MFES – Métodos Formais em Engenharia de Software

Alloy

Ana Paiva
apaiva@fe.up.pt www.fe.up.pt/~apaiva

Modeling with Alloy

- **Statics**: Exploring states
  - Signatures
  - Facts
- **Dynamics**: Adding operations
  - Operations
  - Functions

Alloy

- A model in Alloy is an object model: a set of atoms (objects) connected through relations.
- Running a predicate:
  - Checks satisfability: tries to find a model that satisfies the predicate
  - If no model is found, the specification may be inconsistent.
  - Interactively returns all models that satisfy the predicate (if any) within a limited scope.
- Checking an assertion:
  - Checks validity: tries to find a counter-example that violates the assertion
  - If no counter-example is found, the specification may be valid.

Checking satisfiability

```
pred Name {
        <formulas>
}

run Name for <scope>
```

Checking validity

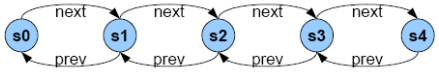
```
assert Name {
        <formulas>
}

check Name for <scope>
```


Ordering module

Establishes linear ordering over atoms of signature S

```
open util/ordering[S]
```

S = s0 + s1 +s2+ s3 + s4




```
first() = s0
last() = s4
next(s2) = s3
prev(s2) = s1
nexts(s2) = s3 + s4
prevs(s2) = s0 + s1
```

```
lt(s1, s2) = true
lt(s1, s1) = false
gt(s1, s2) = false
lte(s0, s3) = true
lte(s0, s0) = true
gte(s2, s4) = false
```


Modeling dynamics

- Representing state with signatures:
  sig State { ... }
- Representing initial conditions as predicates:
  pred init[s : State] { ... }
- Representing operations as predicates:
  pred op_1[s, s': State, a: Data_1, ...] { ... }
  pred op_2[s, s': State, a: Data_2, ...] { ... }
  ...
  pred op_n[s, s': State, a: Data_n, ...] { ... }

Modeling dynamics

- Pre- and post-conditions:

```
pred op_1[s, s':State, ...] {
        precondition[s]
        postcondition[s,s']
}
```

- Generating traces (imposing an ordering on states)

```
open util/ordering[State]
...
fact traces {
     init[first]
     all s : State - last |
          let s' = s.next | op_1[s,s',...] or op_2[s,s',...] or ... or op_n[s,s',...]
}
```

Modeling dynamics

- Representing invariants as predicates or facts:

```
fact invariant1 { ... }
pred invariant2[s : State,...] { ... }
```

- Checking whether an invariant is preserved by an operation:

```
assert Op1PreservesInvariant2 {
    all s, s': State, a: Data_1 |
        invariant2[s] and op_1[s,s',a] implies invariant2[s']
}
```

- Checking that the invariant is established by an initial state

```
assert initEstablishesInvariant {
    all s: State | init[s] implies invariant2[s]
}
```


Some examples

The shepherd, the wolf, the sheep and cabbage

- A shepherd, accompanied by a sheep, a wolf and a huge cabbage, close to the edge of a river he must cross. Faced him a boat so small that ...
- ...at a time he can only bring himself and one of his companions. But if you let the sheep and cabbage alone in a room, the cabbage will be devoured by the sheep...
- How will you organize the trips of the shepherd in order to be on the other side with his three companions?

The shepherd, the wolf, the sheep and cabbage (statics)

```
// the solution for this problem is a sequence of states
  open util/ordering[State]

// There are some things that eat other things
  abstract sig Thing{
        eat : set Thing
  }

// The things that are important in this example are wolf, sheep
  // and cabbage
  one sig Wolf, Sheep, Cabbage extends Thing {}
```

The shepherd, the wolf, the sheep and cabbage (statics)

```
// fact: The wolf eat sheeps and the sheeps eat cabbages
   fact {  eat = (Wolf -> Sheep) + (Sheep -> Cabbage)  }
// the river has two edges: right and left
   abstract sig Edge{}
   one sig Right, Left extends Edge {}
// the state of the system keeps the edge where the shepherd is
   // and the edge where the other things are
   sig State {
         shepherd: Edge,
         local : Thing -> one Edge
   }
```


The shepherd, the wolf, the sheep and cabbage (statics)

```
// the edge where the shepherd is not, cannot have things that
   // eat each other
fact {
   all s : State, x,y : (s.local).(Edge - s.shepherd) | x not in y.eat
}
// Satisfability ?
run {} for 3 but 5 State
```


The shepherd, the wolf, the sheep and cabbage (dynamics)

```
// The shepherd changes edge alone
pred alone[s, s' : State] {
   s'.local = s.local
   s'.shepherd != s.shepherd
}
run alone for 3
// o barqueiro muda de margem acompanhado
pred withcompany[s, s' : State] {
   one x : Thing {   s.shepherd = s.local[x]
                     s'.shepherd != s.shepherd
                     s'.local[x] = s'.shepherd
                     all y : Thing - x | s'.local[y] = s.local[y]      }
}
run withcompany for 3 but 2 State
```


The shepherd, the wolf, the sheep and cabbage (dynamics)

```
// In the beginning the things are in the same edge
// the system changes states either when the shepherd changes edge (alone or
   // with company)
fact { //this fact could/"should" be expressed as a predicate
   lone Thing.(first.local)
   all s : State, s' : s.next | alone[s,s'] or withcompany[s,s']
}
//How to obtain the solution (sequence of steps performed to change the edge
   //of all things)?
// There is no state where the edge is different from the edge in the beginning
   // so it is impossible to change edge
assert ThereIsNoSolution {
   no s : State | Thing.(s.local) = Edge - Thing.(first.local)
}
check ThereIsNoSolution for 3 but 8 State
```

The shepherd, the wolf, the sheep and cabbage