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 Model Instances

```
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 can 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 = {}
```

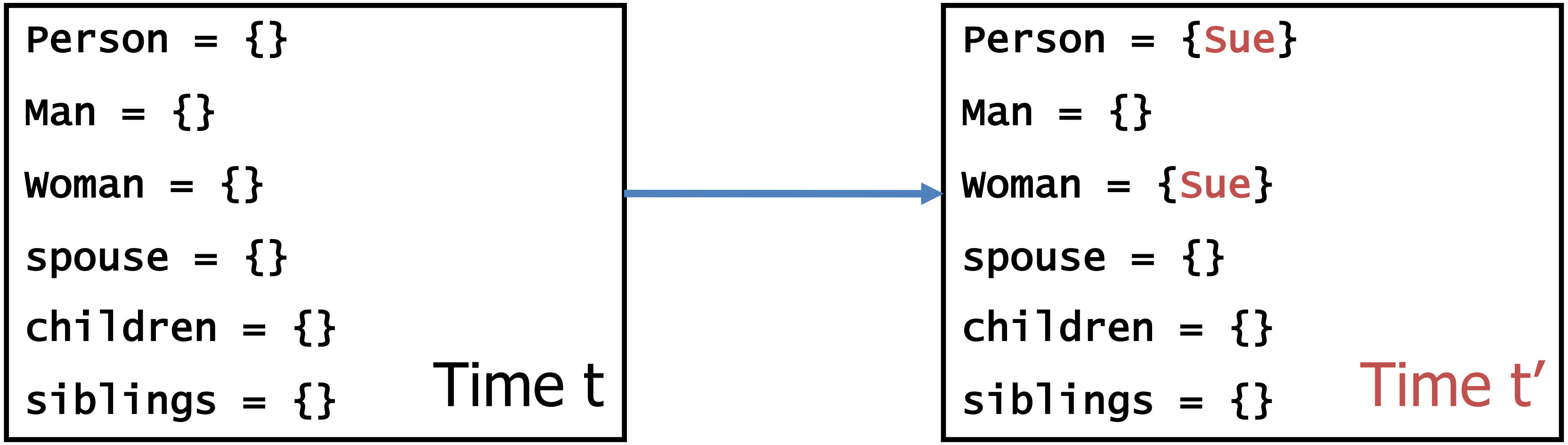
Time t

```
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



```
Person = {}
Man = {}
Woman = {}
spouse = {}
children = {}
siblings = {}
```

Time t

```
Person = {Sue}
Man = {}
Woman = {Sue}
spouse = {}
children = {}
siblings = {}
```

Time t’

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?

Alternative Approach: Electrum Alloy

A new version of Alloy with an implicit, built-in notion of (discrete) time

- A model instance is an infinite sequence of states
- Signatures/relations can change from state to state
- A new set of temporal operators allows us to express properties over time

Temporal Operators

Formula	**Meaning**
always p	p holds from current state forward
historically p	p holds from current state backward
after p	p holds in the next state
before p	p holds in the previous state
eventually p	p holds in the current state or a later on
once p	p holds in current state or an earlier one
p **until** q	p holds continuously until q holds
p **since** q	p has held continuously since last time q held
e’	value of e in next state

Example Traces

Time steps	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**…**												
p	•	•	•	•	•		•	•	•	•	•				•	•	•	•	•	•	•	…
q						•								•	•							…
always p															•	•	•	•	•	•	•	…
historically p	•	•	•	•	•																	…
after p	•	•	•	•		•	•	•	•	•				•	•	•	•	•	•	•	•	…
before p		•	•	•	•	•		•	•	•	•	•				•	•	•	•	•	•	…
eventually q	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•							…
once q						•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	…
p **until** q	•	•	•	•	•	•								•	•							…
p **since** q							•	•	•	•	•	•			•	•	•	•	•	•	•	…

Relations can Change Over Time

```
enum Liveness { Alive, Dead, Unborn }

abstract sig Person {
  var children: set Person,
  var parents: set Person,
  var siblings: set Person,
  var spouse: lone Person,
  var liveness: Liveness
}
sig Man, Woman extends Person {}
```

Revising Constraints

```
enum Liveness { Alive, Dead, Unborn }

abstract sig Person {
  var children: set Person,
  var spouse: lone Person,
  var liveness: Liveness
}
sig Man, Woman extends Person {}

fun parents[] : Person->Person {~children}
fun siblings[p: Person]:Person {{q: Person | …}}
```

Revising Constraints

```
pred BloodRelatives [p, q: Person] {
  some p.*parents & q.*parents
}
pred isAlive [p: Person] { p.liveness = Alive }
pred isDead [p: Person] { p.liveness = Dead }
pred isUnborn [p: Person] {p.liveness = Unborn}

-- a newborn is someone who has just been born
pred newBorn[p: Person] {
  isAlive[p] and before !isAlive[p]
}

pred isMarried [p: Person] { some p.spouse }
```

Revising *Static* Constraints

```
-- People cannot be their own ancestors
always no p: Person | p in p.^parents

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

-- the spouse relation is symmetric
always spouse = ~spouse
```

Revising *Static* Constraints

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


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

Revising *Static* Constraints

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


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

Adding *Temporal* Constraints

```
-- Dead people stay dead
always all p: Person |
  isDead[p] implies after isDead[p]

-- Dead people where once alive
always all p: Person |
  isDead[p] implies once isAlive[p]

-- No one lives forever
always all p: Person |
  isAlive[p] implies eventually isDead[p]
```

Adding *Temporal* Constraints

```
-- Living people never become unborn
always all p: Person |
  isAlive[p] implies always !isUnborn[p]

-- Live people stay alive until they die
always all p: Person |
  isAlive[p] implies
    (isAlive[p] until isDead[p])

-- Newborns have a father and a mother
always all p: Person | newBorn[p] implies
  some m:Man | some w: Woman | p.parents = m+w
```

Adding *Temporal* Constraints

```
-- Children were born from previously alive
-- parents
always all p, q: Person |
  p in q.children implies
    once (newBorn[p] and once isAlive[q])

-- People with parents have had those parents
-- since birth
always all p, q: Person |
  p in q.children implies
    (p in q.children since newBorn[p])
```

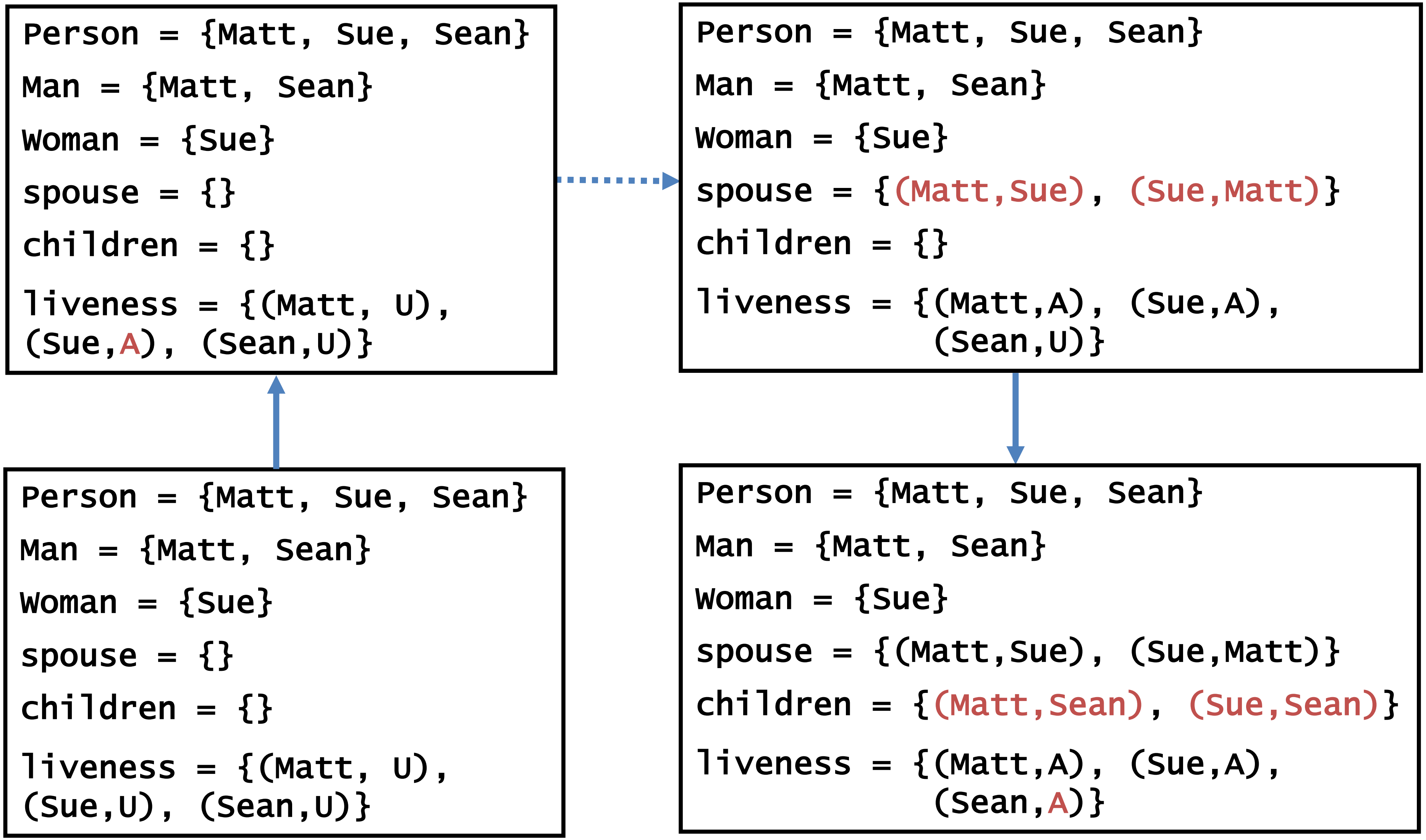
Exercises

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

Dynamics as State Transitions

- The evolution of a dynamic system can be modeled as a set of traces
- Each trace is a sequence of transitions from one state to another
- A transition can be thought of as caused by the application of a state transformer
- A state transformer is an operator that modifies the current state

Possible Trace



```
Person = {Matt, Sue, Sean}
Man = {Matt, Sean}
Woman = {Sue}
spouse = {}
children = {}
liveness = {(Matt, U),
(Sue,U), (Sean,U)}
```

```
Person = {Matt, Sue, Sean}
Man = {Matt, Sean}
Woman = {Sue}
spouse = {}
children = {}
liveness = {(Matt, U),
(Sue,A), (Sean,U)}
```

```
Person = {Matt, Sue, Sean}
Man = {Matt, Sean}
Woman = {Sue}
spouse = {(Matt,Sue), (Sue,Matt)}
children = {}
liveness = {(Matt,A), (Sue,A),
               (Sean,U)}
```

```
Person = {Matt, Sue, Sean}
Man = {Matt, Sean}
Woman = {Sue}
spouse = {(Matt,Sue), (Sue,Matt)}
children = {(Matt,Sean), (Sue,Sean)}
liveness = {(Matt,A), (Sue,A),
               (Sean,A)}
```

Transitions

A person is born from parents

State transformer that modifies `children` and `liveness` relations

```
Person = {Matt, Sue, Sean}
Man = {Matt, Sean}
Woman = {Sue}
spouse = {(Matt,Sue), (Sue,Matt)}
children = {}
liveness = {(Matt,Alive), (Sue,Alive),
            (Sean,Unborn)}
```

```
Person = {Matt, Sue, Sean}
Man = {Matt, Sean}
Woman = {Sue}
spouse = {(Matt,Sue), (Sue,Matt)}
children = {(Matt,Sean), (Sue,Sean)}
liveness = {(Matt,Alive), (Sue,Alive),
            (Sean,Alive)}
```

Expressing Transitions in Electrum

- A state transformer is modeled as a predicate over two states:
  1. the state right before the transition (current state) and
  2. the state right after it (next state)

- We use the temporal operators of Electrum Alloy to express constraints on the current and the next state

Expressing State Transformers

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

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

Example: Marriage

```
pred getMarried [p,q: Person] {
-- preconditions
    -- p and q are both alive
    isAlive[p] and isAlive[p]
    -- neither is married
    no (p+q).spouse
    -- they are not be blood relatives
    not BloodRelatives[p, q]
-- post-conditions
    -- p and q are each other’s spouses
    p.spouse' = q
    q.spouse' = p
-- frame conditions
                              ??
}
```

```
enum Liveness { Alive, Dead,
                Unborn }
abstract sig Person {
  var children: set Person,
  var spouse: lone Person,
  var liveness: Liveness }
sig Man,Woman extends Person {}
pred isAlive [p: Person]
  { p.liveness = Alive }
```

`spouse'` is the next version of `spouse`

Frame Condition

How is each relation impacted by marriage?

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

Frame Condition Predicates

```
pred noChildrenChangeExcept [P: set Person] {
  all p: Person - P |
    p.children' = p.children
}

pred noSpouseChangeExcept [P: set Person] {
  all p: Person - P |
    p.spouse' = p.spouse
}

pred noLivenessChangeExcept [P: set Person] {
  all p: Person - P |
    p.alive' = p.alive
}
```

Marriage Operator

```
pred getMarried [p, q: Person]
{
-- preconditions
   isAlive[p] and isAlive[q]
   no (m+w).spouse
   not BloodRelatives[m, w]
-- post-conditions
   p.spouse' = q and q.spouse' = p
-- frame conditions
    noSpouseChangeExcept[p+q]
    noChildrenChangeExcept[none]
   noLivenessChangeExcept[none]
}
```

Instance of Marriage

```
…
pred someMarriage {
  some m: Man | some w: Woman |
    getMarried[m, w]
}
-- there is a marriage initially
run { someMarriage }
-- there is a marriage initially or later on
run { eventually someMarriage }
-- there is a marriage eventually but not initially
run { not someMarriage and eventually someMarriage }
```

Birth from Parents Operator

```
pred isBornFromParents [p: Person, m: Man, w: Woman] {

  -- Pre-condition
     isUnborn[p]
     once (isAlive[w] and isAlive[m])
     isAlive[w]

  -- Post-condition and frame condition
     after isAlive[p]
     children' = children + (m -> p) + (m -> q)

  -- Frame condition
     noLivenessChangeExcept[p]
     noSpouseChangeExcept[none]
}
```

Instance of Birth

```
pred someBirth {
  some p1: Person, p2: Man, p3: Woman |
    isBornFromParents[p1, p2, p3]
}

run { eventually someBirth }
```

Death Operator

```
pred dies [p: Person] {

  -- Pre-condition
     isAlive[p]

  -- Post-condition
     after isDead[p]

 -- Post-condition and frame condition
     let q = p.spouse |
       spouse' = spouse - ((p -> q) + (q -> p))

  -- Frame conditions
     noChildrenChangeExcept[none]
     noLivenessChangeExcept[p]
}
```

Instance of Death

```
pred someDeath {
  some p: Person | dies[p]
}

run { eventually someDeath }

run {
  some p: Person |
    isAlive[p] and after isAlive[p] and
    eventually dies[p]
}
```

Specifying Transition Systems

- A transition system can be defined as a set of traces (aka executions):

  sequences of states generated by the operators

- In our example, for every execution:
  - The initial state satisfies some initialization condition
  - Each pair of consecutive states are related by
    - a birth operation, or
    - a death operation, or
    - a marriage operation

Initial State Specification

`init` specifies constraints on the initial state

```
fun LivingPeople[]: Person
{
   liveness.Alive
}
```

```
pred init [] {
   no children
   no spouse
   #LivingPeople > 2
   #Person > #LivingPeople
}
```

Transition Relation Specification

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

```
pred trans []  {
  (some m: Man, w: Woman | getMarried [m, w])
  or
  (some p: Person, m: Man, w: Woman |
     isBornFromParents [p, m, w])
  or
  (some p: Person | dies [p])
  or
  other  ???
}
```

The Need for a No-op

- For convenience, Electrum considers only infinite traces
- So we need a do-nothing operator for systems that can have finite executions

```
pred other [] {
  -- the relevant relations stay the same
  children' = children
  spouse' = spouse
  liveness' = liveness
}
```

System Specification

`System` specifies that

- each execution starts in a state satisfying the initial state condition and
- moves from one state to the next by the application of one operator at a time

```
pred System {
  init and always trans
}
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

- We can check that a property is invariant for a given system `System` (within a given scope) by
  - encoding it as a formula `F` and
  - checking the assertion

```
System => always F
```

Expected Invariants: Examples

```
-- People cannot be their own ancestors
assert a1 { System =>
  always no p: Person | p in p.^parents
}
check a1 for 6

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

Exercises

- Load `family-7-elec.als` in Electrum Alloy
- 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?

Exercises

- Load `dynamic/trash-1-elec.als` in Electrum Alloy
- Complete the model as instructed there
- Execute it
- Check each of the assertions you have written
- Are they all valid?
- If not, how would you change the model to fix that?