# CS:5810 Formal Methods in Software Engineering

## Introduction to Alloy 6

## Part 3

# Facts

Explicit constraints on signatures and fields are expressed in Alloy as facts

```
fact Name {
  Formula1
  Formula2
  …
}
```

AA generates only instances that also satisfy all of the `fact` constraints in a model

# Example Facts

```
-- No one 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 person'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

```
-- No one 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          // alternative syntax for
                  lone p.parents & Woman }      // conjunctive body
}

-- A person's siblings are other persons with the same parents
fact siblingsDefinition {
  all p: Person |
    p.siblings = {q: Person | p.parents = q.parents} - p
}
```

Formulas separated by white space in a { ... } block are treated conjunctively

# Example Facts

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

  -- 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
}
```

Formulas separated by white space in a { ... } block are treated conjunctively

# Run Command

- Used to ask AA to generate an instance of the model

- May include *run 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 and some Woman and some Married} for 2

-- Other scenarios with conditions
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 run conditions to encode *realism constraints*

– e.g., to force generated models to include at least one married person, or one married man, etc.

Run conditions can abstracted in *constraint macros* via the definition of *predicates*

– 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 favors smaller model instances

- It often produces empty signatures or otherwise trivial instances
- It is useful to know that these instances satisfy the constraints (especially if you do not want them to)

Usually, small instances 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 expect our model to entail additional 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 BNonEmpty { some B }
check BNonEmpty
```

# Assertions

If the constraint in an assertion does not hold (i.e., does not follow from the model) 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, or
- reflect on whether your expectation that it held was correct to start with!

# Assertions

- No one has a parent who 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 one 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
}
```

# 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 scopes it can

# Scope Examples

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

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, 5 File, 3 Alias
```

- ill-formed, for leaving the scope of `File` unspecified:

```
check A for 3 Directory, 3 Alias
```

# Scope Examples

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

  places a bound of 3 just 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 `Directory` and `Alias` to have exactly 3 tuples each

# Size Determination

Size determined by a signature declaration has priority over 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 counterexamples
- 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 counterexamples
  - Does varying the scope make a difference?
  - Does this mean that the assertions hold for all models?

# Functions and Predicates

Parametrized macros for relational expressions and formulas

- Can be named and reused in different contexts (facts, assertions, and run conditions)
- Can have zero or more parameters
- Used to abstract and factor out common patterns

Functions are good for:

- relational expressions you want to reuse in different contexts

Predicates are good for:

- formulas you want to reuse in different contexts

# Predicates

A named formula template, with zero or more parameters

**Examples:**

– Two people 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]
                some (p.*parents & p.spouse.*parents)
```

**Note:** Predicates are ignored unless they are applied to terms in a fact or assertion

# Functions

A named relation expression template, 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
}
```

```
q.^parents
=
q.^~children
```

```
sisters[q]
=
{w: Woman | w in q.siblings}
```

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

# Predicate or Fact ?

- Predicates are (parametrized) definitions of constraints
- Facts are assumed constraints

**Note:** You can package constraints as predicates and then instantiate 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

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

2. Define a function `father` that returns the father of a given person

# Exercises

1. Define a binary predicate that characterizes the notion of "in-law" (mother/father/brother/sister/son/daugther) for the family example
2. Write a fact stating that a person is an in-law of their in-laws
3. Add these to one of the family examples and run it through AA
4. Can you express this same notion in another way in the Alloy model?
    a) Do so and run it through AA
    b) Which approach is better?  Why?

# Exercises

1. Add an assertion stating that a person has no married in-laws
2. What is the minimum scope for set `Person` for which AA can find a counterexample?
3. How would you use AA to prove that your answer is truly the minimum scope?
4. Prove it!

# Acknowledgements

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