?
Exercises

- Add an assertion stating that a person has no married in-laws
- What is the minimum scope for set Person for which ACA can find a counterexample?
- How would you use ACA to demonstrate that your answer is truly the minimum scope?
- Demonstrate it!
Acknowledgements

- The family structure example is based on an example by Daniel Jackson distributed with the Alloy Analyzer.