Alloy

Vatche Ishakian

Boston University- CS511

March/24/2008

Contributors: Andrei Lapets, Michalis Potamias, Mark Reynolds

Why write a Model

- To describe formally the components of a system and relationships between them
- Check properties about the model
- Exclude ill-formed examples of Models.
- Two kind of problems might arise
  - Bugs (in the logic of the model)
  - Errors (in the subject that you are modeling)

Why Alloy

- Conceptual simplicity and minimalism
  - Very little to learn
  - WYSIWYG: no special semantics
- high-level notation
  - Constraints -- can build up incrementally
  - Relations flexible and powerful
  - Much more succinct than most model checking notations

Ref: Alloy in 90 minutes. http://people.csail.mit.edu/dnj/talks/re05-tutorial/alloy-90.pdf

Alloy Case Studies

- In Industry
  - Animating requirements (Venkatesh, Tata)
  - Military simulation (Hashii, Northtrop Grumman)
  - Role-based access control (Zao, BBN)
  - Generating network configurations (Narain, Telcordia)
- In Research
  - security features (Pincus, MSR)
  - exploring design of switching systems (Zave, AT&T)

Ref: Alloy in 90 minutes. http://people.csail.mit.edu/dnj/talks/re05-tutorial/alloy-90.pdf

Alloy Characteristics

- Finite scope check:
  - The analysis is sound, but incomplete
- Infinite model:
  - Finite checking does not get reflected in your model
- Declarative: first-order relational logic
- Automatic analysis:
  - visualization a big help
- Structured data:

Bad Things

- Sequences are awkward
- Recursive functions hard to express

Alloy model structures

- Structures are expressed as a set of tuples (vectors of atoms)
  - individual elements are treated as singleton sets
- Atoms are Alloy's primitive entities
  - Indivisible: can’t be broken down into smaller parts
  - Immutable: it does not change.
- Relations
  - Associate atoms with one another.
  - Consists of a set of tuples eache tuple being a sequence of atoms.

Relations

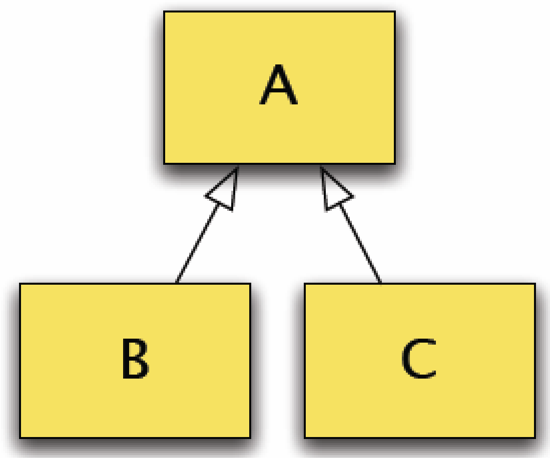
- The "arity" of the relation is the number of atoms in each tuple
- A relation with no tuples is empty

Alloy declarations

- First line module declaration
  - module chapter4/filesystem
- sig Object {}
  - Defines a set named Object represents all objects
  - abstract sig Object{} abstract sig has no elements except those belonging to its extensions.
- sig File extends Object {}
  - A set can be introduced as a subset of another set

Signatures

- sig A {}
- sig B extends A {}
- sig C extends A {}



- Means
  - B **in A**
  - C **in A**
  - **no B & C**

Ref: Alloy in 90 minutes. http://people.csail.mit.edu/dnj/talks/re05-tutorial/alloy-90.pdf

Signatures

- **abstract sig A {}**
- **sig B extends A {}**
- **sig C extends A {}**
- *B in A*
- *C in A*
- *no B & C*
- *A = (B + C)*

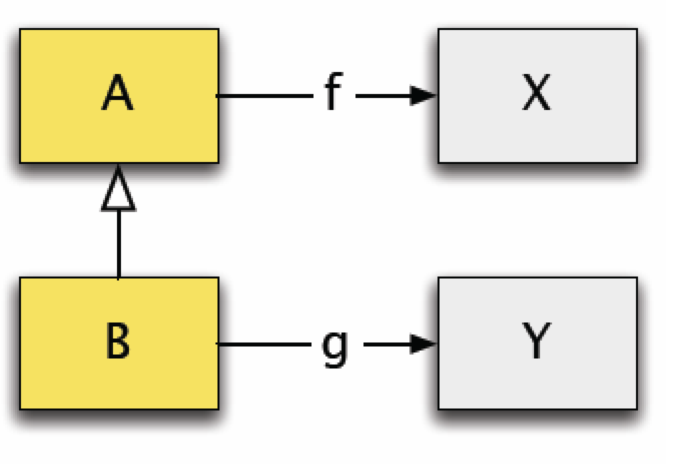
Declaring relations

- Relations are declared as fields of signatures
- sig Object{}
  sig Dir extends Object {

  ```
      entries: set Object,
      parent: lone Dir
  }
  ```

Fields

- **sig A {f: set X}**
- **sig B extends A {g: set Y}**
- means
  - B **in A**
  - f: A -> X
  - g: B -> Y



Set Multiplicities

- set : any number.
- one: exactly one.
- lone: zero or one.
- some: one or more.
  - sig Man extends Person { Wife: lone Woman}
  - sig Woman extends Person {Husband: lone Man}
  - sig Object{}
    sig Dir extends Object {
      entries: set Object,
      parent: lone Dir
      }

Facts and Assertions

- fact:
  - Constraints that are assumed always to hold (used to restrict space of possible counterexamples)
  - A Model can have any number of facts

    sig A {}

    fact { all z: A | F }
- assert:
  - Constraints that are intended to follow from the facts of the model
  - Can be checked using the check command

assert, check

- **fact F {…}**
- **assert A {…}**
- **check A**
- means
  - fact: assume constraint F holds
  - assert: believe that A follows from F
  - check: find an instance that satisfies F *and not A*

Ref: Alloy in 90 minutes. http://people.csail.mit.edu/dnj/talks/re05-tutorial/alloy-90.pdf

Quantifiers

- all x: e | F
- some x: e | F
- no x: e | F
- lone x: e | F
- one x: e | F
  - fact EntriesSameName {all e1, e2 : entries | e1.name = e2.name => e1 = e2}
  - fact nametoaddress { all n:Name| lone d: Address| d in n.address}
  - fact {no p: Person | p in p.^(mother + father) }
  - fact FourPassengersPerCar { all c:Car | #{c.contents } <= 4 }
  - fact OneVehiclePerPassenger { all p:Passenger | one v:Vehicle | p in v.contents }

Examples of Asserts

- assert NoSelfFather {   no m: Man | m = m.father}
- check NoSelfFather

- assert FileInDir {all f: File | some d: Dir | f in d.contents}
- check FileInDir

- assert RootTop {no o: Object | Root in o.contents}
- check RootTop

Constants and Operators

- The language of relations has its own constants and operators
- Constants
  - none = empty set
  - univ = universal set
  - iden = identity

Constants and Operators

- Name = {(N0), (N1)}
  Addr = {(D0), (D1)}
- none={}
  univ = {(N0), (N1), (D0), (D1)}
  iden = {(N0,N0), (N1,N1), (D0,D0), (D1,D1)}

- Operators fall into Two Categories
  – Set
  – Relational

Set Operators

- + : union
  - sig Vehicle {} {Vehicle = Car + Truck}
- & : intersection
  - fact DisjSubtrees { all t1, t2:Tree | (t1.^children) & (t2.^children) = none }
- - : difference
  - fact dirinothers { all d: Directory - Root | some contents.d }
- in : subset
  - Sibling = (brother + sister)
  - sister in sibling
- = : equality

Relational operators

- . : dot (Join)
  - {(N0), (A0)} . {(A0), (D0)} = {(N0), (D0)}
- -> : arrow (product)
  - s -> t is their cartesian product
  - r: s -> t says r maps atoms in s to atoms in t
- ^ : transitive closure
  - fact DisjSubtrees { all t1, t2:Tree | (t1.^children) & (t2.^children) = none }
  - fact {no p: Person | p in p.^(mother + father) }
- * : reflexive-transitive closure
  - fact {Object in Root.*contents}
- ~ : transpose
  - Takes its mirror image s.~r = r.s (image of s navigating backwards through rel r)

Relational Operators

- ~ : transpose (continued)
  - fact dirinothers{ all d: Directory - Root | some d.~contents }
  - fact dirinothers { all d: Directory - Root | some contents.d }
- [] : box (join)
  - Semantically identical to join, takes arguments in different order. Expressions: e1[e2] = e2.e1
- <: :domain restriction
  - Contains those tuples of r that start with an element in s.
- :> : range restriction
  - Contains the tuples of r that end with an element in s
- ++: Override
  - Override p ++ q (just like union), except that tuples of q can replace tuples of p rather than augmenting them.

Relational Operators

- Alias = {(N0), (N1)}
- Addr = {(A0)}
- address = {(N0, N1), (N1, N2), (N2, A0)}

- Domain restriction
- Alias <: address = {(N0, N1), (N1, N2)}

- Range restriction:
- address :> Addr = {(N2, A0)}

- workAddress = {(N0, N1), (N1, A0)}
- address ++ workAddress = {(N0, N1), (N1, A0), (N2, A0)}

Logical operators

- ! : negation
- && : conjunction (and)
- || : disjunction (OR)
- => : implication
- else : alternative
- <=> : bi-implication (IFF)

Logic Cardinalities

- = *equals*
- < *less than*
- > *greater than*
- =< *less than or equal to*
- >= *greater than or equal to*
- #r *number of tuples in r*
- 0,1,... *integer literal*
- + *plus*
- - *minus*

- all b: Bag | #b.marbles =< 3
  all bags have 3 or less marbles
- fact FourPassengersPerCar { all c:Car | #{c.contents} <= 4 }

Functions and Predicates

- A function is a named expression
- Zero or more declarations for arguments
- fun grandpas [p: Person]: set Person {

  p.(mother + father).father

  }
- fun colorSequence: Color -> Color {

  Color <: iden + Red->Green + Green->Yellow + Yellow->Red

  }

Predicates and functions

- A predicate is a named constraint
- Zero or more declarations for arguments
- Can be used to represent an operation
- Only holds when invoked (unlike fact)
  - ```
    sig Name, Addr {}
    sig Book { addr: Name -> Addr }
    pred add (b, b': Book, n: Name, a: Addr) {
      b'.addr = b.addr + n->a
    }
    ```
  - pred Above(m, n: Man) {m.floor = n.ceiling}
    Man m is “above” Man n if m's floor is n's ceiling
    ```

fact, pred, run

- **fact F {…}**
- **pred P () {…}**
- **run P**
- means
  - – fact: assume constraint F holds
  - – pred: define constraint P
  - – run: find an instance that satisfies P *and F*

Ref: Alloy in 90 minutes. http://people.csail.mit.edu/dnj/talks/re05-tutorial/alloy-90.pdf

Analyzing the model

- Write a command to analyze a model
- A run command searches for an instance of a predicate
  - run ownGrandpa
- A check command searches for a counter example of an assertion
  - check NoSelfFather
- You may give a scope that bounds the size of instances.
  - run solve for 5 State

- Demo The File System.