CS:5810
Formal Methods in Software Engineering

Introduction to Alloy

Part 2

Alloys Constraints

- Signatures and fields resp. 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 Set Constants

There are three predefined set constants in Alloy:

- `none` : empty set
- `univ` : universal set of all atoms
- `iden` : identity relation over all atoms

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)}
iden ={(M0,M0),(M1,M1),(M2,M2),(W0,W0),(W1,W1)}
```

Everything is a Relation in Alloy

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

```
one sig Matt extends Person {}
```

- Quantified variables always denote singletons:

```
all x : S | … x …
```

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

Set Operators and Predicates

`+`	union	operators
`&`	intersection	
`-`	difference	
`in`	subset	predicates
`=`	equality	
`!=`	disequality	

Example. Matt is a married man:

```
Matt in (Married & Man)
```

Relational Operators

->	arrow (cross 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 (same as *flat* cross product)

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

`~ p`

take the mirror image of the relation `p`,
i.e., reverse the order of atoms in each tuple

Example

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

How would you use `~` to express the `parents` relation if you already have the `children` relation?

`~children`

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 ?
  - $(a_1, ..., a_m)$
  - $(b_1, ..., b_n)$

  If $a_m \neq b_1$ then the join is undefined

  If $a_m = b_1$ then it is: $(a_1, ..., \mathbf{a_{m-1}}, \mathbf{b_2}, ..., b_n)$

  Example

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

- What about (a) . (a) ? Not defined !

  $t_1.t_2$ is not defined if t_1 and t_2 are **both** unary tuples

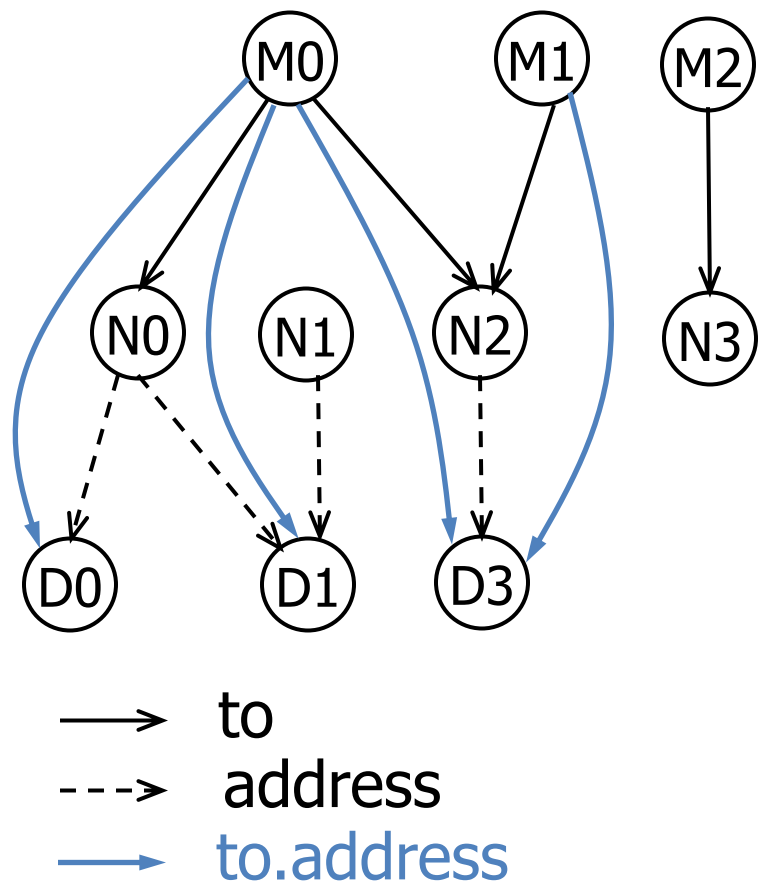
Examples

- to maps a message to the name(s) it should be sent 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 address(es) it should be sent to

```
to.address = {(M0,D0),
  (M0,D1),(M0,D3),(M1,D3)}
```



Exercise

What's the result of these join applications?

1. {(a,b)}.{(c)}
2. {(a)}.{(a,b)}
3. {(a,b)}.{(b)}
4. {(a)}.{(a,b,c)}
5. {(a,b,c)}.{(c,e),(c,d),(b,c)}
6. {(a,b)}.{(a,b,c)}
7. {(a,b,c,d)}.{(d,e,f),(d,a)}
8. {(a)}.{(b)}

Exercises

- Given a relation `addr` of arity 4 that contains the tuple `b->n->a->t` when book `b` maps name `n` to address `a` at time `t`, and given a specific 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.addr.T` is the name-address mapping of book `B` at time `T`. What is the value of `B.addr.T` ?

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

- Join is not associative (i.e., (p.q).r and p.(q.r) are not always equivalent), why ?

Example: Family Structure

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

sig Man, Woman extends Person {}

one sig Matt extends Person {}

sig Married in Person {
   spouse: one Married
}
```

Example: Family Structure

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

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

Every married man (woman) has a wife (husband)

One's spouse can't be one's sibling

Example: Family Structure

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

Every married man (woman) has a wife (husband)

```
all p: Married |
 (p in Man => p.spouse in Woman)
 and
 (p in Woman => p.spouse in Man)
```

One’s spouse can’t be one’s 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 ?

- `children[Matt]` // Matt’s children
- `children.children[Matt]` // Matt’s grandchildren
- `children[children[Matt]]` // Matt’s grandchildren

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

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

r

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

^r

`^r = r + r.r + r.r.r + …`

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 ≡ ^r + iden`

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

r

$$\left.\begin{array}{l}\left.\begin{array}{l}(S0,S1)\\(S1,S2)\\(S2,S3)\\(S4,S7)\\(S0,S2)\\(S0,S3)\\(S1,S3)\end{array}\right\}\ \text{^r}\\[4pt]\left.\begin{array}{l}(S0,S0)\\(S1,S1)\\(S2,S2)\\(S3,S3)\\(S4,S4)\\(S7,S7)\end{array}\right\}\ \text{iden}\end{array}\right.\quad \text{*r}$$

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

```
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 Precedence

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

Low

High

Example: Family Structure

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

Example: Family Structure

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

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

Equivalently:

```
all p: Person |
 (lone (Man <: children).p) and
 (lone (Woman <: children).p)
```

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?

(assuming `Person` has a `parents` field)

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.parents = matt.parents }
```

(assuming `Person` has a `parents` field)

Example: Family Structure

How would you express the constraint

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

Example: Family Structure

How would you 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.parents = q.parents} - p
```

Let

You can factor expressions out:

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

– Each occurrence of the variable `x` in `A` will be replaced by the expression `e`

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

```
all p: Married |
  let q = p.spouse |
    (p in Man => q in Woman) and
    (p in Woman => q in Man)
```

Acknowledgements

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

Exercise

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

Write facts stating the following:

1. Siblings have the same father and the same mother
2. Two married people have the same children