CS:5810 Formal Methods in Software Engineering

Introduction to Alloy 5

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
- `_ iff _` `_ <=> _` equivalence

Quantifiers

Alloy includes a rich collection of quantifiers

`all x: S	F`	states that F holds for every x in S
`some x: S	F`	states that F holds for some x in S
`no x: S	F`	states that F holds for no x in S
`lone x: S	F`	states that F holds for at most one x in S
`one x: S	F`	states that F holds for exactly one x in S

Quantifiers

Alloy includes a rich collection of quantifiers

```
all x: S | F       (e.g., all m : Man | m in Person)
some x: S | F      (e.g., some p : Person | p in Man)
no x: S | F        (e.g., no p : Person | m in Man & Woman)
lone x: S | F      (e.g., lone m : Man | m in Matt.children)
one x: S | F       (e.g., one m : Woman | m in Matt.children)
```

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 Man {}
```

- Quantified variables always denote singletons:

```
all x : S | … x …
```

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

Predefined Set Constants

There are three predefined set constants in Alloy:

- **`none`** : empty set
- **`univ`** : universal set of all atoms
- **`ident`** : 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)}
ident ={(M0,M0),(M1,M1),(M2,M2),(W0,W0),(W1,W1)}
```

Set Operators and Predicates

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

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?

  $(a_1,...,a_m)$ and $(b_1,...,b_n)$

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

  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

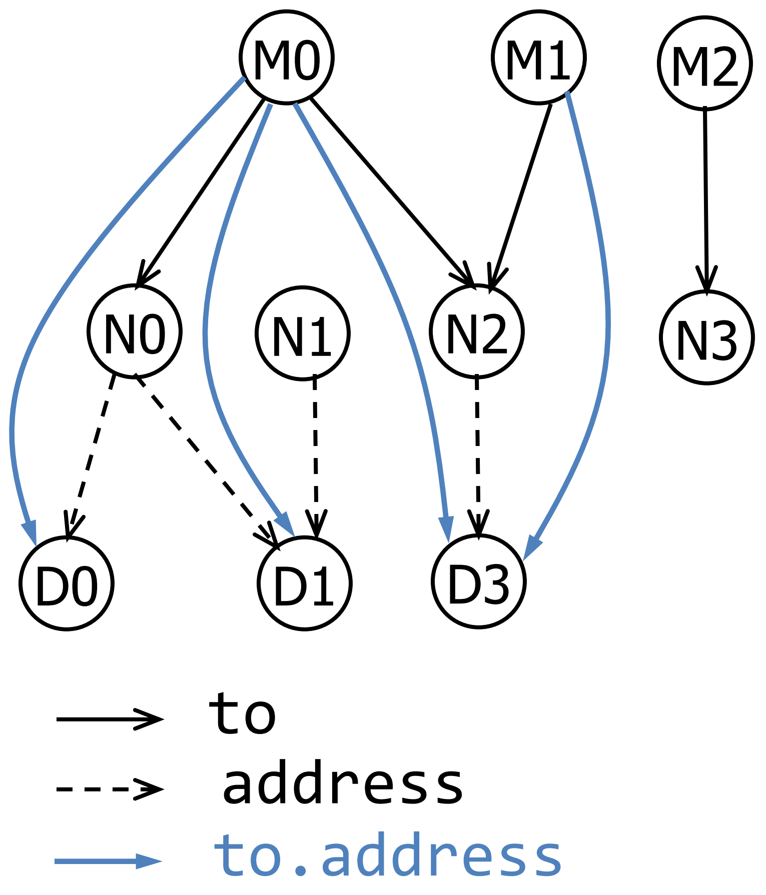
Examples

- `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 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

```
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?

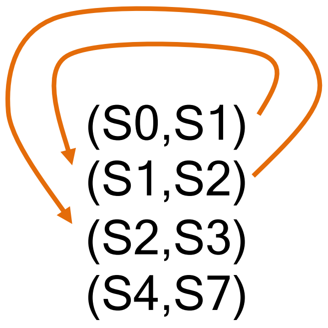
```
– children[Matt]              ≡ Matt.children
– children.children[Matt]     ≡ (children.children)[Matt]
                              ≡ Matt.(children.children)
– children[children[Matt]]    ≡ children[Matt.children]
                              ≡ (Matt.children).children
```

Transitive Closure

`^ r`

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

`r`



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

`^r`

– Formally, `^r` is the smallest transitive relation of type `S -> S` that contains `r`

`^r = r + r.r + r.r.r + …`

Example: Family Structure

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

- How would you 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
  - `^(children).Matt` // also Matt’s ancestors

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

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

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)}
```

Reflexive-transitive closure

`*r ≡ ^r + iden :> S` for `r : S -> S`

`r`

(S0,S1)
(S1,S2)
(S2,S3)
(S4,S5)

`S` = { S0, …, S5 }

`*r`

`^r`:

(S0,S1)
(S1,S2)
(S2,S3)
(S4,S5)
(S0,S2)
(S0,S3)
(S1,S3)

`iden :> S`:

(S0,S0)
(S1,S1)
(S2,S2)
(S3,S3)
(S4,S4)
(S5,S5)

`*r` is the smallest reflexive and transitive relation of type `S -> S` that contains `r`

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

High → Low

```
~   *   ^
.
[ ]
<:   :>
->
&
++
#
+   -
no   some   lone   one   set    // multiplicities
=   !=   in   !in
!   not
&&   and
=>   implies   else
<=>   iff
||   or
let   all   no   some   lone   one   // binders
```

relations: `~ * ^` through `+ -`

formulas: `no some lone one set` through `let all no some lone one`

Parsing Conventions

- All binary operators associate to the left, except for implication which associates to the right

  Ex. `a & b & c` is parsed as `(a & b) & c`

  `p => q => r` is parsed as `p => (q => r)`

- In an implication, an else-clause is associated with its closest then-clause

  Ex. `p => q => r else s` is parsed as `p => (q => r else s)`

Note: The scope of a quantifier extends as far as possible to the right

Ex. `all x : A | p & q => r` is parsed as `all x : A | (p & q => r)`

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))
```

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)*

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)
```

Let

You can factor expressions out:

```
let x = e { A1 … An }
```

– 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
     p in Woman => q in Man
  }
```

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. Two married people have the same children

2. Siblings have the same father and the same mother

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. Two married people have the same children

```
all p: Married | p.children = p.spouse.children
```

2. Siblings have the same father and the same mother

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. Two married people have the same children

2. Siblings have the same father and the same mother

```
all p: Person | all q: p.siblings {
  children.p & Man = children.(p.siblings) & Man
  children.p & Woman = children.(p.siblings) & Woman
}
```

Acknowledgements

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