Métodos Formais
2025.2

Introduction to Alloy: Constraints

Área de Teoria DCC/UFMG

Alloy Constraints

- Signatures and fields define classes (of atoms) and relations between them
- Alloy models can be refined further by adding *formulas* expressing additional constraints over those classes and relations
- Several operators are available to express both logical and relational constraints

Logical operators

The usual logical operators are available, often in two forms

```
-- not        !      (Boolean) negation
-- and        &&     conjunction
-- or         ||     disjunction
-- implies    =>     implication
-- else              alternative
--            <=>    equivalence
```

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 holds for no x in S
lone x: S | F         F holds for at most one x in S
one x: S | F          F holds for exactly one x in S
```

Predefined sets in Alloy

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

- Example. For a model instance with just:

  Man = {(M0),(M1),(M2)}
  Woman = {(W0),(W1)}

  the constants have the values

  none = {}
  univ = {(M0),(M1),(M2),(W0),(W1)}
  ident ={(M0,M0),(M1,M1),(M2,M2),(W0,W0),(W1,W1)}

Everything is a Set in Alloy

- There are *no scalars*
  - We never speak directly about elements (or tuples) of relations
  - Instead, we can use *singleton* relations:

```
one sig Matt extends Person
```

- Quantified variables *always* denote singleton relations:

```
all x : S | ... x ...
```

`x = {t}` for some element t of S

Set operators

$+$	union
&	intersection
$-$	difference
in	subset
$=$	equality
!=	disequality

- Example. Married men:

  ```
  Married & Man
  ```

Relational operators

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

Relational composition (Join)

p . q

- p and q are two relations that are *not both unary*

- p.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

How to join tuples?

- What is the join of theses two tuples ?

```
( a1 , . . . , am)
( b1 , . . . , bn )
```

- If $\mathrm{am} \neq \mathrm{b1}$, then join is undefined
- If $\mathrm{am} = \mathrm{b1}$, then it is

```
( a1 , . . . , am−1,b2 , . . . , bn )
```

- Examples.

```
( a , b ) . ( a , c , d )          undefined
( a , b ) . ( b , c , d )    =     ( a , c , d )
```

- What about (a).(a)?

How to join tuples?

- What is the join of theses two tuples ?

```
( a1 , . . . , am)
( b1 , . . . , bn )
```

- If $\mathrm{am} \neq \mathrm{b1}$, then join is undefined
- If $\mathrm{am} = \mathrm{b1}$, then it is

```
( a1 , . . . , am−1,b2 , . . . , bn )
```

- Examples.

```
( a , b ) . ( a , c , d )            undefined
( a , b ) . ( b , c , d )     =      ( a , c , d )
```

- What about (a).(a)? Not defined!
  - t1.t2 is not defined if t1 and t2 are *both* unary tuples

Example: family structure

```
abstract sig Person {
  children: set Person,
  siblings: set Person
}
sig Man, Woman, Other extends Person {}
one sig Matt in Man {}
sig Married in Person {
  spouse: one Married
}
```

How would you use join to find Matt's children or grandchildren ?

Example: family structure

```
abstract sig Person {
  children: set Person,
  siblings: set Person
}
sig Man, Woman, Other extends Person {}
one sig Matt in Man {}
sig Married in Person {
  spouse: one Married
}
```

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?

Example: family structure

How would you model the *constraint*:

Every married person has one spouse

Example: family structure

How would you model the *constraint*:

Every married person has one spouse

```
all p: Married | one p.spouse
```

A spouse can't be a sibling

Example: family structure

How would you model the *constraint*:

Every married person has one spouse

```
all p: Married | one p.spouse
```

A spouse can't be a sibling

```
no p: Married |
 p.spouse in p.siblings
```

Box Join

p [q]

- Semantically identical to dot join, but takes its arguments in different order

  p [q] <=> q . p

- Example: Matt's children or grandchildren?

Box Join

```
p[q]
```

- Semantically identical to dot join, but takes its arguments in different order

```
p[q] <=> q.p
```

- Example: Matt's children or grandchildren?

```
children[Matt]            -- Matt's children
children[children[Matt]]  -- Matt's grandchildren
```

Transpose

```
~p
```

- Take the mirror image of the relation p
  - The reverse the order of atoms in each tuple

```
p[q] <=> q.p
```

- Example:

```
 p = {(a0,a1,a2,a3),(b0,b1,b2,b3)}
~p = {(a3,a2,a1,a0),(b3,b2,b1,b0)}
```

- Example: Matt's parents or grand parents?

Transpose

```
~p
```

- Take the mirror image of the relation p
  - The reverse the order of atoms in each tuple

```
p[q] <=> q.p
```

- Example:

```
 p = {(a0,a1,a2,a3),(b0,b1,b2,b3)}
~p = {(a3,a2,a1,a0),(b3,b2,b1,b0)}
```

- Example: Matt's parents or grand parents?

```
~children[Matt]             -- Matt's parents
~children[~children[Matt]]  -- Matt's grandparents
```

Transitive Closure

```
^r
```

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

```
^r = r + r.r + r.r.r + ...
```

Example: family structure

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

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 to express the constraint "No person can be their own ancestor?"

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 to express the constraint "No person can be their own ancestor?"

```
no p: Person | p in p.^(~children)
```

Reflexive-transitive Closure

```
*r = ^r + iden
```

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

```
*r = iden + r + r.r + r.r.r + ...
```

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 (same as flat cross product)
- Example

```
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)}
```

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
- Examples

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

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)}
```

Operator precederce

From lower to higher:

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

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?

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.~children = matt.~children }
```

Example: family structure

How to express the constraint "A person P's siblings are those people, other than P, with the same parents as P"

Example: family structure

How to express the constraint "A person P's siblings are those people, other than P, with the same parents as P"

```
all p: Person |
    p.siblings =
      {q: Person | p.~children = q.~children} - p
```

Functions and Predicates

- Parametrized macros for terms and formulas
  - Can be named and reused in different contexts (facts, assertions and conditions of run)
  - Can have zero or more parameters
  - Used to factor out common patterns

- Functions are good for *set expressions* you want to reuse in different contexts

- Predicates are good for *formulas* you want to reuse in different contexts

Functions

- A named *set expression*, with zero or more parameters

- The parents relation:

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

fun parents [] : Person -> Person {~children}
```

- Example in a formula:

```
all p: Person |
   p.siblings =
      {q: Person | p.parents = q.parents} - p
```

Predicates

- A named *formula*, with zero or more parameters

- The parents relation:

```
pred BloodRelated [p: Person, q: Person] {
  some (p.*parents & q.*parents)
}
```

- Example in a formula:

```
no p: Married | BloodRelated[p, p.spouse]
```

Let

```
let x = e | A
```

- You can factor expressions out
- Each occurrence of the variable x will be replaced by the expression e in A
- Example: "Each married peson has one spouse"

```
all p: Married |
  let q = p.spouse | one q
```

Facts

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

```
fact Name {
   F1
   F2
    ...
}
```

- AA looks for instances of a model that also satisfy all of its *facts*

Example Facts

- No person can be their own ancestor

Example Facts

- No person can be their own ancestor

```
fact selfAncestor {
  no p: Person | p in p.^parents
}
```

Example Facts

- No person can be their own ancestor

```
fact selfAncestor {
  no p: Person | p in p.^parents
}
```

- a persons's siblings are other persons with the same parents

Example Facts

- No person can be their own ancestor

```
fact selfAncestor {
  no p: Person | p in p.^parents
}
```

- a persons's siblings are other persons with the same parents

```
fact siblingsDefinition {
  all p: Person |
    p.siblings =
      {q: Person | p.parents = q.parents} - p
}
```

Example Facts

```
fact social {
  -- Every married person has one spouse
   all p: Married |  one p.spouse

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

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

Assertions

- Often we believe that our model *entails* certain *constraints* that are not directly expressed
  - `some A && (A in B)` entails `some B`
- We can define these constraints as assertions and ask the analyzer to check if they hold (similarly specifying checking scopes)

```
assert myAssertion { some B }
check myAssertion for 5
```

- If the constraint in an assertion does not hold, the analyzer will produce a *counterexample instance*
- If you expect the constraint to hold but it does not, you can either
  - make it into a fact, or
  - refine your model until the assertion holds

Example Assertions

- No person has a parent that is also a sibling

Example Assertions

- No person has a parent that is also a sibling

```
assert a1 { all p: Person |
              no p.parents & p.siblings }
```

Example Assertions

- No person has a parent that is also a sibling

```
assert a1 { all p: Person |
              no p.parents & p.siblings }
```

- A person's siblings are his/her siblings' siblings

Example Assertions

- No person has a parent that is also a sibling

```
assert a1 { all p: Person |
              no p.parents & p.siblings }
```

- A person's siblings are his/her siblings' siblings

```
assert a2 { all p: Person |
              p.siblings = p.siblings.siblings }
```

Example Assertions

- No person has a parent that is also a sibling

```
assert a1 { all p: Person |
                no p.parents & p.siblings }
```

- A person's siblings are his/her siblings' siblings

```
assert a2 { all p: Person |
                p.siblings = p.siblings.siblings }
```

- No person shares a common ancestor with their spouse (i.e., spouse isn't related by blood)

Example Assertions

- No person has a parent that is also a sibling

```
assert a1 { all p: Person |
              no p.parents & p.siblings }
```

- A person's siblings are his/her siblings' siblings

```
assert a2 { all p: Person |
              p.siblings = p.siblings.siblings }
```

- No person shares a common ancestor with their spouse (i.e., spouse isn't related by blood)

```
assert a3 { no p: Married |
            some (p.^parents & p.spouse.^parents) }
```

Acknowledgments

These notes are heavily based on notes from Matt Dwyer, John Hatcliff, Rod Howell, Laurence Pilard and Cesare Tinelli.