Politecnico
di Milano

Alloy – syntax and semantics

Video available here
https://web.microsoftstream.com/video/19e80b3f-530b-4cf0-a576-b36f6e493b48

Alloy: Syntax and Semantics

- Alloy is a language
  - It has a syntax *how do I write a right specification?*
  - It has a semantics *what does it mean?*
- In a programming language
  - Syntax defines correct programs (i.e., allowed programs)
  - Semantics defines the meaning of a program as its possible (many?) computations
- In Alloy
  - Syntax as usual...
  - Semantics defines the meaning of a specification as the collection of its models, i.e., of the worlds that make our Alloy description true

Alloy = logic + language + analysis

- Logic
  - first order logic + relational calculus
- Language
  - syntax for structuring specifications in the logic
- Analysis
  - shows bounded snapshots of the world that satisfy the specification
  - bounded exhaustive search for counterexample to a claimed property using SAT

Logic Elements

- Relations
- Constants
- Set Operators
- Join Operators
- Quantifiers and Boolean Operators
- Set and Relation Declaration
- If and let

Logic: relations of atoms

- Atoms are Alloy's primitive entities
  - indivisible, immutable, uninterpreted

- Relations associate atoms with one another
  - set of tuples, tuples are sequences of atoms

Logic: everything is a relation

- Sets are unary (1 column) relations

```
Name = {(N0),        Addr = {(A0),        Book = {(B0),
        (N1),                (A1),                (B1)}
        (N2)}                (A2)}
```

- Scalars are singleton sets

```
myName   = {(N1)}
yourName = {(N2)}
myBook   = {(B0)}
```

- Ternary relation

```
addr = {(B0, N0, A0),
        (B0, N1, A1),
        (B1, N1, A2),
        (B1, N2, A2)}
```

Relations

- Relation = set of ordered n-tuples of atoms
  - n is called the *arity* of the relation

- Relations in Alloy are typed
  - Determined by the declaration of the relation
  - Example: a relation with type
    `Person -> String`
    only contains pairs
    - whose first component is a Person
    - and whose second component is a String

Logic: constants

`none`	*empty set*
`univ`	*universal set*
`iden`	*identity relation*

```
Name = {(N0), (N1), (N2)}
Addr = {(A0), (A1)}

none = {}
univ = {(N0), (N1), (N2), (A0), (A1)}
iden = {(N0,N0), (N1,N1), (N2,N2), (A0,A0),
  (A1,A1)}
```

Logic: set operators

`+`	*union*
`&`	*intersection*
`-`	*difference*
`in`	*subset*
`=`	*equality*

```
Name  = {(N0), (N1), (N2)}
Alias = {(N1), (N2)}
Group = {(N0)}
RecentlyUsed = {(N0), (N2)}

Alias + Group = {(N0), (N1), (N2)}
Alias & RecentlyUsed  = {(N2)}
Name - RecentlyUsed   = {(N1)}
RecentlyUsed in Alias = false
RecentlyUsed in Name  = true
Name = Group + Alias  = true
```

```
greg = {(N0)}
rob = {(N1)}

greg + rob   = {(N0), (N1)}
greg = rob   = false
rob in none  = false
```

```
cacheAddr = {(N0, A0), (N1, A1)}
diskAddr = {(N0, A0), (N1, A2)}

cacheAddr + diskAddr = {(N0, A0), (N1, A1), (N1, A2)}
cacheAddr & diskAddr = {(N0, A0)}
cacheAddr = diskAddr = false
```

Logic: product operator

```
->      cross product
```

```
Name = {(N0), (N1)}
Addr = {(A0), (A1)}
Book = {(B0)}

Name->Addr = {(N0, A0), (N0, A1),
              (N1, A0), (N1, A1)}
Book->Name->Addr =
  {(B0, N0, A0), (B0, N0, A1),
   (B0, N1, A0), (B0, N1, A1)}
```

```
b  = {(B0)}
b' = {(B1)}
address  = {(N0, A0), (N1, A1)}
address' = {(N2, A2)}

b->b' = {(B0, B1)}

b->address + b'->address' =
  {(B0, N0, A0), (B0, N1, A1), (B1, N2, A2)}
```

Logic: relational composition dot join

	p		q		
p.q ≡	(a, b) (a, c) (b, d)	.	(a, d, c) (b, c, c) (c, c, c) (b, a, d)	=	(a, c, c) (a, a, d)

	x		f		
x.f ≡	(c)	.	(a, b) (b, d) (c, a) (d, a)	=	(a)

in db join the match of columns is by name, not by position
and the matching column is not dropped

Logic: join operators

`.`	*dot join*
`[]`	*box join*

```
  e1[e2] = e2.e1
a.b.c[d] = d.(a.b.c)
```

```
Book = {(B0)}
Name = {(N0), (N1), (N2)}
Addr = {(A0), (A1), (A2)}
Host = {(H0), (H1)}

myName = {(N1)}
myAddr = {(A0)}

address = {(B0, N0, A0), (B0, N1, A0), (B0, N2, A2)}
host = {(A0, H0), (A1, H1), (A2, H1)}

Book.address = {(N0, A0), (N1, A0), (N2, A2)}
Book.address[myName] = {(A0)}
Book.address.myName = {}

host[myAddr] = {(H0)}
address.host = {(B0, N0, H0), (B0, N1, H0), (B0, N2, H1)}
```

Logic: unary operators

~	*transpose*
^	*transitive closure*
*	*reflexive transitive closure*

apply only to binary relations

```
^r = r + r.r + r.r.r + …
*r = iden + ^r
```

```
Node = {(N0), (N1), (N2), (N3)}
next = {(N0, N1), (N1, N2), (N2, N3)}  a list

~next = {(N1, N0), (N2, N1), (N3, N2)}
^next = {(N0, N1), (N0, N2), (N0, N3),
         (N1, N2), (N1, N3),
         (N2, N3)}
*next = {(N0, N0), (N0, N1), (N0, N2), (N0, N3),
         (N1, N1), (N1, N2), (N1, N3),
         (N2, N2), (N2, N3), (N3, N3)}
```

```
first = {(N0)}
rest = {(N1), (N2), (N3)}

first.^next = rest
first.*next = Node
```

Logic: restriction and override

```
<:     domain restriction
:>     range restriction
++     override
```

Apply to any relations (normally binary)

```
p ++ q =
p - (domain[q] <: p) + q
```

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

address :> Addr   = {(N2, A0)}
Alias <: address = {(N0, N1), (N1, N2)}
address :> Alias = {(N0, N1)}

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

```
m' = m ++ (k -> v)
```

update map m with key-value pair (k, v)

Logic: restriction and override

- Examples of usage of these operators with non binary relations
  - `addr = {(B0, N1, A2), (B1, N1, A2), (B2, N0, A1), (B2, N1, A0)}`
  - `{B2} <: addr = {(B2, N0, A1), (B2, N1, A0)}`
  - `addr :> {A2} = {(B0, N1, A2), (B1, N1, A2)}`
  - `addr ++ {(B0, N1, A0)} = {(B0, N1, A0), (B1, N1, A2), (B2, N0, A1), (B2, N1, A0)}`
  - `addr ++ {(B0, N0, A0)} = {(B0, N0, A0), (B1, N1, A2), (B2, N0, A1), (B2, N1, A0)}`
  - `addr :> {N1} = {}`

Logic: boolean operators

`!`	`not`	*negation*
`&&`	`and`	*conjunction*
`||`	`or`	*disjunction*
`=>`	`implies`	*implication*
`,`	`else`	*alternative*
`<=>`	`iff`	*bi-implication*

four equivalent constraints:

```
F => G else H

F implies G else H

(F && G) || ((!F) && H)

(F and G) or ((not F) and H)
```

Logic: quantifiers

```
all x: e | F
all x: e1, y: e2 | F
all x, y: e | F
all disj x, y: e | F
```

`all`	*F holds for every x in e*
`some`	*F holds for at least one x in e*
`no`	*F holds for no x in e*
`lone`	*F holds for at most one x in e*
`one`	*F holds for exactly one x in e*

`some n: Name, a: Address | a in n.address`
some name maps to some address — address book not empty

`no n: Name | n in n.^address`
no name can be reached by lookups from itself — address book acyclic

`all n: Name | lone a: Address | a in n.address`
every name maps to at most one address — address book is functional

`all n: Name | no disj a, a': Address | (a + a') in n.address`
no name maps to two or more distinct addresses — same as above

Logic: relation declarations

multiplicity constraint

`x: m e`

`set`	*any number*	
`one`	*exactly one*	default
`lone`	*zero or one*	
`some`	*one or more*	

`RecentlyUsed: set Name`
RecentlyUsed is a subset of the set Name

`senderAddress: Addr`
senderAddress is a singleton subset of Addr

`senderName: lone Name`
senderName is either empty or a singleton subset of Name

`receiverAddresses: some Addr`
receiverAddresses is a nonempty subset of Addr

Logic: relation declarations

```
r: A m -> n B
```

```
(r: A m -> n B) <=>
   ((all a: A | n a.r) and (all b: B | m r.b))
```

`workAddress: Name -> lone Addr`
each name refers to at most one work address

`homeAddress: Name -> one Addr`
each name refers to exactly one home address

`members: Group lone -> some Addr`
address belongs to at most one group
and group contains at least one address

Logic: set definitions

```
{x1: e1, x2: e2, ..., xn: en | F}
```

```
{n: Name | no n.^address & Addr}
```
set of names that don't resolve to any actual addresses

```
{n: Name, a: Address | n -> a in ^address}
```
binary relation mapping names to reachable addresses

Logic: if and let

```
f implies e1 else e2
let x = e | formula
let x = e | expression
```

four equivalent constraints:

```
all n: Name |
  (some n.workAddress
     implies n.address = n.workAddress
     else n.address = n.homeAddress)

all n: Name |
  let w = n.workAddress, a = n.address |
    (some w implies a = w else a = n.homeAddress)

all n: Name |
  let w = n.workAddress |
    n.address = (some w implies w else n.homeAddress)

all n: Name |
  n.address = (let w = n.workAddress |
    (some w implies w else n.homeAddress))
```

Logic: cardinalities

`#r`	*number of tuples in r*
`0,1,...`	*integer literal*
`+`	*plus*
`-`	*minus*

`=`	*equals*
`<`	*less than*
`>`	*greater than*
`=<`	*less than or equal to*
`>=`	*greater than or equal to*

```
sum x: e | ie
```
sum of integer expression ie for all singletons x drawn from e

```
all b: Bag | #b.marbles =< 3
```
all bags have 3 or less marbles

```
#Marble = sum b: Bag | #b.marbles
```
the sum of the marbles across all bags equals the total number of marbles

Alloy Language (1)

- An Alloy document is a “source code” unit
- It may contain:
  - **Signatures**: define types and relationships
  - **Facts**: properties of models (constraints!)
  - **Predicates/functions**: reusable expressions
  - **Assertions**: properties we want to check
  - **Commands**: instruct the Alloy Analyzer which assertions to check, and how

A predicate is run to find a world that satisfies it

An assertion is checked to find a counterexample

Alloy Language (2)

- Signatures, predicates, facts and functions tell how correct worlds (models) are made
  - When the Alloy analyzer tries to build a model, it must comply with them
- Assertions and commands tell which kind of checks must be performed over these worlds
  - E.g., “find, among all the models, one that violates this assertion”
- Of course, the Analyzer cannot check all the (usually infinite) models of a specification
  - So you must also tell the Analyzer how to limit the search