CS:5810 Formal Methods in Software Engineering

Introduction to Alloy 6

Part 1

Outline

- Introduction to basic Alloy constructs using a simple example of a static model
  - How to define sets, subsets, relations with multiplicity constraints
  - How to use Alloy's quantifiers and predicate forms
- Basic use of the Alloy Analyzer 4 (AA)
  - Loading, compiling, and analyzing a simple Alloy specification
  - Adjusting basic tool parameters
  - Using the visualization tool to view instances of models

Roadmap

Alloy: Rationale and Use Strategies

- What types of systems have been modeled with Alloy
- What types of questions can AA answer
- What is the purpose of each of the sections in an Alloy specification

Alloy Specifications

- Parameterized conditionals
- Indexed relations
- Graphical representations of Alloy models
- More complex examples

Alloy --- Why was it created?

Lightweight

small and easy to use, and capable of expressing common properties tersely and naturally

Precise

having a simple and uniform mathematical semantics

Tractable

amenable to efficient and fully automated semantic analysis (within scope limits)

Alloy --- Comparison

UML

- Has similarities (graphical notation, OCL constraints) but it is neither lightweight, nor precise
- UML includes many modeling notions omitted from Alloy (use-cases, state-charts, code architecture specs)
- Alloy’s diagrams and relational navigation are inspired by UML

Z

- Precise, but intractable. Stylized typography makes it harder to work with
- Z is more expressive than Alloy, but more complicated
- Alloy’s set-based semantics is inspired by Z

Alloy --- What is it used for?

Alloy is a textual modeling language aimed at expressing:

structural and behavioral properties of software systems

It is not meant for modeling code architecture
(*a la* class diagrams in UML)

But there may be a close relationship between the Alloy specification and an implementation in an OO language

Example Applications

The Alloy 6 distribution comes with several example models that together illustrate the use of Alloy's constructs

Examples

- Specification of a distributed spanning tree
- Model of a generic file system
- Model of a generic file synchronizer
- Tower of Hanoi model
- …

Alloy in General

Alloy is general enough that it can model

- any (finite) domain of individuals and
- any relations between them

We will then start with a few simple examples that are not necessarily about about software

Example: Family Structure

We want to…

- Model parent/child relationships as primitive relations
- Model spousal relationships as primitive relations
- Model relationships such as "siblings" as *derived* relations
- Enforce certain biological constraints via 1st-order constraints (e.g., people have only one mother)
- Enforce certain social constraints via 1st-order constraints (e.g., a wife isn't a sibling)
- Confirm or refute the existence of certain derived relationships (e.g., no one has a sister who is also their wife)

Example: addressBook

An address book for an email client that maintains a mapping from names to addresses

FriendBook
Ted -> ted@gmail.com Ryan -> ryan@hotmail.com

WorkBook
Pilard -> pilard@uiowa.edu Ryan -> ryan@uiowa.edu

Atoms and Relations

In Alloy, everything is built from atoms and relations

An *atom* is a primitive entity that is

- *indivisible*: it cannot be broken down into smaller parts
- *immutable*: it does not change over time
- *uninterpreted*: it does not have any built-in property (the way numbers do for example)

A *relation* is a structure that relates atoms

- It is a set of tuples of the same type

Atoms and Relations: Examples

- Unary relations: a set of names, a set of addresses and a set of books

  Name = { (N0), (N1), (N2) }

  Addr = { (D0), (D1) }

  Book = { (B0), (B1) }

- A binary relation from names to addresses

  address = { (N0,D0), (N1,D1) }

- A ternary relation from books to name to addresses

  addr = { (B0,N0,D0), (B0,N1,D1), (B1,N1,D2) }

Atoms

Tuples

Relations

Size of a relation: the number of tuples in the relation

Arity of a relation: the number of atoms in each tuple of the relation

relations with arity 1, 2, and 3 are said to be *unary*, *binary*, and *ternary* relations

Examples.

- relation of arity 1 and size 1: myName = {(N0)}
- relation of arity 2 and size 3: address = {(N0,D0), (N1,D1), (N2,D1))

Main Components of Alloy Models

- Signatures and Fields
- Predicates and Functions
- Facts
- Assertions
- Commands and scopes

Signatures and Fields

Signatures

- Describe classes of entities we want to reason about
- Sets defined in signatures are fixed (we will see how to model dynamic aspects later)

Fields

- Define relations between signatures

Simple constraints

- Multiplicities on signatures
- Multiplicities on relations

Signatures

- A *signature* introduces a set of atoms

- The declaration

  ```
  sig A {}
  ```

  introduces a set named A

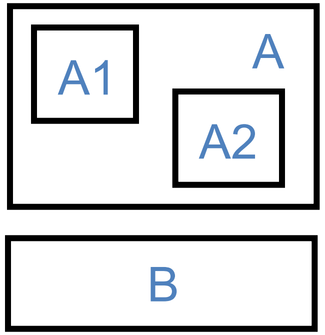
- A signature can be declared as an *extension* of another

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

  introduces a set name A1 that is a subset of A

Signatures

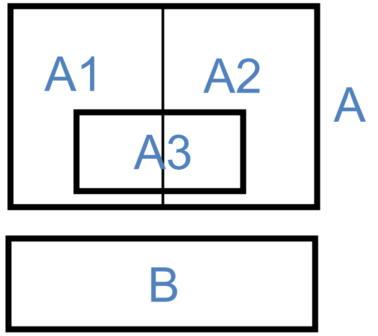
```
sig A {}
sig B {}
sig A1 extends A {}
sig A2 extends A {}
```



- A1 and A2 are extensions of A
- A signature declared independently of any other one is a *top-level signature*, e.g., A and B
- Extensions of the same signature are mutually disjoint, as are top-level signatures

Signatures

```
abstract sig A {}
sig B {}
sig A1 extends A {}
sig A2 extends A {}
```



- A signature can be introduced as a subset of another

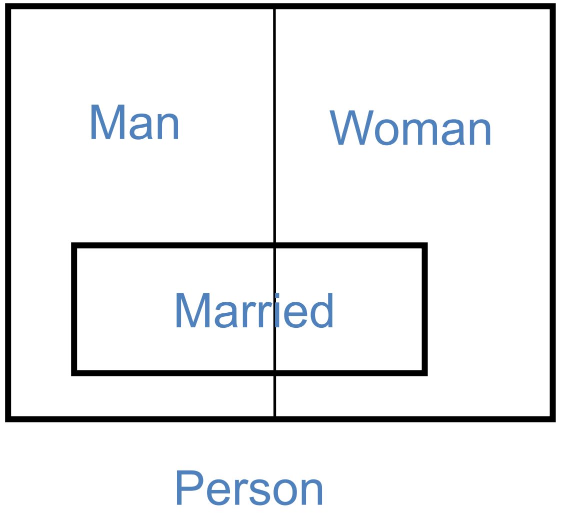
```
sig A3 in A {}
```

- An *abstract* signature has no elements except those belonging to its extensions or subsets
- All extensions of an abstract signature A form a partition of A

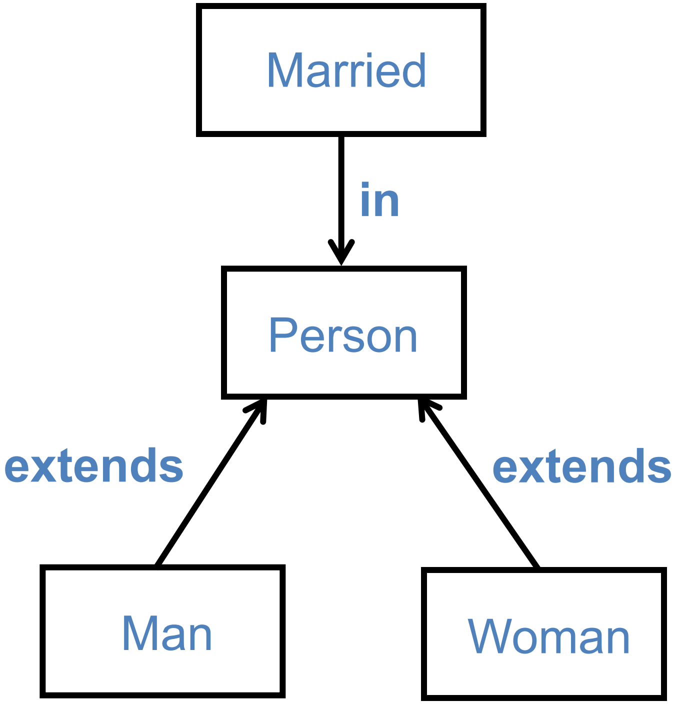
Example: Family Structure

Alloy Model

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



Graphical Representation



Model Instances

The Alloy Analyzer will generate instances of models so that we can check if they match our intentions. Which of the following are instances of our current model?

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

✗ B.
Person = { (P0), (P1), (P2) }
Man = { (P1), (P2) }
Married = { }
Woman = { (P0), (P1) }

✗ D.
Person = { (P0), (P1) }
Man = { (P0) }
Married = { (P1) }
Woman = { }

✓ A.
Person = { (P0), (P1), (P2) }
Man = { (P1), (P2) }
Married = { }
Woman = { (P0) }

✓ C.
Person = { (P0), (P1), (P2), (P3) }
Man = { (P0), (P1), (P2), (P3) }
Married = { (P2), (P3) }
Woman = { }

✓ E.
Person = { (P0), (P1) }
Man = { (P0) }
Married = { (P1), (P0) }
Woman = { (P1) }

Fields

- *Relations* are declared as *fields* of signatures
  - Writing

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

    introduces a relation f of type A x e,
    where e is an expression denoting a product of signatures
- **Examples:** (with signatures A, B, C)
  - Binary Relation:

    ```
    sig A { f1: B }        // f1 is a subset of A x B
    ```
  - Ternary Relation:

    ```
    sig A { f2: B -> C }   // f2 is a subset of A x B x C
    ```

Example Signatures and Fields

Family Structure:

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

sig Man, Woman extends Person {}

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

Fields

Example: Family Structure

Alloy Model with siblings

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

siblings *is a binary relation*

it *is a subset of* Person x Person

Example instance

Person = { (P0), (P1) }
Man = { (P0) }
Married = { }
Woman = { (P1) }

siblings = { (P0,P1), (P1,P0) }

Intuition: P0 and P1 are siblings

Multiplicities

Allow us to constrain the sizes of sets

- A multiplicity keyword placed before a signature declaration constraints the number of elements in the signature

```
m sig A {}
```

- We can also make multiplicities constraints on fields

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

There are four multiplicities `m`:

`set` : any number
`some` : one or more
`one` : exactly one
`lone` : zero or one

Cardinality Constraints

Multiplicities can also be applied to expressions denoting relations

- `some e` — e is non-empty
- `no e` — e is empty
- `lone e` — e has at most one tuple
- `one e` — e has exactly one tuple

Multiplicities: Examples

Without multiplicity:

A set of colors, each of which is a red, yellow or green color

```
abstract sig Color {}
sig Red, Yellow, Green extends Color {}
```

(can have more than one red, one yellow and one green color)

With multiplicity:

An enumeration of colors

```
abstract sig Color {}
one sig Red, Yellow, Green extends Color {}
```

(exactly one red, one yellow and one green color)

Multiplicities: Examples

- A file system in which each directory contains any number of objects, and each alias points to exactly one object

```
abstract sig Object {}
sig Directory extends Object { contents: set Object }
sig File extends Object {}
sig Alias in File { to: one Object }
```

redundant

- The default multiplicity for fields is **one**, so:

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

are equivalent

Multiplicities: Examples

- A book maps names to addresses
  - There is at most one address per name
  - An address is associated to at least one name

```
sig Name, Addr {}
sig Book {
   addr: Name some -> lone Addr
}
```

Multiplicities: Examples

- A collection of weather forecasts, each of which has a field `weather` associating every city with exactly one weather condition

```
sig Forecast { weather: City -> one Weather }
sig City {}
abstract sig Weather {}
one sig Rainy, Sunny, Cloudy extends Weather {}
```

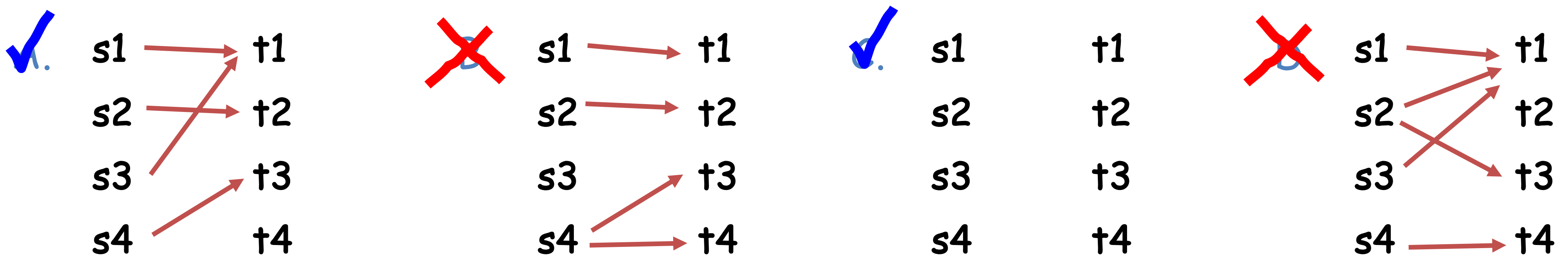
- Instance:

  City = { (Iowa City), (Chicago) }
  Rainy = { (rainy) }
  Sunny = { (sunny) }
  Cloudy = { (cloudy) }
  Forecast = { (f1), (f2) }
  weather = { (f1, Iowa City, rainy), (f1, Chicago, rainy),
  (f2, Iowa City, rainy), (f2, Chicago, sunny) }

Multiplicities and Binary Relations

- `sig S { f: lone T }`
  - says that, for each element s of S, f maps s to at most one value in T
- Potential instances:

Conventional name: partial function

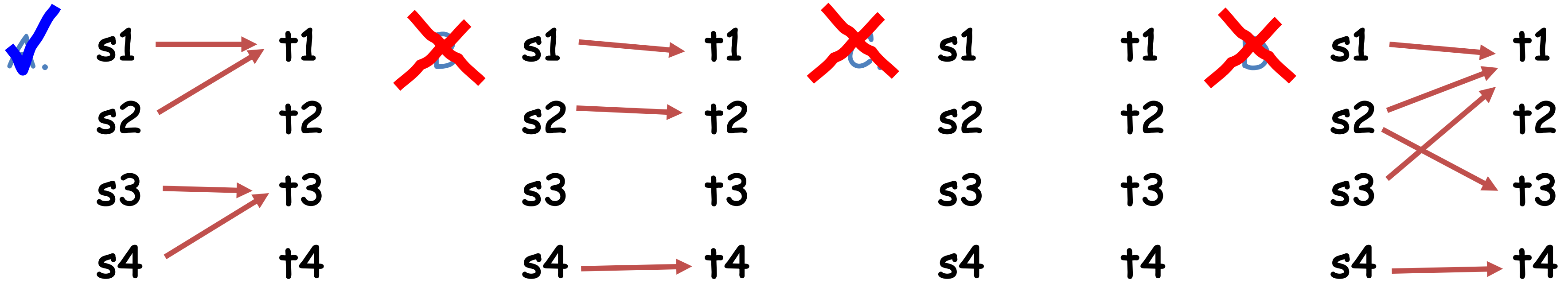


Multiplicities and Binary Relations

- `sig S { f: one T }`
  - says that, for each element s of S, f maps s to exactly one value in T

- Potential instances:

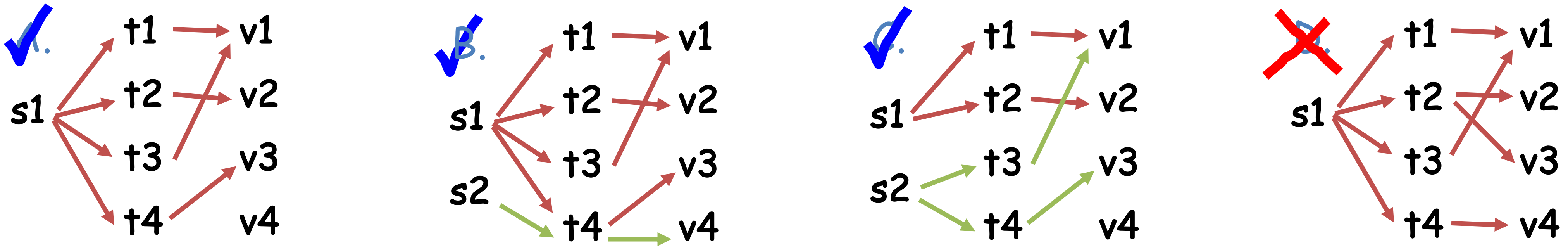
Conventional name: total function



Multiplicities and Ternary Relations

- `sig S { f: T -> one V }`
  - For each element s of S, *over the triples that start with* s:
    f maps each T-element to exactly one V-element

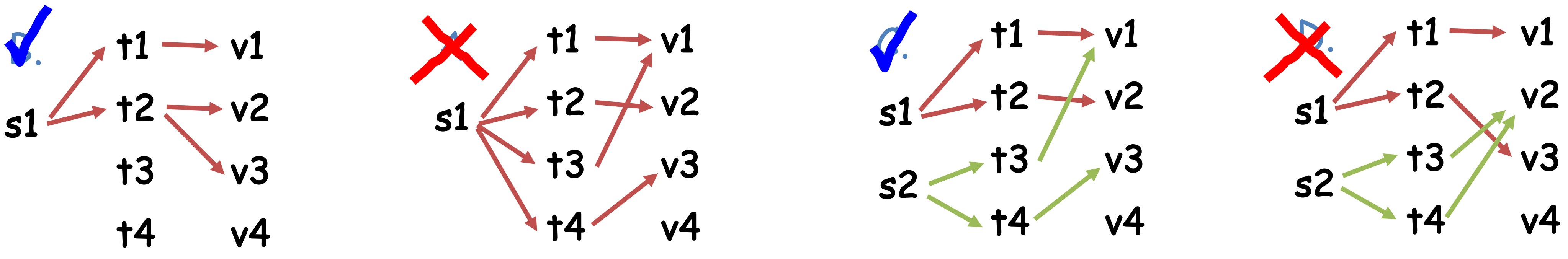
- Potential instances:



Multiplicities and Ternary Relations

- `sig S { f: T lone -> V }`
  - For each element s of S, *over the triples that start with* s: f maps at most one T-element to the same V-element

- Potential instances:



Multiplicities and Relations

- Other kinds of relational structures can be specified using multiplicities

- Examples:
  - `sig S { f: some T }` total relation
  - `sig S { f: set T }` partial relation
  - `sig S { f: T set -> set V }`
  - `sig S { f: T one -> V }`
  - …

Example: Family Structure

- How would you use multiplicities to define the children relation?

```
sig Person { children: set Person }
```

  – Intuition: each person has zero or more children

- How would you use multiplicities to define the spouse relation?

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

  – Intuition: each married person has exactly one spouse

Summarizing

Alloy Model

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

sig Man, Woman extends Person {}

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

Exercises

- Start the Alloy Analyzer
- Load file `family-1.als` from the **Resources** section of the course website
- Execute it
- Analyze the model instance
- Look at the generated instance
- Does it look correct?
- What, if anything, would you change about it?

Model Instance

Instance found:

```
Person = {Man0,Man1,Man2}
Man = {Man0,Man1,Man2}
Woman = {}
Married = {Man0,Man1,Man2}

children = { (Man0,Man0),(Man0,Man1),(Man1,Man0),(Man2,Man1),
             (Man2,Man2) }

siblings = { (Man0,Man0),(Man0,Man1),(Man1,Man0),(Man1,Man2),
             (Man2,Man2) }

spouse = { (Man1,Man0),(Man0,Man2),(Man2,Man0) }
```

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

No Women?

Instance found:

```
Person = {Man0,Man1,Man2}
Man = {Man0,Man1,Man2}
Woman = {}
Married = {Man0,Man1,Man2}

children = { (Man0,Man0),(Man0,Man1),(Man1,Man0),(Man2,Man1),
             (Man2,Man2) }

siblings = { (Man0,Man0),(Man0,Man1),(Man1,Man0),(Man1,Man2),
             (Man2,Man2) }

spouse = { (Man1,Man0),(Man0,Man2),(Man2,Man0) }
```

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

Man is his own child ?

Instance found:

```
Person = {Man0,Man1,Man2}
Man = {Man0,Man1,Man2}
Woman = {}
Married = {Man0,Man1,Man2}

children = { (Man0,Man0),(Man0,Man1),(Man1,Man0),(Man2,Man1),
             (Man2,Man2) }

siblings = { (Man0,Man0),(Man0,Man1),(Man1,Man0),(Man1,Man2),
             (Man2,Man2) }

spouse = { (Man1,Man0),(Man0,Man2),(Man2,Man0) }
```

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

Multiple Fathers?

Instance found:

```
Person = {Man0,Man1,Man2}
Man = {Man0,Man1,Man2}
Woman = {}
Married = {Man0,Man1,Man2}

children = { (Man0,Man0),(Man0,Man1),(Man1,Man0),(Man2,Man1),
             (Man2,Man2) }

siblings = { (Man0,Man0),(Man0,Man1),(Man1,Man0),(Man1,Man2),
             (Man2,Man2) }

spouse = { (Man1,Man0),(Man0,Man2),(Man2,Man0) }
```

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

Own-Siblings?

Instance found:

```
Person = {Man0,Man1,Man2}
Man = {Man0,Man1,Man2}
Woman = {}
Married = {Man0,Man1,Man2}

children = { (Man0,Man0),(Man0,Man1),(Man1,Man0),(Man2,Man1),
             (Man2,Man2) }

siblings = { (Man0,Man0),(Man0,Man1),(Man1,Man0),(Man1,Man2),
             (Man2,Man2) }

spouse = { (Man1,Man0),(Man0,Man2),(Man2,Man0) }
```

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

Asymmetric Siblings?

Instance found:

```
Person = {Man0,Man1,Man2}
Man = {Man0,Man1,Man2}
Woman = {}
Married = {Man0,Man1,Man2}

children = { (Man0,Man0),(Man0,Man1),(Man1,Man0),(Man2,Man1),
             (Man2,Man2) }

siblings = { (Man0,Man0),(Man0,Man1),(Man1,Man0),(Man1,Man2),
             (Man2,Man2) }

spouse = { (Man1,Man0),(Man0,Man2),(Man2,Man0) }
```

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

No `(Man2,Man1)?`

Child and Sibling?

Instance found:

```
Person = {Man0,Man1,Man2}
Man = {Man0,Man1,Man2}
Woman = {}
Married = {Man0,Man1,Man2}

children = { (Man0,Man0),(Man0,Man1),(Man1,Man0),(Man2,Man1),
             (Man2,Man2) }

siblings = { (Man0,Man0),(Man0,Man1),(Man1,Man0),(Man1,Man2),
             (Man2,Man2) }

spouse = { (Man1,Man0),(Man0,Man2),(Man2,Man0) }
```

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

Asymmetric Marriage?

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

Instance found:

```
Person = {Man0,Man1,Man2}
Man = {Man0,Man1,Man2}
Woman = {}
Married = {Man0,Man1,Man2}

children = { (Man0,Man0),(Man0,Man1),(Man1,Man0),(Man2,Man1),
             (Man2,Man2) }

siblings = { (Man0,Man0),(Man0,Man1),(Man1,Man0),(Man1,Man2),
             (Man2,Man2) }

spouse = { (Man1,Man0),(Man0,Man2),(Man2,Man0) }
```

where is (Man0,Man1)?

Model Weaknesses

- The model is underconstrained
  - It doesn't fully match our domain knowledge
  - We can add constraints to enrich the model

- Under-constrained models are common early on in the development process
  - The Alloy Analizer gives quick feedback on weaknesses in our model
  - We can incrementally add constraints until we are satisfied with it

Adding Constraints

We'd like to enforce the following constraints which are simply matters of biology

- *No person can have more than one (biological) father or mother*
- *People cannot be their own (biological) parent or, more generally, their own ancestor*
- *A person's siblings are people with the same parents as the person's parents*

Adding Constraints

We'd like to enforce the following social constraints

- *The spouse relation is symmetric*

- *You cannot marry your own sibling*