CS:5810 Formal Methods in Software Engineering

Dynamic Models in Alloy

Overview

- Basics of dynamic models
  - Modeling a system's states and state transitions
  - Modeling operations causing transitions
- Simple example of operations

Static Models

- So far we've used Alloy to define the allowable values of state components
  - values of sets
  - values of relations
- A model instance is a set of state component values that
  - Satisfies the constraints defined by multiplicities, fact, "realism" conditions, …

Static Models

```
Person = {Matt, Sue}
Man = {Matt}
Woman = {Sue}
Married = {}
spouse = {}
children = {}
siblings = {}
```

```
Person = {Matt, Sue}
Man = {Matt}
Woman = {Sue}
Married = {Matt, Sue}
spouse = {(Matt,Sue), (Sue,Matt)}
children = {}
siblings = {}
```

```
Person = {Matt, Sue, Sean}
Man = {Matt, Sean}
Woman = {Sue}
Married = {Matt, Sue}
spouse = {(Matt,Sue), (Sue,Matt)}
children = {(Matt,Sean), (Sue,Sean)}
siblings = {}
```

Dynamic Models

- Static models allow us to describe the legal states of a dynamic system
- We also want to be able to describe the legal transitions between states

  E.g.
  - To get married one must be alive and not currently married
  - One must be alive to be able to die
  - A person becomes someone's child after birth

Example

Family Model

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

sig Man, Woman extends Person {}

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

State Transitions

- **Two people get married**
  - At time t, spouse = {}
  - At time t', spouse = {(Matt, Sue), (Sue,Matt)}

⇒ We add the notion of time in the relation spouse

```
Person = {Matt,Sue}
Man = {Matt}
Woman = {Sue}
Married = {}
spouse = {}
children = {}
siblings = {}
```

Time t

→

```
Person = {Matt, Sue}
Man = {Matt}
Woman = {Sue}
Married = {Matt, Sue}
spouse = {(Matt, Sue), (Sue, Matt)}
children = {}
siblings = {}
```

Time t'

Modeling State Transitions

- Alloy has no predefined notion of state transition
- However, there are several ways to model dynamic aspects of a system in Alloy
- A general and relatively simple way is to:
  1. introduce a `Time` signature expressing time
  2. add a time component to each relation that changes over time

Family Model Signatures

```
abstract sig Person {
      children: set Person,
      siblings: set Person set
}
sig Man, Woman extends Person {}

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

Family Model Signatures with Time

```
sig Time {}

abstract sig Person {
      children: Person set -> Time,
      siblings: Person set -> Time
}
sig Man, Woman extends Person {}

sig Married in Person {
      spouse: Married one -> Time
}
```

Transitions

- **Two people get married**
  - At time t, `Married` = {}
  - At time t', `Married` = {Matt, Sue}
  - Actually, we can't have a time-dependent signature such as Married because signatures are not time dependent.

```
Person = {Matt,Sue}
Man = {Matt}
Woman = {Sue}
Married = {}
spouse = {}
children = {}
siblings = {}
```

Time t

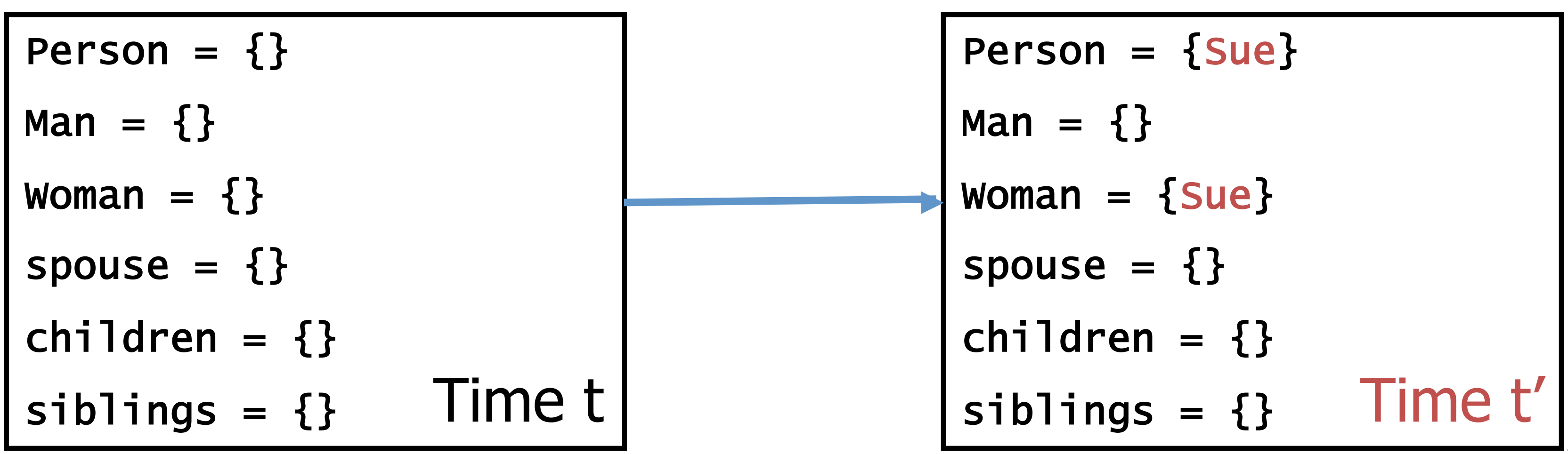
→

```
Person = {Matt, Sue}
Man = {Matt}
Woman = {Sue}
Married = {Matt, Sue}
spouse = {(Matt, Sue), (Sue, Matt)}
children = {}
siblings = {}
```

Time t'

Transitions

- **A person is born**
  - At time t, `Person` = {}
  - At time t’, `Person` = {Sue}
  - We cannot add the notion being born to the signature Person because signatures are not time dependent



Signatures are Static

```
abstract sig Person {
  children: Person set -> Time,
  siblings: Person set -> Time,
  spouse: Person lone -> Time
}
sig Man, Woman extends Person {}

sig Married in Person {
      spouse: Married one -> Time
}
```

Signatures are Static

```
abstract sig Person {
  children: Person set -> Time,
  siblings: Person set -> Time,
  spouse: Person lone -> Time
  alive: set Time
}

sig Man, Woman extends Person {}
```

Revising Constraints

```
abstract sig Person {
  children: Person set -> Time,
  siblings: Person set -> Time,
  spouse: Person lone -> Time,
  alive: set Time

}
sig Man, Woman extends Person {}
fun parents[] : Person->Person {~children}
```

Revising Constraints

```
abstract sig Person {
  children: Person set -> Time,
  siblings: Person set -> Time,
  spouse: Person lone -> Time,
  alive: set Time
  parents: Person set -> Time
}
sig Man, Woman extends Person {}
fun parents[] : Person->Person {~children}
fact parentsDef {
  all t: Time | parents.t = ~(children.t)
}
```

Revising Constraints

```
-- Time-dependent parents relation
fact parentsDef {
  all t: Time | parents.t = ~(children.t)
}

-- Two persons are blood relatives iff
-- they have a common ancestor
pred BloodRelatives [p, q: Person, t: Time]
{
  some p.*(parents.t) & q.*(parents.t)
}
```

Revising *Static* Constraints

```
-- People cannot be their own ancestors
all t: Time | no p: Person |
  p in p.^(parents.t)

-- No one can have more than one father
-- or mother
all t: Time | all p: Person |
  lone (p.parents.t & Man)
  and
  lone (p.parents.t & Woman)

...
```

Revising *Static* Constraints

```
-- A person p's siblings are those people, other
-- than p, with the same parents as p
all t: Time | all p: Person |
  p.siblings.t =
  { q: Person - p | some q.parents.t and
                    p.parents.t = q.parents.t }

-- Each married man (woman) has a wife (husband)
all t: Time | all p: Person |
  let s = p.spouse.t |
   (p in Man implies s in Woman) and
   (p in Woman implies s in Man)
```

Revising *Static* Constraints

```
-- A spouse can't be a sibling
all t: Time | no p: Person |
  some p.spouse.t and
  p.spouse.t in p.siblings.t


-- People can't be married to a blood relative
  all t: Time | no p: Person |
    let s = p.spouse.t |
      some s and
      BloodRelatives[p, s, t]
```

Revising *Static* Constraints

```
-- a person can't have children with
-- a blood relative
all t: Time | all p, q: Person |
  (some (p.children.t & q.children.t) and
  p != q)
  implies
  not BloodRelatives[p, q, t]

-- the spouse relation is symmetric
all t: Time |
  spouse.t = ~(spouse.t)
```

Exercises

- Load `family-6.als`
- Execute it
- Analyze the model
- Look at the generated instance
- Does it look correct?
- What, if anything, would you change about it?

Transitions

A person is born from parents

- Add to `alive` relation
- Modify `children`/`parents` relations

```
Person = {Matt, Sue, Sean}
Man = {Matt, Sean}
Woman = {Sue}
spouse = {(Matt,Sue), (Sue,Matt)}
children = {}
siblings = {}
alive = {Matt, Sue}
```

↓

```
Person = {Matt, Sue, Sean}
Man = {Matt, Sean}
Woman = {Sue}
spouse = {(Matt,Sue), (Sue,Matt)}
children = {(Matt,Sean), (Sue,Sean)}
siblings = {}
alive = {Matt, Sue, Sean}
```

State Sequences

```
Person = {Matt, Sue, Sean}
Man = {Matt, Sean}
Woman = {Sue}
spouse = {}
children = {}
siblings = {}
alive = {Sue}
```

```
Person = {Matt, Sue, Sean}
Man = {Matt, Sean}
Woman = {Sue}
spouse = {(Matt,Sue), (Sue,Matt)}
children = {}
siblings = {}
alive = {Sue, Matt}
```

```
Person = {Matt, Sue, Sean}
Man = {Matt, Sean}
Woman = {Sue}
spouse = {}
children = {}
siblings = {}
alive = {}
```

```
Person = {Matt, Sue, Sean}
Man = {Matt, Sean}
Woman = {Sue}
spouse = {(Matt,Sue), (Sue,Matt)}
children = {(Matt,Sean), (Sue,Sean)}
siblings = {}
alive = {Sue, Matt, Sean}
```

Expressing Transitions in Alloy

- A transition can be thought of as caused by the application of an operator to the current state

- An operator can be modeled as predicates between two states:
  1. the state right before the transition and
  2. the state right after it

- We define it as predicate with (at least) two formal parameters: `t, t': Time`

- Constraints over time `t` (resp., `t'`) model the state right before (resp., after) the transition

Expressing Transitions in Alloy

- Pre-condition constraints
  - Describe the states to which the transition applies
- Post-condition constraints
  - Describes the effects of the transition in generating the next state
- Frame-condition constraints
  - Describes what does not change between pre-state and post-state of a transition

Distinguishing the pre-, post- and frame-conditions in comments provides useful documentation

Example: Marriage

```
pred getMarried [m: Man, w: Woman, t,t': Time] {
-- preconditions
   -- m and w must be alive
   m+w in alive.t
   -- neither one is married
   no (m+w).spouse.t
   -- they are not be blood relatives
   not BloodRelatives[m, w, t]
-- post-conditions
   -- w is m’s wife
   m.spouse.t' = w
   -- m is w’s husband
   w.spouse.t' = m
-- frame conditions      ??
}
```

Frame Condition

How is each relation touched by marriage?

- 5 relations :
  - `children`, `parents`, `siblings`
  - `spouse`
  - `alive`
- `parents` and `siblings` relations are defined in terms of the `children` relation
- Thus, the frame condition has only to consider `children`, `spouse` and `alive` relations

Frame Condition Predicates

```
pred noChildrenChangeExcept [ps: set Person
                             t,t': Time] {
  all p: Person - ps |
    p.children.t' = p.children.t
}

pred noSpouseChangeExcept [ps: set Person
                           t,t': Time] {
  all p: Person - ps |
    p.spouse.t' = p.spouse.t
}

pred noAliveChange [t,t': Time] {
  alive.t' = alive.t
}
```

Example: Marriage

```
pred getMarried[m: Man, w: Woman, t,t': Time]
{
-- preconditions
   m+w in alive.t
   no (m+w).spouse.t
   not BloodRelatives[m, w, t]
-- post-conditions
   m.spouse.t' = w
-- frame conditions
  noChildrenChangeExcept[none, t, t’]
   noSpouseChangeExcept[m+w, t, t’]
   noAliveChange[t, t’]
}
```

Instance of Marriage

```
open ordering [Time] as T
…

pred marriageInstance {
  some t: Time |
  some m: Man | some w: Woman |
      getMarried[m, w, t, T/next[t] ]
}
run { marriageInstance }
```

Example: Birth from Parents

```
pred isBornFromParents [p: Person, m,w: Person,
                        t,t': Time] {
  -- Pre-condition
     m+w in alive.t
     p !in alive.t

  -- Post-condition and frame condition
     alive.t' = alive.t + p
     m.children.t' = m.children.t + p
     w.children.t' = w.children.t + p

  -- Frame condition
     noChildrenChangeExcept[m+w, t, t']
     noSpouseChangeExcept[none, t, t']
}
```

Instance of Birth

```
pred birthInstance {
  some t: Time |
  some p1, p2, p3: Person |
    isBornFromParents[p1, p2, p3, t, T/next[t]]
}

run { birthInstance }
```

Example: Death

```
pred dies [p: Person, t,t': Time] {
   -- Pre-condition
      p in alive.t

   -- Post-condition
      no p.spouse.t'

  -- Post-condition and frame condition
      alive.t' = alive.t - p
      all s: p.spouse.t |
        s.spouse.t' = s.spouse.t - p

   -- Frame condition
      noChildrenChangeExcept[none, t, t']
      noSpouseChangeExcept[p + p.spouse.t, t, t']

}
```

Instance of Death

```
pred deathInstance {
  some t: Time|
  some p: Person |
    dies[p, t, T/next[t]]
}

run { deathInstance }
```

Specifying Transition Systems

- A transition system can be defined as a set of executions:

  sequences of time steps generated by the operators

- In our example, for every execution:
  - The first time step satisfies some initialization condition
  - Each pair of consecutive steps are related by
    - a birth operation, or
    - a death operation, or
    - a marriage operation

Initial State Specification

`init` specifies constraints on the initial state

```
pred init [t: Time] {
  no children.t
  no spouse.t
  #alive.t > 2
  #Person > #alive.t
}
```

Transition Relation Specification

`trans` specifies that each transition is a consequence of the application of one of the operators to some individuals

```
pred trans [t,t': Time]  {
  (some m: Man, w: Woman |
    getMarried [m, w, t, t'])
  or
  (some p: Person, m: Man, w: Woman |
    isBornFromParents [p, m, w, t, t'])
  or
  (some p :Person | dies [p, t, t'])
}
```

System Specification

`System` specifies that each execution of the system start in a state satisfying the initial state condition and move from one state to the next through one transition at a time, until it reaches the final state

```
pred System {
  init[T/first]
  all t: Time - T/last | trans[t, T/next[t]]
}
run { System }
```

System Invariants

- Many of the facts that we stated in our static model now become expected system invariants
- These are properties that
  - should hold in initial states
  - should be preserved by system transitions
- In Alloy we can check that a property is invariant (in a given scope) by
  - encoding it as a formula P and checking
  - checking the assertion

```
System => all t: Time | P
```

Expected Invariants: Examples

```
-- People cannot be their own ancestors
assert a1 { System => all t: Time |
  no p: Person | p in p.^(parents.t)
}
check a1 for 8

-- No one can have more than one father or mother
assert a2 { System => all t: Time |
  all p: Person |
    lone (p.parents.t & Man) and
    lone (p.parents.t & Woman)
}
check a2 for 8
```

Exercises

- Load `family-7.als`
- Execute it
- Look at the generated instance
- Does it look correct?
- What if anything would you change about it?
- Check each of the given assertions
- Are they all valid?
- If not, how would you change the model to fix that?