CS:5810 Formal Methods in Software Engineering

Introduction to Alloy

Part 3

Facts

Explicit 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 fact constraints

Example Facts

```
-- No person can be their own ancestor




-- At most one father and mother





-- 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
}

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

-- 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 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)

   -- one’s spouse can't be one’s 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 Example

Family Structure:

```
-- The simplest run command
-- The scope of every signature is 3
run {}

-- The scope scope of every signature is 5
run {} for 5

-- With conditions forcing each set to be populated
-- Setting 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}
```

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 to signatures

    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 elements in a top-level signature
- Default value = 3 for each top-level signature

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

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 Signatures

- The analyzer's algorithms prefer smaller instances
  - Often it produces empty signatures 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) and (A in B) entails some B
- We can define these constraints as assertions and ask 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 an assertion to hold but it does not, you can either
  - add it directly as a fact, or
  - refine your model with other constraints 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
- However, if a signature has been given a scope, then
  - a scope of its subignatures can be always determined
  - sometimes the scope of its supersignatures can be determined as well
- The AA will compute the tightest scope it can

Scope Examples

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

We consider some assertion A

- all well-formed commands:

```
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, for leaving 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 in 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

Size Determination

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 in `family-4.als`
  - 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?

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

Examples:

- The sisters function

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

- The parents relation defined as a constant function

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

- Used in a formula

```
all q: Person | not (q in q.^parents or
                     q in sisters[q])
```

Predicates

A named formula, with zero or more parameters

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 [p1: Person, p2: Person] {
  some (p1.*parents & p2.*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 use those predicates in facts

```
pred IsSingle[p: Person] { not (p in Married) }
pred IsFather[p: Man] { some p.children }

fact { some q: Man | IsSingle[q] && IsFather[q] }
```

Exercises

- Define a predicate `IsChildless` that characterizes the notion of not having children

- Define a function `father` that returns the father of a given person

Exercises

- Define a predicate that characterizes the notion of "in-law" for the family example
- Write a 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 prove that your answer is truly the minimum scope?
- prove it!

Acknowledgements

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