Alloy
Specifications using Relational Logic

Radek Mařík

Czech Technical University
Faculty of Electrical Engineering
Department of Telecommunication Engineering
Prague CZ

November 2, 2020

Outline

Outline

I Am My Own Grandpa - song

- The challenge is to create the situation of a man who is his own grandfather without being incest was committed or time travel was required.
- The lyrics of the song describe the solution.

I Am My Own Grandpa

Many many years ago, when I was twenty-three,
I was married to a widow as pretty as can be,
This widow had a grown-up daughter who had hair of red,
My father fell in love with her and soon the two were wed.

I'm my own grandpa, I'm my own grandpa.
It sounds funny, I know, but it really is so
I'm my own grandpa.

This made my dad my son-in-law and changed my very life,
For my daughter was my mother, for she was my father's wife.
To complicate the matter, even though it brought me joy,
I soon became the father of a bouncing baby boy.

My little baby thus became a brother-in-law to dad,
And so became my uncle, though it made me very sad,
For if he was my uncle then that also made him brother
To the widow's grown-up daughter, who of course was my step-mother.

. . . by Dwight B. Latham and Moe Jaffe

Father's wife then had a son who kept them on the run.
And he became my grandchild for he was my daughter's son.
My wife is now my mother's mother and it makes me blue,
Because although she is my wife, she's my grandmother, too.

Oh, if my wife's my grandmother then I am her grandchild.
And every time I think of it, it nearly drives me wild.
For now I have become the strangest case you ever sawn
As the husband of my grandmother, I am my own grandpa.

I'm my own grandpa, I'm my own grandpa.
It sounds funny, I know, but it really is so I'm my own grandpa.
I'm my own grandpa, I'm my own grandpa.
It sounds funny, I know, but it really is so
I'm my own grandpa.

I Am My Own Grandpa - Alloy Solution

```
module grandpa
abstract sig Person {
  father: lone Man,
  mother: lone Woman }
sig Man extends Person { wife: lone Woman }
sig Woman extends Person { husband: lone Man }
fact Biology { no p: Person | p in p.^(mother+father) }
fact Terminology { wife = ~husband }
fact SocialConvention {
  no wife & *(mother+father).mother
  no husband & *(mother+father).father }
fun grandpas [p: Person]: set Person {
  let parent = mother + father + father.wife + mother.husband |
  p.parent.parent & Man }
pred ownGrandpa [m: Man] { m in grandpas[m]  }
run ownGrandpa for 4 Person expect 1
```

Alloy - Usage

- Alloy is a modeling language for software design (see also Electrum)
- *??? However, it is not intended for modeling architecture (such as UML).*
- It is general enough to be able to model
  - any domain of individuals,
  - the relationship between them.
- syntax of Alloy 4.2 - 5.1 (watch out for tutorials for 3.x)
- **Typical Usage Steps**
  1. Specify model conditions
     - structures and relations between them
     - conditions as general facts
  2. Find a solution or counterexample
     - Solution: Find an instance of the model that meets all the conditions
     - Verification: conjecture construction
       - either cannot be refuted
       - or a counterexample is found

Outline

Atoms

- Everything is built on atoms and relations.
- **Atom** is a primitive entity that is
  - **indivisible:** cannot be divided into smaller parts,
  - **immutable:** its properties do not change over time,
  - **uninterpreted:** has no built-in property,
- **Relation** is a structure that captures the relationships between atoms.
  - It is a set of n-**tuples**,
  - each n-tuple is a sequence of atoms.

Signature

- **Signature** introduces a set of atoms.
- A declaration defining a set named A

```
sig A { }
```

- A set can be introduced as a **extension** of another set, thus $A1$ is a subset of the set A.

```
sig A1 extends A { }
```

- A signature declared independently of any other signature is the so-called **top-level** signature.
- Extensions of the same signature are mutually disjunct as well as top-level signatures.
- A set can be introduced as a **subset** of another set

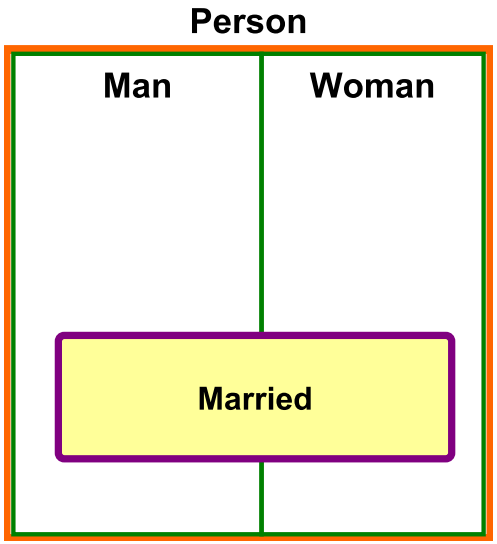
```
sig A1 in A { }
```

- **An abstract signature** has no elements except those belonging to its extensions or subsets.

```
abstract sig A { }
```

Signature - an Example

```
abstract sig Person { }
sig Man extends Person { }
sig Woman extends Person { }
sig Married in Person { }
```



Arrays

- Relations are declared as **fields** of signatures.
- A declaration defining a relation f,
  - whose domain is A a
  - whose range is given by the expression e.

```
sig A { f: e }
```

Examples

- Binary relation ... $f1$ is a subset of $A \times A$

```
sig A { f1: A }
```

- Ternary relation ... $f2$ is a subset of $B \times A \times A$

```
sig B { f2: A -> A }
```

Arrays

- Relations are declared as **fields** of signatures.
- A declaration defining a relation f,
  - whose domain is A a
  - whose range is given by the expression e.

```
sig A { f: e }
```

Examples

- Binary relation . . . $f1$ is a subset of $A \times A$

```
sig A { f1: A }
```

- Ternary relation . . . $f2$ is a subset of $B \times A \times A$

```
sig B { f2: A -> A }
```

Multiplicity

- It enables to constrain a size of sets.
  - A multiplicity keyword placed before a signature declaration constrains the number of elements in the signature's set

    ```
    m sig A { f: e }
    ```

  - We can constrain field multiplicity

    ```
    sig A { f:  m e }
    sig A { f: e1  m ->  n e2 }
    ```

- Four kinds of multiplicity exist
  - `set`: any number,
  - `some`: one or more,
  - `lone`: zero or one (L),
  - `one`: exactly one,
- The default keyword, if omitted, is `one`.
  Thus, the following declarations are equivalent:

  ```
  sig A { f:  e }
  sig A { f: one e }
  ```

Relations - Example 2

```
abstract sig Person {
  children: set Person,
  siblings: set Person,
}
sig Man, Woman extends Person { }
sig Married in Person {
  spouse: one Married,
}
```

Quantification

- Alloy supports a rich collection of quantifiers
  - `all x: S | F` : F holds for every x in S,
  - `some x: S | F` : F holds for some x in S,
  - `no x: S | F` : F holds for no x in S,
  - `lone x: S | F` : F holds for at most one x in S,
  - `one x: S | F` : F holds for exactly one x in S,

Logical Operators

- One can use usual logical operators

`not`	`!`	negation
`and`	`&&`	conjunction
`or`	`||`	disjunction
`implies`	`=>`	implication
`else`	`,`	alternativa
`iff`	`<=>`	iff (equivalence, bi-implication)

- An Example

  `a != b` is equivalent to `not a = b`

Sets and Their Operators

- Predefined set constants
  - `none` : an empty set,
  - `univ` : the universal set (contains all the atoms),
  - `ident` : the identity (each atom is mapped to itself),
- Set operators
  - `+` : union
  - `&` : intersection
  - `-` : difference
  - `in` : subset
  - `=` : equality
  - An example: married men

    ```
    Married & Man
    ```
- Set comprehension (CZ vymezená množina)
  - A set of values of the set S, for which F holds

    ```
    { x : S | F }
    ```

Relational Operators

`->`	arrow (product)
`~`	transpose
`.`	dot (join)
`[]`	box (join)
`^`	transitive closure
`*`	reflexive-transitive closure
`<:`	domain restriction
`:>`	range restriction
`++`	override

Product, Transpose

- The arrow product `p -> q`
  - p and q are two relations,
  - `p -> q` is the relation that contains every combination of a tuple from p and a tuple from q as their concatenation.
  - An example

```
Name = { (N0), (N1) }
Addr = { (D0), (D1) }
Book = { (B0) }
Book -> Name -> Addr = { (B0,N0,D0), (B0,N0,D1),
                         (B0,N1,D0), (B0,N1,D1) }
```

- Transpose `~p`
  - produces a mirror image of relation p
  - i.e. it revers the order of atoms in every tuple.
  - An example

```
example = { (a0,a1,a2,a3), (b0,b1,b2,b3) }
~example = { (a3,a2,a1,a0), (b3,b2,b1,b0) }
```

Tuple Dot Join

- `p.q` What is the composition (join) of these two tuples?
  - $p = (s_1, \ldots, s_n)$
  - $q = (t_1, \ldots, t_m)$
  - If $s_n ! = t_1$, then the result is empty.
  - If $s_n = t_1$, then the result is a tuple $(s_1, \ldots, s_{n-1}, t_2, \ldots, t_m)$
- An example for relations

  `{(a,b)}.{(a,c)} = {}`
  `{(a,b)}.{(b,c)} = {(a,c)}`
- What happens in the case `{(a)}.{(a)}`?
  - It is not defined!
  - $p.s$ is defined if and only if p and s are not both unary relations

Relation Join, Closures, Relation Restrictions

- `p.q`
  - p and q are two relations and both are not unary.
  - `p.q` is the relation that contains every combination of a tuple from p and a tuple from q as their join if that exists.
- `p[q]` (box join)
  - is semantically identical to dot join, but takes its arguments ordered in the reversed order.

$$p[q] \equiv q.p$$

- `^r = r + r.r + r.r.r + ...`
- `*r = ^r + iden`
- `s<:r` contains those tuples of r that **start** with an element in s (`range(s<:r)=s.r`)
- `r:>s`contains those tuples of r that **end** with an element in s (`domain(r<:s)=r.s`)
- `p++q = p - (domain(q) <: p) + q`

Dot Join Examples

```
module grandpa
abstract sig Person {
  father: lone Man,
  mother: lone Woman }
sig Man extends Person { wife: lone Woman }
sig Woman extends Person { husband: lone Man }
fact Biology { no p: Person | p in p.^(mother+father) }
```

```
Man =  {(Jirka), (Tomas),(Josef), (Vlada), (Franc)}
Woman =  {(Jana), (Lenka), (Tereza), (Olga)}
father = {(Jirka,Tomas), (Lenka,Tomas),(Tomas,Josef),
          (Josef, Vlada), (Jana, Franc)}
mother = {(Jirka, Jana), (Jana, Tereza), (Tomas, Olga)}
{(Jirka)}.father = {(Tomas)}
{(Jirka)}.mother = {(Jana)}
{(Jirka)}.father.father = {(Josef)}
{(Jirka)}.father.mother = {(Olga)}
{(Jirka)}.mother.father = {(Franc)}
{(Jirka)}.mother.mother = {(Tereza)}
```

Closure Principle Examples

```
module grandpa
abstract sig Person {
  father: lone Man,
  mother: lone Woman }
sig Man extends Person { wife: lone Woman }
sig Woman extends Person { husband: lone Man }
fact Biology { no p: Person | p in p.^(mother+father) }
```

```
Man =  {(Jirka), (Tomas),(Josef), (Vlada), (Franc)}
Woman =  {(Jana), (Lenka), (Tereza), (Olga)}
father = {(Jirka,Tomas), (Lenka,Tomas),(Tomas,Josef),
          (Josef, Vlada), (Jana, Franc)}
mother = {(Jirka, Jana), (Jana, Tereza), (Tomas, Olga)}
father + mother = {(Jirka,Tomas), (Lenka,Tomas),(Tomas,Josef),
          (Josef, Vlada), (Jana, Franc), (Jirka, Jana),
          (Jana, Tereza), (Tomas, Olga)}
{(Jirka)}.(father+mother) = {(Tomas), (Jana)}
{(Jirka)}.(father+mother).(father+mother) =
  {(Josef), (Olga), (Franc), (Tereza)}
{(Jirka)}.(father+mother).(father+mother).(father+mother) = {(Vlada)}
```

Let

- Expressions might be simplified.

  `let x = e | A`

  - A with each occurrence of the variable x replaced by the expression e.
- Examples
  - "A married person has just one spouse."

    ```
    sig Married in Person { spouse: one Married }
    ```
  - "Each married man (woman) has a wife (husband)."

    ```
    all p: Married |
      let q = p.spouse |
        (p in Man => q in Woman) and
        (p in Woman => q in Man)
    ```

Scalars

- Everything in Alloy is a set.
  - There are no scalars.
  - A singleton relation is used instead of scalars.

    ```
    let matt =  one Person
    ```

  - An interpretation using quantifications (a constraint that makes the set a singleton):

    ```
    all x : S | ... x ...
    ```

    $x = \{t\}$ for an element t of S

Facts

- Additional restrictions on signatures and fields can be expressed in Alloy as **facts**.
- AA (Alloy Analyzer) looks for instances of the model that also satisfy all of them restrictions determined by the facts.
- Example: "No person can be her/his own predecessor."

```
fact selfAncestor {
  no p: Person | p in p.^parents
}
```

Functions and Predicates

- They can be used as "macros".
  - They might be named and used multiple times in different contexts. (facts, statements, execution constraints).
  - They might be parametrized and used for specification shortening.
- **Functions**:
  - A named expression with no or more arguments.
  - It returns an expression as a returned value.
  - Functions are called during an analysis if they are referred by their name.
  - An example: "Parent relation."

```
fun parents []: Person -> Person { ~children }
```

  - An example: "Sisters."

```
fun sisters [p: Person]:
  { {w: Woman | w in p.siblings } }
```

  - An example: "No person can be her/his own ancestor or sister."

```
all p: Person |
  not (p in p.^parents or p in sisters[p])
```

Predicates

Predicates are useful in the following situations:

- Constraints that we do not want records as facts.
- constraints that can be used multiple times.
- They are called during an analysis only if they are referred by their name.
- An example: "Two people are blood related if they share an ancestor."

```
pred BloodRelated [p: Person, q: Person] {
  some p.*parents & q.*parents
}
```

- Příklad: "A person cannot be married to a blood relative."

```
no p: Married | BloodRelated [p, p.spouse]
```

Outline

Command Run

The **run** command

- Causes AA to parse the model by executing it.
- Instructs the tool to search for model instances.
- AA executes only a selected **run** command in the file.
- AA searches only in a limited instance space specified by scope.
- **Scope** represents the maximum number of tuples in each vertex signature.
- Default value of range $= 3$
- Examples

```
run {}              /* the scope is 3 */
run {} for 5        /* the scope is 5 */
run {some Man && no Married}     /* with conditions */
```

Statements

- We often believe that our model satisfies a certain limitation, a property that is not directly expressed.
- We can define such additional constraints as statements and use AA to validate them.
- If the constraint expressed by the statement is not met, AA produces an instance of the counterexample.
- Examples
  - "No person has a parent who is also a cousin."

```
assert a1 {all p: Person |
  no p.parents & p.siblings }
```

I Am My Own Grandpa - Alloy Solution

```
module grandpa
abstract sig Person {
  father: lone Man,
  mother: lone Woman }
sig Man extends Person { wife: lone Woman }
sig Woman extends Person { husband: lone Man }
fact Biology { no p: Person | p in p.^(mother+father) }
fact Terminology { wife = ~husband }
fact SocialConvention {
  no wife & *(mother+father).mother
  no husband & *(mother+father).father }
fun grandpas [p: Person]: set Person {
  let parent = mother + father + father.wife + mother.husband |
  p.parent.parent & Man }
pred ownGrandpa [m: Man] { m in grandpas[m]  }
run ownGrandpa for 4 Person expect 1
```

Outline

Ceilings and floors problem - Specifications

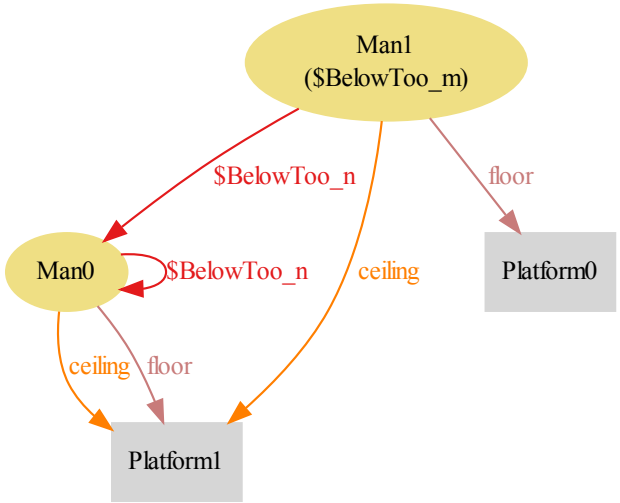
Paul Simons's song, 1973

"One Man's Ceiling Is Another Man's Floor"

```
module CeilingsAndFloors
sig Platform {}
sig Man {ceiling, floor: Platform}
fact PaulSimon {all m: Man | some n: Man | n.Above[m]}
pred Above[m, n: Man] {m.floor = n.ceiling}
```

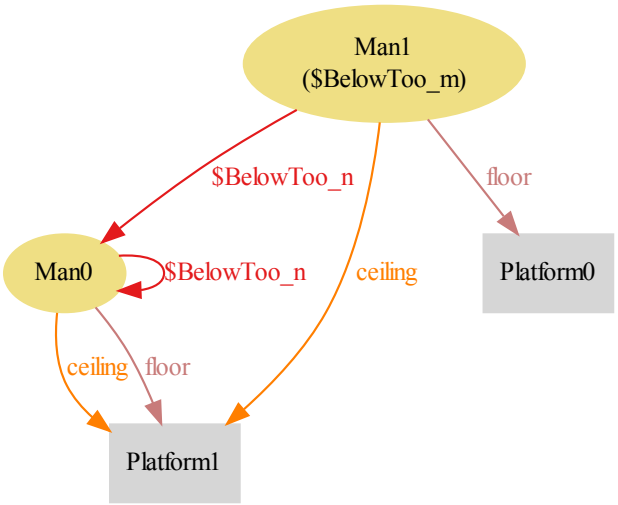
Ceilings and Floors - a counterexample solution

```
open CeilingsAndFloors
assert BelowToo { all m: Man | some n: Man | m.Above[n] }
check BelowToo for 2 expect 1
```



Ceilings and Floors - a counterexample solution

```
open CeilingsAndFloors
assert BelowToo { all m: Man | some n: Man | m.Above[n] }
check BelowToo for 2 expect 1
```



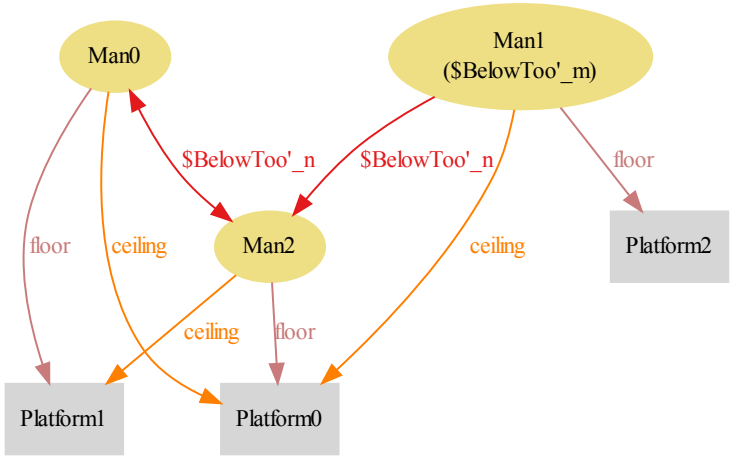
John Mc Naughton's Twisted House

Ceilings and Floors - a Counterexample Solution with Geometry

```
open CeilingsAndFloors
pred Geometry {no m: Man | m.floor = m.ceiling}
assert BelowToo' { Geometry =>
  (all m: Man | some n: Man | m.Above[n]) }
check BelowToo' for 2 expect 0
```

```
Executing "Check BelowToo' for 2 expect 0"
   Solver=sat4j Bitwidth=0 MaxSeq=0 SkolemDepth=1 Symmetry=20
   211 vars. 18 primary vars. 335 clauses. 15ms.
   No counterexample found. Assertion may be valid, as expected. 16ms.
```

```
check BelowToo' for 3 expect 1
```



Ceilings and Floors - a Solution with NoSharing

```
open CeilingsAndFloors

pred NoSharing {
  no m,n: Man | m!=n
    && (m.floor = n.floor || m.ceiling = n.ceiling) }

assert BelowToo'' { NoSharing
  => (all m: Man | some n: Man | m.Above[n]) }
check BelowToo'' for 6 expect 0
check BelowToo'' for 10 expect 0
```

```
Executing "Check BelowToo'' for 10 expect 0"
   Solver=sat4j Bitwidth=0 MaxSeq=0 SkolemDepth=1 Symmetry=20
   6750 vars. 330 primary vars. 14472 clauses. 296ms.
   No counterexample found. Assertion may be valid, as expected. 889ms.
```

Outline

Blue Planet Puzzles

- Blue planet creatures:
  - insane geniuses (blue creatures), who sometimes tell the truth and sometimes lie,
  - normals (yellow), who always lie, and
  - sane geniuses (green), who always tell the truth.
- The Master of Universe poses puzzles so that in each of the puzzles there is always only one insane genius, one sane genius and one normal.
- **Problem 1**
  - A: I am blue.
  - B: I am green if A is a normal.
  - C: I am yellow and A's statement is true.
  - *Who is what?*

Bijective Relation I

```
module SaneInsaneNormalOnBluePlanet
open util/boolean // Bool, True, False

abstract sig Color {}
one sig Blue, Green, Yellow extends Color {}

abstract sig Speaker {}
one sig Sane, Insane, Normal extends Speaker {}

abstract sig Creature { // the relation table
  speaker:  Speaker,
  color:    Color,
  speech:   Bool,
}
```

Bijective Relation II - (1:1) constrains

```
pred IsSane    [a:Creature]
  {a.speaker = Sane and a.speech = True}
pred IsNormal [a:Creature]
  {a.speaker = Normal and a.speech = False}
pred IsInsane [a:Creature]
  {a.speaker = Insane}
pred TrueSentence [a: Creature]
  {IsSane[a] or IsNormal[a] or IsInsane[a]}

fact BluePlanet {
  all c: Creature | TrueSentence[c]
  Blue.~color.speaker = Insane // (1:1) correspondences
  Yellow.~color.speaker = Normal
  Green.~color.speaker = Sane
}
```

Bijective Relation III - Specifications

```
fact MasterPuzzles { // (1:1) mappings
  all s: Speaker | one s.~speaker
  all c: Color | one c.~color
}
one sig A, B, C extends Creature {}

pred Problem1 [] {
  A.speech=True <=> A.color = Blue
  B.speech=True <=> (A.speaker = Normal => B.color = Green)
  C.speech=True <=> (A.speech = True and C.color = Yellow)
}
run {Problem1}
```

A solution: A is a normal, B a sane genius and C an insane genius.

Ordering - a sequence of houses

```
open util/ordering[House] as hsOrd
abstract sig House {}
one sig H1, H2, H3, H4 extends House {}{
  hsOrd/next[H1] = H2
  hsOrd/next[H2] = H3
  hsOrd/next[H3] = H4 }
abstract sig Nationality {
  lives:  one House,
  drinks: one Drink,
  owns:   one Animal,
  smokes: one Cigarette, }
fact {
  H3.~lives.drinks = Milk //9
  Chesterfields.~smokes.lives in
    (  hsOrd/prev[Fox.~owns.lives]
     + hsOrd/next[Fox.~owns.lives]) //11 }
```

Outline

Dynamic Models

- system modeling with **states** and **transitions**
- modeling **operations** that cause transitions

Alloy

- does not know the concept of state transition,
- Sets defined in signatures are fixed.
- Dynamic aspects can be modeled by time-dependent relations.
- There are several ways to model system dynamics
  - signature specification `Time` expressing time
  - add a time component to each session that changes over time.

Dynamic Models

- system modeling with **states** and **transitions**
- modeling **operations** that cause transitions

Alloy

- does not know the concept of state transition,
- Sets defined in signatures are fixed.
- Dynamic aspects can be modeled by time-dependent relations.
- There are several ways to model system dynamics
  - signature specification `Time` expressing time
  - add a time component to each session that changes over time.

Static Family Model

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

sig Man, Woman extends Person { }

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

Dynamic Family Model

```
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,
}
```

Signatures are time independent

- `Married` is not modeled correctly
- $t : \text{Married} = \{\}$ vs. $t' : \text{Married} = \{(\text{Peter}), (\text{Sue})\}$

Transition Specification in Alloy I

- A transition can be modeled as a predicate between two states:
  - **state just before** transition a
  - **state just after** transition.
- Defined as a predicate with (at least) two formal parameters:
  `t, t ': Time`
- Constraints that define the state at both times.

Transition Specification in Alloy II

- **Input Conditions**
  - What applies in the states before the transition.
- **Output conditions**
  - A description of the effects of the transition that generate the following state
- **Invariant**
  - A description of what does not change
- It is recommended to comment well on individual categories of conditions and invariant

Hotel Keys Protocol - Basic Signatures

```
module hotel
open util/ordering [Time] as TO
open util/ordering [Key] as KO
sig Key {}
sig Time {}
sig Room {
  keys: set Key,
  currentKey: Key one -> Time }
sig Guest { gkeys: Key -> Time }
one sig FrontDesk {
  lastKey: (Room -> lone Key) -> Time,
  occupant: Room -> Guest -> Time }
fun nextKey [k: Key, ks: set Key]: set Key {
  KO/min [KO/nexts[k] & ks] }
```

Hotel Keys Protocol -Invariants

```
fact {
  all k: Key | lone keys.k
  all r:Room, t:Time| r.currentKey.t in r.keys }

pred noFrontDeskChange [t,t': Time] {
  FrontDesk.lastKey.t = FrontDesk.lastKey.t'
  FrontDesk.occupant.t = FrontDesk.occupant.t' }

pred noRoomChangeExcept [rs: set Room, t,t': Time] {
  all r: Room - rs | r.currentKey.t = r.currentKey.t' }

pred noGuestChangeExcept [gs: set Guest, t,t': Time]  {
  all g: Guest - gs | g.gkeys.t = g.gkeys.t' }
```

Hotel Keys Protocol - Registration

```
pred checkin [ g: Guest, r: Room, k: Key, t,t': Time ] {
  // the guest holds the input key
  g.gkeys.t' = g.gkeys.t + k
  let occ = FrontDesk.occupant | {
    // the room has no current occupant
    no r.occ.t
    // the guest becomes the new occupant of the room
    occ.t' = occ.t + r->g }
  let lk = FrontDesk.lastKey | {
    // the input key becomes the room's current key
    lk.t' = lk.t ++ r->k
    // the input key is the successor of the last key in
    // the sequence associated to the room
    k = nextKey [r.lk.t, r.keys] }
  noRoomChangeExcept [none, t, t']
  noGuestChangeExcept [g, t, t'] }
```

Hotel Keys Protocol - Room Entry

```
pred entry [ g: Guest, r: Room, k: Key,  t, t': Time ] {
  // the key used to open the lock is one of
  // the key the guest holding
  k in g.gkeys.t
  // pre and post conditions
  let ck = r.currentKey |
    // not a new guest
    (k = ck.t and ck.t' = ck.t)
    // new guest
    or (k = nextKey [ck.t, r.keys] and ck.t' = k)
  // frame conditions
  noRoomChangeExcept [r, t, t']
  noGuestChangeExcept [none, t, t']
  noFrontDeskChange [t, t'] }
```

Hotel Keys Protocol - Checkout

```
pred checkout [ g: Guest, t,t': Time ] {
  let occ = FrontDesk.occupant | {
    // the guest occupies one or more rooms
    some occ.t.g
    // the guest's room become available
    occ.t' = occ.t - (Room -> g)
  }
  // frame condition
  FrontDesk.lastKey.t = FrontDesk.lastKey.t'
  noRoomChangeExcept [none, t, t']
  noGuestChangeExcept [none, t, t']
}
```

Hotel Keys Protocol - Dynamics (in Time)

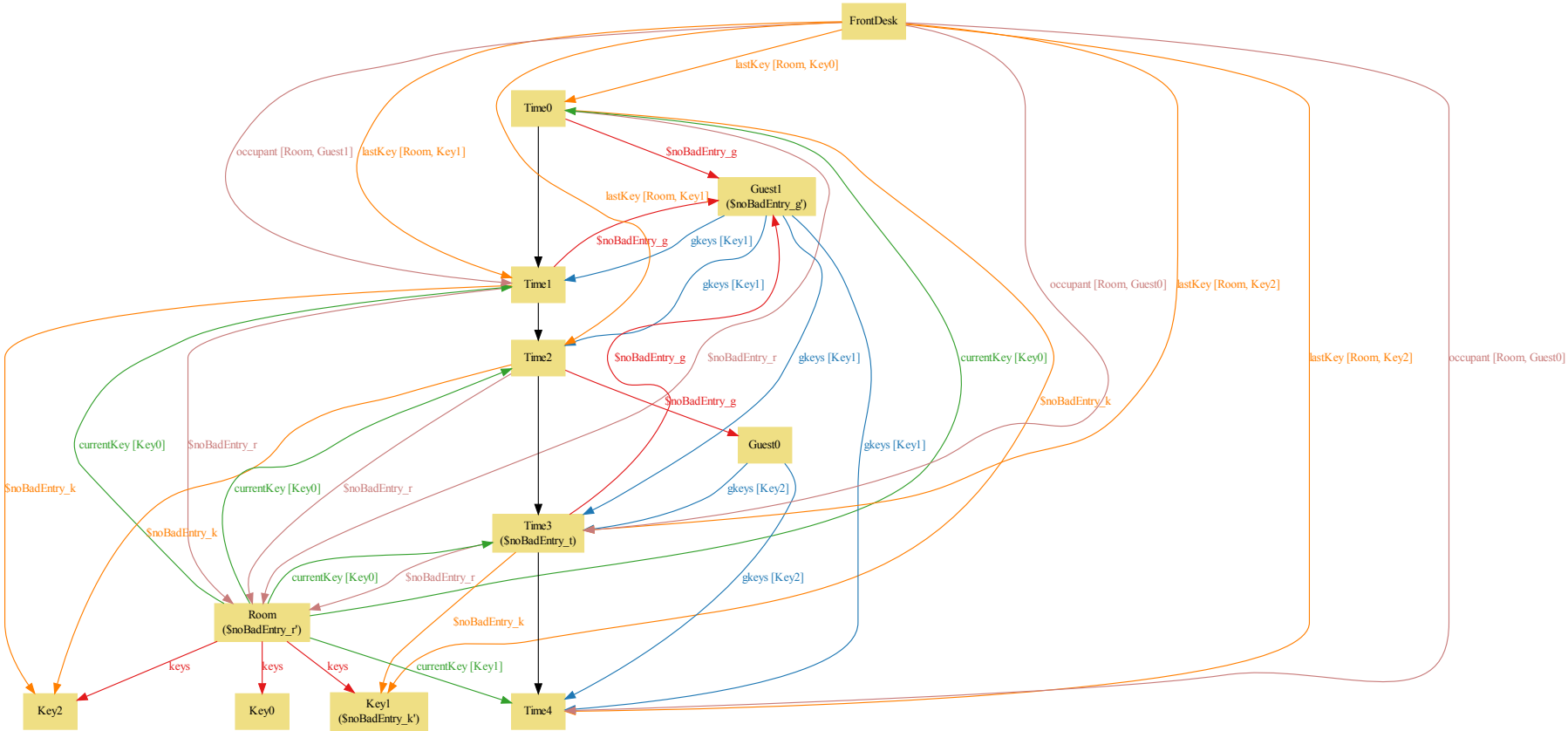
```
pred init [t: Time] {
  // no guests have keys
  no Guest.gkeys.t
  // the roster at the front desk shows no room as occupied
  no FrontDesk.occupant.t
  // the record of each room's key at the
  // front desk is synchronized with the
  // current combination of the lock itself
  all r: Room | r.(FrontDesk.lastKey.t) = r.currentKey.t }
fact Traces {
  init [TO/first]
  all t: Time - TO/last |
    let t' = TO/next [t] | some g: Guest, r: Room, k: Key |
      entry [g, r, k, t, t']
      or checkin [g, r, k, t, t'] or checkout [g, t, t'] }
```

Hotel Keys Protocol - Verification

```
assert noBadEntry  {
  all t: Time, r: Room, g: Guest, k: Key |
    let t' = TO/next [t], o = r.(FrontDesk.occupant).t |
      (entry [g, r, k, t, t'] and some o)
      implies g in o
}

check noBadEntry for 3 but 2 Room, 2 Guest, 5 Time
```

Hotel Keys Protocol - a Counterexample



Hotel Keys Protocol - a Correction

```
fact noIntervening {
  all t: Time - TO/last |
    let t' = TO/next [t], t'' = TO/next [t'] |
      all g: Guest, r: Room, k: Key |
        checkin [g, r, k, t, t']
        implies (entry [g, r, k, t', t''] or no t'')
}

check noBadEntry for 3 but 2 Room, 2 Guest, 10 Time
```

References I