CIS 771: Software Specifications

Lecture 4: More Alloy Basics

Outline

- Alloy: Rationale and Use Strategies
  - What types of systems have been modeled with Alloy
  - What types of questions can ACA answer
  - What is the purpose of each of the sections in an Alloy specification

- Additional Alloy Features
  - Parameterized conditionals
  - Indexed relations
**Alloy --- Why was it created?**

- **Lightweight**
  - small and easy to use, and capable of expressing common properties tersely and naturally

- **Precise**
  - having a simple and uniform mathematical semantics

- **Tractable**
  - amenable to efficient and fully automated semantic analysis

**Alloy --- Comparison**

- **UML**
  - Has similarities (graphical notation, OCL constraints) but it is neither lightweight, nor precise
  - UML includes many modeling notions omitted from Alloy (use-cases, state-charts, code architecture specs)
  - Alloy's diagrams and relational navigation are inspired by UML

- **Z**
  - Precise, but intractable. Stylized typography makes it harder to work with.
  - Z is more expressive than Alloy, but more complicated
  - Alloy's set-based semantics is inspired by Z
Alloy --- What is it used for?

- Alloy supports high-level semantic object modeling
  - “object modeling” != “object oriented code design”
  - “object modeling” means modeling with entities where identity is a primitive notion (not exact)
- Alloy is meant for modeling relationships and constraints between basic semantic entities
- Alloy is not meant for modeling code architecture (ala class diagrams in UML) although there might be a close relationship between the Alloy specification and an implementation in an OO language
  - Alloy does not include notions like field, method, or even integer values

Alloy --- Example Applications

- The ACA distribution comes with over 15 example models that together illustrate all of Alloy’s constructs.
  - Legal configurations of a package router
  - Composition issues in Microsoft’s COM
  - Runtime configurations of DFC (telephone switching)
  - Basic structures in an air-traffic control system
Alloy Specifications

- Domains
- State schema
- Definitions/Invariants
- Conditions
- Assertions

Domains

- Basic sets that you want to reason about
- Domains are fixed (concept related to operations and dynamic models)
- Domains are disjoint
Example Domains

Family Structure:

```alloy
model Family {
  domain {Person}
  ...
}
```

Employment Structure:

```alloy
model Employment {
  domain {Person, Company}
  ...
}
```

State Schemas

- Describes the data values that you want to reason about
- Define sets as subsets of other sets and domains
- Define relations between sets and domains
- Simple constraints
  - Multiplicities on sets
  - Multiplicities on relations
  - Fixed/static constraints on sets and relations
Example State Schema

Family Structure:

```alloy
model Family {
    domain {Person}
    state {
        partition Man, Woman : Person
        Married : Person
        siblings : Person -> Person
        children (~parents) : Person -> Person
        wife (~husband) : Man? -> Woman?
    }
}
```

Example State Schema

Employment Structure:

```alloy
model Employment {
    domain {Person, Company}
    state {
        disjoint alice, bob, carol : fixed Person!
        borland, compaq : fixed Company!
        Employee : Person
        Boss, Lackee, Ceo : Employee

        // at least one company per Employee
        employer (~employees) : Employee -> Company+

        // each Lackee has at least one Boss (and vice versa)
        boss : Lackee+ -> Boss+
    }
}
```
Definition/Invariant Schemas

- Constraints on state data
- In general we will ignore the distinction between definitions and invariants
  - Using Alloy's default settings, there is no difference.
  - Advance users can selectively include/exclude them to gain efficiency in the constraint analyzer
- With Alloy’s default settings, definitions and invariants are always included with analyzing a state schema.

Example Invariants

**Family Structure:**

```
inv Biological {
    // nobody's their own ancestor
    no p | p in p.+parents
    // at most one father and mother
    all p | (sole p.parents & Man) && (sole p.parents & Woman)
    // p's siblings are persons with same parents excluding p
    all p | p.siblings = {q | q.parents /= p.parents} - p
}
```

```
inv Social {
    // each married man has a wife
    // and everyone with a wife is a married man
    all p | some p.wife <-> p in Man & Married
    // each married woman has a husband etc.
    all p | some p.husband <-> p in Woman & Married
    // wife isn't sibling
    no p | p.wife /in p.siblings
    no p | p.wife /in p.siblings
}
```
**Condition Schemas**

- Used to guide ACA to pick model instances with certain characteristics
  - E.g., force certain sets & relations to be non-empty
  - In this case, not part of the “true” specification
- Can be used as “constraint macros”
  - Can be named as used in other invariants, assertions and conditions
  - Can be parameterized
  - Used to factor out common constraint patterns
- Conditions are NOT included when analyzing other schemas unless they are referenced by name in the schemas being analyzed

**Example Conditions**

Family Structure:

// Force each basic set to be populated
cond C1 {some Man && some Woman && Some Married}

// Other scenarios
cond C2 {some Woman && no Man}
cond C3 {some Man && some Married && no Woman}
Example Conditions

Family Structure: using a condition as a constraint

// two people are cousins
//   if their parents are siblings
cond Cousins (p : Person!, q : Person!) {
    some (p.parents.siblings & q.parents.siblings)
}

inv MoreSocial {
    // cousins cannot marry
    no p | Cousins(p, p.wife)

    // a person can't have children with a cousin
    all p | all q | (some (p.children & q.children) && p != q) --> not Cousins(p, q)
}

For you to do (pause here)

- Define a condition that characterizes the notion of “in-law” for the family example
- Write an invariant stating that a person is an in-law of their in-laws
- Add these to the family example and run it through ACA
- Can you express this same notion in another way in the Alloy model?
  - Do so and run it through ACA
  - Which approach is better? Why?
 Assertion Schemas

- Used to properties which you believe can be derived from the existing specification
- Assertions are NOT included as additional constraints when analyzing other schemas
- Are distinguished from other schemas by the way ACA responds
  - ACA doesn’t give an example instance
  - Instead, it tries to see if the assertion can be falsified within the given scope
  - If an assertion can be falsified, the falsifying instance (a “counterexample”) is displayed
- Important: if no counterexample is found, this doesn’t mean that the assertion is true. It means that it can’t be falsified within the given scope

Example Assertions

Family Structure:

```
// no person has a parent who's also a sibling. valid
assert ParentsArentSiblings { all p | no p.parents & p.siblings }

// a person's siblings are his/her siblings' siblings. invalid
assert SiblingsSiblings { all p | p.siblings = p.siblings.siblings }

// no person has a wife with whom he ...
// ... shares a common ancestor. invalid
assert NoIncest { no p | some (p.*parents & p.wife.*parents) }

// cannot be falsified with SCOPE=1
assert AtMostOneMan {sole Man}
```
For you to do (pause here)

- Add an assertion stating that a person has no married in-laws
- What is the minimum scope for domain Person for which ACA can find a counter-example?
- How would you use ACA to demonstrate that your answer is truly the minimum scope?
- Demonstrate it!

Indexed Relations

- Often we would like to have a family of relations indexed by a certain domain

- Family Structure Example: Matchmakers
  - Persons who keep tables of people that should get married
  - Matchmakers themselves should be married (experience)
  - A matchmakers list should relate single men to single women
Example: Family Structure

```alloy
model Family {
  domain {Person}
  state {
    ...# Revised
    partition Single, Married : Person
    matches[Person] : Single -> Single
  }
}
```

- We want domain set to be Single and Man and range set to be Single and Woman (but we can't do that using Alloy's declarations)
- We want index set to be Married (but Alloy only allows a domain as an index set)
- These additional constraints must be imposed in definition/invariant schemas.

Example: Family Structure

- Must be married to be a matchmaker

  ```alloy
  all p | p !in Married -> all q | no q.matches[p]
  ```

  Intuition: if you're not married, you can't maintain a table of matches

- Match tables must relate men (domain) and women (range)

  ```alloy
  all p | all q | q.matches[p] in Woman &
  q.-matches[p] in Man
  ```
For you to do

- Add the sets and constraints for matchmakers to the family example
- Generate an instance of the model with ACA that has a matchmaker table
  - Do you need to add any conditions to force this?
  - What is the smallest scope you can use to produce an instance?