Alloy

30/11/16

Kalev Alpernas

Lets Model a Family Tree

Lets Model a Family Tree

- First, we need to define a domain:

```
sig Person {}
```

Lets Model a Family Tree

- First, we need to define a domain:

```
sig Person {}
```

- Need parents to define a family tree:

```
sig Person {
  father: lone Person,
  mother: lone Person
}
```

Lets Model a Family Tree

- First, we need to define a domain:

```
sig Person {}
```

- Need parents to define a family tree:

```
sig Person {
  father: lone Person,
  mother: lone Person
}
```

- Alloy gives us some weird families.

Lets add constraints

- Father and mother are different:

```
fact { all p: Person | p.father != p.mother }
```

Lets add constraints

- Father and mother are different:

```
fact { all p: Person | p.father != p.mother }
```

- A person can’t be their father or mother:

```
fact { all p: Person | p.father != p and
                       p.mother != p }
```

Lets add constraints

- Father and mother are different:

```
fact { all p: Person | p.father != p.mother }
```

- A person can't be their father or mother:

```
fact { all p: Person | p.father != p and
                       p.mother != p }
```

- A little awkward, defining each constraint for both father and mother
  - Parent makes more sense

Predicates

- Lets define a new predicate:

```
pred parent(p1: Person, p2:Person) {
   p1 = p2.father or
   p1 = p2.mother
}
```

Predicates

- Lets define a new predicate:

```
pred parent(p1: Person, p2:Person) {
   p1 = p2.father or
   p1 = p2.mother
}
```

- And use it in some constraints:

```
fact { all p: Person | !parent[p , p] }
fact { all p1, p2: Person | ! (parent[p1 , p2] and
                               parent[p2, p1] ) }
```

Predicates

- Lets define a new predicate:

```
pred parent(p1: Person, p2:Person) {
   p1 = p2.father or
   p1 = p2.mother
}
```

- And use it in some constraints:

```
fact { all p: Person | !parent[p , p] }
fact { all p1, p2: Person | ! (parent[p1 , p2] and
                               parent[p2, p1] ) }
```

- Can also get an instance that satisfies the predicate
  - `Run parent for 5`

Functions

- A person shouldn’t be able to be their own grandparent as well

Functions

- A person shouldn’t be able to be their own grandparent as well
- Lets define a grandparents function:

```
fun grands (p: Person): set Person {
   p.(father+mother).(father+mother)
}
```

Functions

- A person shouldn’t be able to be their own grandparent as well
- Lets define a grandparents function:

```
fun grands (p: Person): set Person {
   p.(father+mother).(father+mother)
}
```

- And use it in a constraint:

```
fact { no p: Person | p in grands[p] }
```

Lets check something

- Siblings shouldn’t be each others parents

Lets check something

- Siblings shouldn’t be each others parents
- Lets define an assertion:

```
assert sibsNotParents {
   no p1, p2: Person |
        some p3: Person |
             parent[p3, p1] and
             parent[p3, p2] and
             parent[p1, p2] and p1 != p2
}
```

Lets check something

- Siblings shouldn’t be each others parents
- Lets define an assertion:

```
assert sibsNotParents {
   no p1, p2: Person |
        some p3: Person |
             parent[p3, p1] and
             parent[p3, p2] and
             parent[p1, p2] and p1 != p2
}
```

- And check our assertion:

```
check sibsNotParents for 5
```

Quantifiers

- `some` – same as ∃, one or more elements (match the property)
- `all` – same as ∀, all elements (match the property)
- `no` – same as ¬∃, there exist no elements (match the property)
- `one` – exactly one element (matches the property)
- `lone` – zero or one element (match the property)

Transitive Closure

- `p.mother`
  `p.mother.mother`
  `p.mother.mother.mother`
  …
- `p.^mother` – a set of persons, closed under the *mother* operation

- `p.*mother` – a set of persons, closed under the *mother* operation, includes `p`