CS:5810 Formal Methods in Software Engineering

Introduction to Alloy

Part 2

Alloys Constraints

- Signatures and fields define classes (of atoms) and relations between them
- Alloy models can be refined further by adding formulas that express additional constraints over those sets 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 1 x in S
  - `one x: S | F` — F holds for exactly 1 x in S

Predefined Sets in Alloy

- Three predefined set constants:
  - **`none`** : empty set
  - **`univ`** : universal set
  - **`ident`** : identity

- Example. For a model with just the two sets:

```
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

- Ex: Married men,

```
Married & 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

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

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

- How would you use ~ to express the parents 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: $(a_1, ..., a_{m-1}, b_2, ..., b_n)$

- Examples :
  - (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

Exercises

- What's the result of these join applications?
  - `{(a,b)}.{(c)}`
  - `{(a)}.{(a,b)}`
  - `{(a,b)}.{(b)}`
  - `{(a)}.{(a,b,c)}`
  - `{(a,b,c)}.{(c)}`
  - `{(a,b)}.{(a,b,c)}`
  - `{(a,b,c,d)}.{(d,e,f)}`
  - `{(a)}.{(b)}`

Examples:

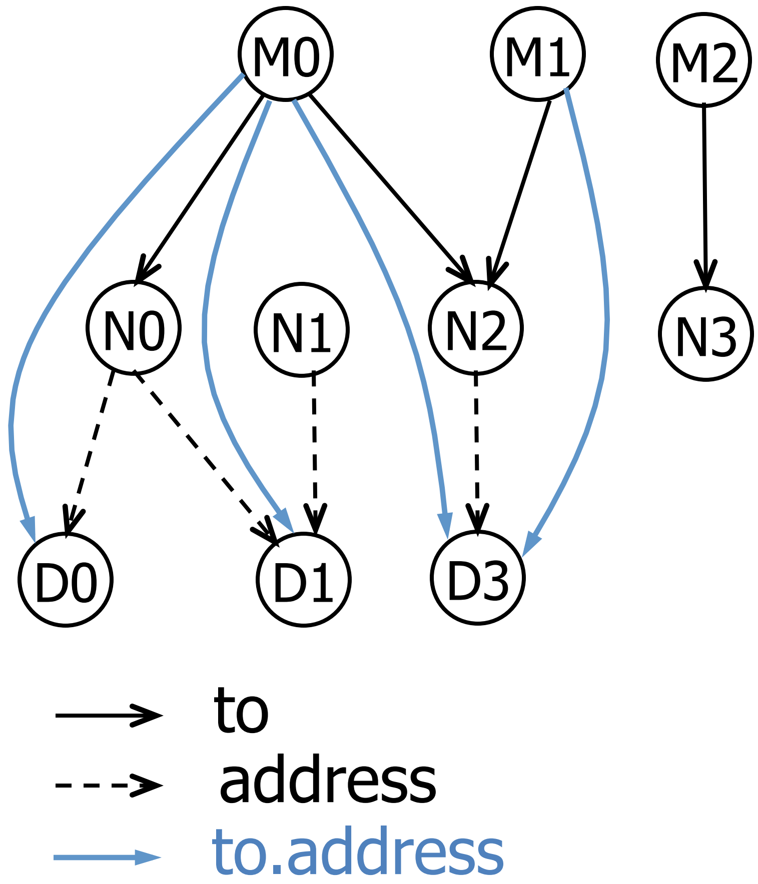
to maps a message to the name it's intended to be send to

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 4 that contains the tuple `b->n->a->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.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, why ?

  (i.e. (p.q).r and p.(q.r) 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

- ^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

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

^r

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

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))
```

- 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.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
```

Example: Family Structure

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

- *A spouse can't be a sibling*

Example: Family Structure

- *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)
```

- *A spouse can't be a sibling*

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

Let

- You can factor expressions out:

```
let x = e | 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)*

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

-- At most one father and mother

-- P's siblings are persons with same parents excluding P
```

Example Facts

Family Structure:

```
-- No person can be their own ancestor
fact selfAncestor {
   no p: Person | p in p.^parents
}

-- At most one father and mother
fact loneParents {
   all p: Person |   lone (p.parents & Man)   and
                     lone (p.parents & Woman)
}

-- P's siblings are persons with same parents excluding P
fact siblingsDefinition {
  all p: Person |
    p.siblings = {q: Person | p.parents = q.parents} - p
}
```

Example Facts

Family Structure:

```
fact social {
   -- Every married man (woman) has a wife (husband)



   -- A spouse can't be a sibling


   -- A person can't be married to a blood relative


}
```

Example Facts

Family Structure:

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

   -- 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)
}
```

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

Run Command

- To analyze a model, you add a run command and instruct AA to execute it.
  - the run command

    tells 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

- AA executes only the first run command in a file

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.
- Condition schemas can be used to implement *constraint macros*
  - This allows common constraints to be shared

Run Example

Family Structure:

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

-- The scope is 4
run {} for 5

-- 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
  - e.g., some A & (A in B) entails some B

- We can define these additional constraints as assertions and use the analyzer to check if they hold
  - e.g., assert myAssertion { some B }
    check myAssertion

Assertions

- 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

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 his/her spouse (i.e., spouse isn't related by blood)

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

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 supersignature or any other extension of the same supersignature 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

```
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:

```
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 elements, but assigns a size of exactly 3 to `Color`

Exercises

- Load `family-4.als`
- Execute it
- Look at the generated counter-examples
- Why is SiblingsSibling false?
- Why is NoIncest false?

Problems with Assertions

```
Analyzing SiblingSiblings ...
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)}
  spouse = {(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 = {}
   spouse = {(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 you don’t want to record as fact
  - Constraints you want to reuse in different contexts
- Functions are good for
  - Expressions you want to reuse in different contexts

Functions

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

- Examples:

  – The parents relation

```
fun parents [] : 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.^parents or
                     p in sisters[p])
```

Predicates

- A named constraint, with zero or more arguments
- Predicates are not included when analyzing other schemas (e.g., facts or assertions) unless they are applied to actual arguments 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.*parents & q.*parents)
}
```

  - A person can't be married to a blood relative

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

Predicate or Fact ?

- Predicates are (parametrized) definitions of constraints
- Facts are assumed constraints
- Note: You can package constraints as predicates and then include the predicates in 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.