

Métodos Formais
2022.2

Introduction to Alloy

Áreas de Teoria e de Linguagens de Programação DCC/UFMG

Outline

- Introduction to basic Alloy constructs using a simple example of a static model
 - How to define *sets*, *subsets*, *relations with multiplicity constraints*
 - How to use Alloy's *quantifiers* and *predicate* forms
- Basic use of the Alloy Analyzer 6
 - *Loading, running, and analyzing* a simple Alloy specification
 - Adjusting basic *tool parameters*
 - Using the *visualization* tool to view instances of models

Why was Alloy created?

- *Lightweight*

- Small and easy to use
- capable of expressing common properties tersely and naturally

- *Precise*

- having a simple and uniform mathematical semantics

- *Tractable*

- amenable to efficient and fully automatic semantic analysis
 - within scope limits

What is Alloy used for?

- A textual modeling language aimed at expressing *structural* and *behavioral* properties of software systems
- Not meant for modeling code architecture
- But an Alloy specification can be closely related to an OO implementation

Example applications

- The Alloy 4 distribution comes with several sample models to illustrate the use of Alloy's constructs
- Examples
 - Specification of a distributed spanning tree
 - Model of a generic file system
 - Model of a generic file synchronizer
 - Tower of Hanoi model
 - ...

In summary

- Alloy is general enough that it can model
 - any domain of individuals
 - relations between them
- We will start with a few simple examples
 - Not necessarily about software

Example: Family structure

We want to:

- Model *parent/child relationships* as primitive relations
- Model *spousal relationships* as primitive relations
- Model relationships such as *siblings* as derived relations
- Enforce *biological constraints* via first-order predicates (e.g., people have only one mother)
- Enforce *social constraints* via first-order predicates (e.g., a wife isn't a sibling)
- Confirm or refute the existence of certain *derived relationships* (e.g., no one has a wife with whom he shares a parent)

Example: adressBook

An *address book* for an email client that maintains a mapping from *names* to *addresses*

FriendBook
Ted -> ted@gmail.com
Ryan -> ryan@hotmail.com

WorkBook
Pilard -> lpilard@ufmg.br
Ryan -> ryan@ufmb.br

Atoms and Relations

- In Alloy, everything is built from *atoms* and *relations*
- An *atom* is a primitive entity that is
 - *indivisible*: it cannot be broken down into smaller parts
 - *immutable*: its properties do not change over time
 - *uninterpreted*: it does not have any built in property (the way numbers do for example)
- A *relation* is a structure that *relates atoms*. It is a set of *tuples*, each tuple being a sequence of atoms

Atoms and Relations: Examples

- *Unary relations*: a set of names, a set of addresses and a set of books

Name = {(N0), (N1), (N2)}

Addr = {(D0), (D1)}

Book = {(B0), (B1)}

- A *binary relation* from names to addresses

address = {(N0,D0),(N1,D1)}

- A *ternary relation* from books to names to addresses

address = {(B0,N0,D0),(B0,N1,D1),(B1,N1,D2)}

Relations

- *Size* of a relation: the number of tuples in the relation
- *Arity* of a relation: the number of atoms in each tuple of the relation
 - relations with arity 1, 2, and 3 are said to be unary, binary, and ternary relations
- Examples.
 - relation of arity 1 and size 1:
$$\text{myName} = \{(N0)\}$$
 - relation of arity 2 and size 3:
$$\text{address} = \{(N0,D0),(N1,D1),(N2,D1)\}$$

Main components of Alloy models

- Signatures and Fields
- Predicates and Functions
- Facts
- Assertions
- Commands and scopes

Signatures and Fields

- Signatures

- Describe classes of entities we want to reason about
- Sets defined in signatures are fixed (dynamic aspects can be modeled by time-dependent relations)

- Fields

- Define relations between signatures

- Simple constraints

- Multiplicities on signatures
- Multiplicities on relations

Signatures

- A signature introduces a set of atoms

- The declaration

sig A {}

introduces a set named A

- A set can be introduced as an extension of another; thus

sig A1 **extends** A {}

introduces a set A1 that is a *subset* of A

Signatures

```
sig A {}  
sig B {}  
sig A1 extends A {}  
sig A2 extends A {}
```

- A1 and A2 are *extensions* of A
- A signature declared independently of any other one is a *top-level signature*, e.g., A and B
- Extensions of the same signature are *mutually disjoint*, as are top-level signatures

Signatures

```
sig A {}  
sig B {}  
sig A1 extends A {}  
sig A2 extends A {}
```

- A1 and A2 are *extensions* of A
- A signature declared independently of any other one is a *top-level signature*, e.g., A and B
- Extensions of the same signature are *mutually disjoint*, as are top-level signatures

Signatures

```
abstract sig A {}  
sig B {}  
sig A1 extends A {}  
sig A2 extends A {}
```

- A signature can be introduced as a *subset* of another

```
sig A3 in A {}  
sig A2 extends A {}
```

- An *abstract signature* has no elements except those belonging to its extensions or subsets
- All extensions of an abstract signature A form a *partition* of A

Fields

- *Relations* are declared as *fields* of signatures
- Writing

sig A { f : e }

introduces a relation f of type $A \times e$, where e is an expression denoting a product of signatures

- Examples: (with signatures A, B, C)
 - Binary relation:

sig A { f1 : B }

where f1 is a subset of $A \times B$

- Ternary relation:

sig A { f2 : B \rightarrow C }

where f2 is a subset of $A \times B \times C$

Example signatures and fields

A family structure:

```
abstract sig Person {  
    children: Person ,  
    siblings: Person  
}
```

```
sig Man, Woman extends Person {}
```

```
sig Married in Person {  
    spouse: Married  
}
```

Example: family structure

A family structure:

```
abstract sig Person {}  
sig Man extends Person {}  
sig Woman extends Person {}  
sig Married in Person {}
```

Example: family structure

A family structure:

```
abstract sig Person {  
    siblings: Person  
}  
sig Man extends Person {}  
sig Woman extends Person {}  
sig Married in Person {}
```

Example: family structure

A family structure:

```
abstract sig Person {  
    siblings: Person  
}  
sig Man extends Person {}  
sig Woman extends Person {}  
sig Married in Person {}
```

An example of an instance is

Person = {(P0), (P1)}

Man = {(P0), (P1)}

Married = {}

Woman = {}

siblings = {(P0,P1), (P1,P0)}

- *siblings* is a binary relation, i.e., a subset of Person x Person
- In the instance, P0 and P1 are siblings

Multiplicities

- Allow us to constrain the sizes of sets
 - A multiplicity keyword placed before a signature declaration constrains the number of elements in the signature's

`m sig A {}`

- We can also make multiplicities constraints on fields:

`sig A {f: m e}`

`sig A {f: e1 m -> n e2}`

- There are four multiplicities
 - set : any number
 - some : one or more
 - lone : zero or one
 - one : exactly one

Multiplicities: Examples

- Without multiplicity:

- A set of colors, each of which is red, yellow or green abstract

```
sig Color {}  
sig Red, Yellow, Green extends Color {}
```

- With multiplicity:

- An enumeration of colors

```
abstract sig Color {}  
one sig Red, Yellow, Green extends Color {}
```


Multiplicities: Examples

- A file system in which each directory contains any number of objects, and each alias points to exactly one object

```
abstract sig Object {}  
sig Directory extends Object {contents: set Object}  
sig File extends Object {}  
sig Alias in File {to: one Object}
```

- The *default multiplicity* is **one**, so:

```
sig A {f: e}  
sig A {f: one e}
```

are equivalent

Multiplicities: Examples

- A book maps names to addresses
 - There is at most one address per Name
 - An address is associated to at least one name

```
sig Name, Addr {}  
sig Book {  
  addr: Name some -> lone Addr  
}
```

Multiplicities: Examples

- A collection of weather forecasts, each of which has a field *weather* associating every city with exactly one weather condition

```
sig Forecast {weather: City  $\rightarrow$  one Weather}  
sig City {}  
abstract sig Weather {}  
one sig Rainy, Sunny, Cloudy extends Weather {}
```

- Instance:

```
City = {(BH), (Uberlandia)}  
Rainy = {(rainy)}  
Sunny = {(sunny)}  
Cloudy = {(cloudy)}  
Forecast = {(f1), (f2)}  
weather = { (f1, BH, rainy), (f1, Uberlandia, rainy),  
            (f2, BH, rainy), (f2, Uberlandia, sunny) }
```

Multiplicities and Binary Relations

sig $S \{f: \text{ lone } T\}$

- says that, for each element s of S , f maps s to *at most* a single value in T

Multiplicities and Binary Relations

sig $S \{f: \text{ lone } T\}$

- says that, for each element s of S , f maps s to *at most* a single value in T
 - Note this means that f is a *partial function*

Multiplicities and Binary Relations

sig $S \{f: \text{ lone } T\}$

- says that, for each element s of S , f maps s to *at most* a single value in T
 - Note this means that f is a *partial function*

- What if we had

sig $S \{f: \text{ one } T\}$

Multiplicities and Binary Relations

sig $S \{f: \text{ lone } T\}$

- says that, for each element s of S , f maps s to *at most* a single value in T
 - Note this means that f is a *partial function*

- What if we had

sig $S \{f: \text{ one } T\}$

- Defines a total function

Multiplicities and Ternary Relations

sig S {f: T \rightarrow **one** V}

Multiplicities and Ternary Relations

sig S {f: T \rightarrow **one** V}

- for *each element* s of S, over the triples that start with s: f maps each T-element to *exactly one* V-element

Multiplicities and Ternary Relations

sig S {f: T \rightarrow **one** V}

- for *each element* s of S, over the triples that start with s: f maps each T-element to *exactly one* V-element

sig S {f: T **lone** \rightarrow V}

Multiplicities and Ternary Relations

sig $S \{f: T \rightarrow \text{one } V\}$

- for *each element* s of S , over the triples that start with s : f maps each T -element to *exactly one* V -element

sig $S \{f: T \text{ lone} \rightarrow V\}$

- For *each element* s of S , over the triples that start with s : f maps *at most one* T -element to the same V -element

Multiplicities and Relations

- Other kinds of relational structures can be specified using multiplicities

- Examples

- **sig** S {f: **some** T} ... total relation
- **sig** S {f: **set** T} ... partial relation
- **sig** S {f: T **set** \rightarrow **set** V}
- **sig** S {f: T **one** \rightarrow V}
- ...

Cardinality constraints

- Multiplicities can also be applied to whole expressions denoting relations

- `some e` `e` is non-empty
- `no e` `e` is empty
- `lone e` `e` has at most one tuple
- `one e` `e` has exactly one tuple

Cardinality constraints

- Multiplicities can also be applied to whole expressions denoting relations

- `some e` `e` is non-empty
- `no e` `e` is empty
- `lone e` `e` has at most one tuple
- `one e` `e` has exactly one tuple

Example: family structure

- How would you use multiplicities to define the children relation?

Example: family structure

- How would you use multiplicities to define the children relation?

sig Person {children: **set** Person}

- Intuition: each person has zero or more children

Example: family structure

- How would you use multiplicities to define the children relation?

sig Person {children: **set** Person}

- Intuition: each person has zero or more children

- How would you use multiplicities to define the spouse relation?

Example: family structure

- How would you use multiplicities to define the children relation?

sig Person {children: **set** Person}

- Intuition: each person has zero or more children

- How would you use multiplicities to define the spouse relation?

sig Married {spouse: **one** Married}

- Intuition: each married person has exactly one spouse

Example: family structure

```
abstract sig Person {  
    children: set Person,  
    siblings: set Person  
}  
sig Man, Woman extends Person {}  
sig Married in Person {  
    spouse: one Married  
}
```

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
 - Specific for that run
- We can use conditions to encode *realism constraints* to e.g.,
 - Force generated models to include at least one married person, or one married man, etc.

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
- The 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 is 3 for each top-level signature
- AA executes only the first run command in a file

run Example

— 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}
```

Model weaknesses

- The model is underconstrained
 - It doesn't match our domain knowledge
 - Asymmetric marriage, self child/sibling, asymmetric siblings, multiple fathers...
 - We can add constraints to enrich the model
- Under-constrained models are common early on in the development process
 - AA gives us quick feedback on weaknesses in our model
 - We can incrementally add constraints until we are satisfied with it

Adding constraints

- We'd like to enforce the following constraints (concerning *biology*)
 - No person can be their own parent (or more generally, their own ancestor)
 - No person can have more than one father or mother
 - A person's siblings are those with the same parents
- We could also enforce the following *social* constraints
 - The spouse relation is symmetric
 - A man's wife cannot be one of his siblings

Acknowledgments

These notes are heavily based on notes from Matt Dwyer, John Hatcliff, Rod Howell, Laurence Pilard and Cesare Tinelli.