22c:181 / 55:181
Formal Methods in Software Engineering

Introduction to Alloy

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 model language for software design
- It is not meant for modeling code architecture (*a la* class diagrams in UML)
- But there might be a close relationship between the Alloy specification and an implementation in an OO language

Alloy --- Example Applications

- The Alloy 4 distribution comes with over a dozen of 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 domain of individuals and
  - 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 predicates (e.g., only one mother)
  - Enforce certain social constraints via 1st-order predicates (e.g., a wife isn't a sibling)
  - Confirm or refute the existence of certain derived relationships (e.g., no one has a wife with whom he shares a parent)

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 -> lpilard@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 can't be broken down into smaller parts
  - *Immutable*: its properties don't 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, each tuple being a sequence of atoms

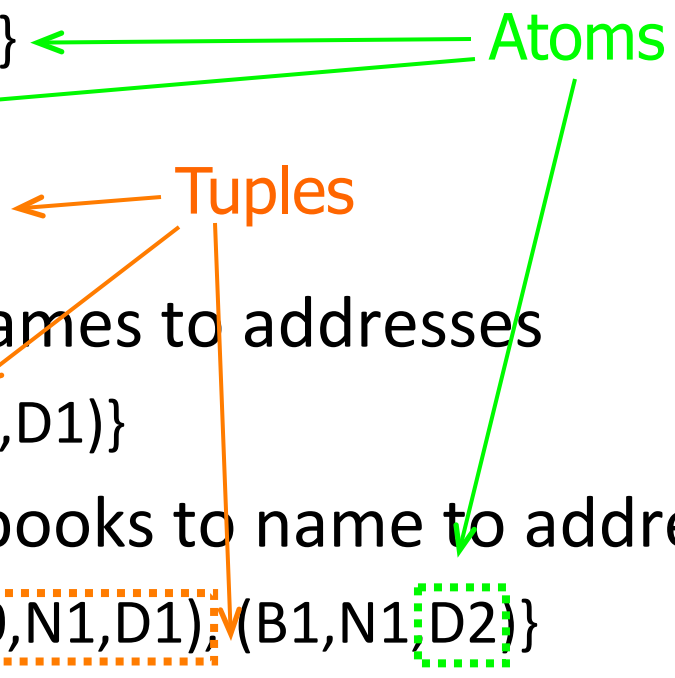
Atoms and Relations: Examples

- 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)}

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))

Alloy Specifications

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

Signatures and Fields

- Signatures
  - Describe the entities that you want to reason about
  - Sets defined in signatures are fixed (concept related to operations and dynamic models)
- 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 set can be introduced as an extension of another; thus

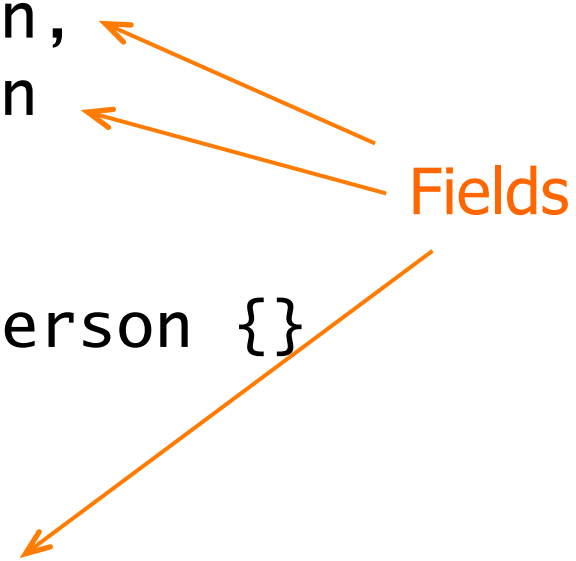
```
sig A1 extends A {}
```

  introduces a set A1 that is a subset of A

Example Signatures and Fields

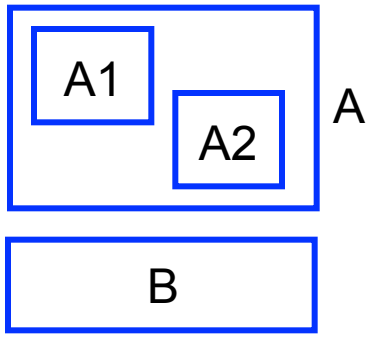
Family Structure:

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

sig Man, Woman extends Person {}

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



Signatures

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

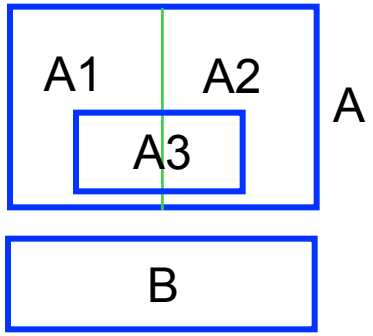


- A1 and A2 are extensions of A
- A signature declared independently of any other one is a top-level signature, e.g., A and B

- 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

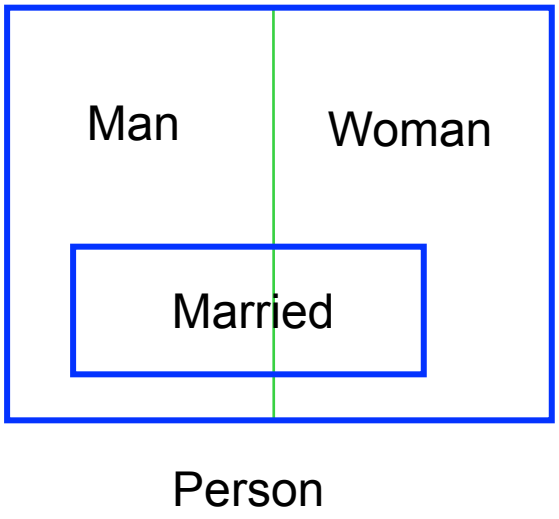
```
sig A3 in A {}
```

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

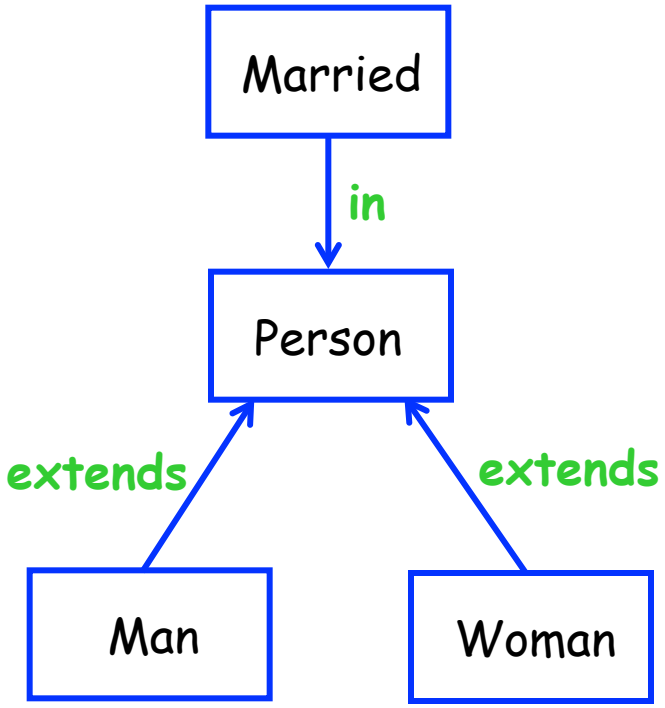
Example: Family Structure

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 see 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 whose domain is A and whose image is given by the expression e.

- Examples:
  - Binary Relation:

```
sig A { f1: A }
```
// f1 is a subset of A x A

  - Ternary Relation:

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

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), (P1)}
Married = {}
Woman = {}

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 element in the signature's set

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

Multiplicities: examples

- Without multiplicity:
  - A set of pixels, each of which is red, green or blue

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

- With multiplicity:
  - An enumeration of color

```
abstract sig Color {}
one sig Red, Yellow, Green extends 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 keyword, if omitted, 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:

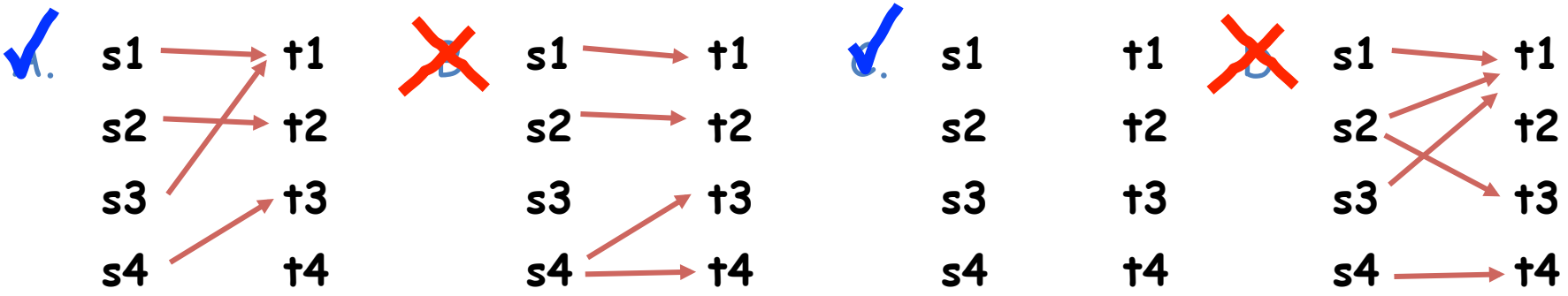
City = {(Iowa City), (Chicago)}
Rainy = {(rainy)}
Sunny = {(sunny)}
Cloudy = {(cloudy)}
Forecast = {(f1), (f2)}
weather = { (f1, Iowa City, rainy), (f1, Chicago, rainy),
(f2, Iowa City, rainy), (f2, Chicago, sunny) }

Multiplicities and Binary Relations

- `sig S {f: lone T}`
  - says that, for each element s of S, f maps s to <u>at most</u> a single value in T

Conventional name: partial function

- Potential instances:

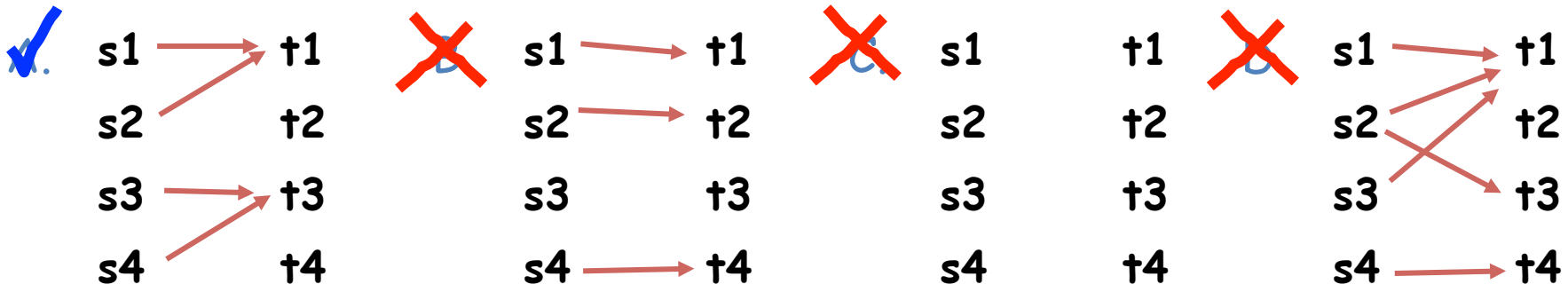


Multiplicities and Binary Relations

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

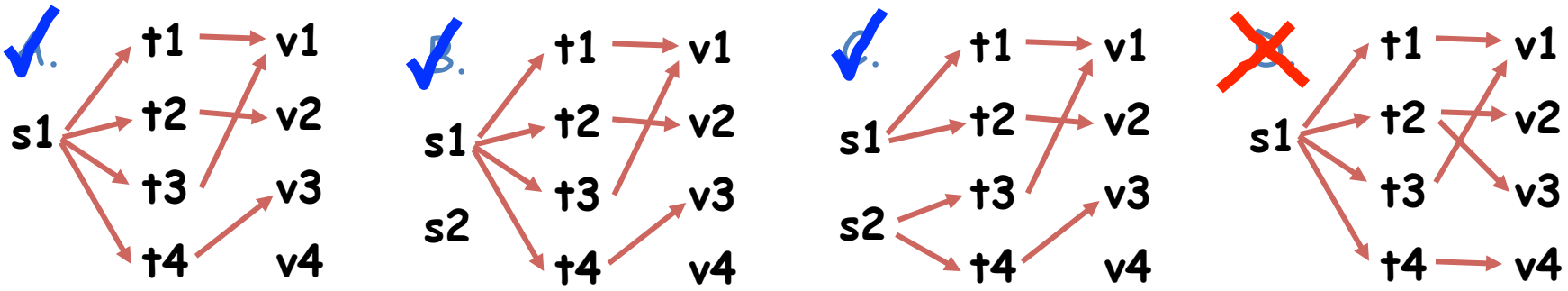
Conventional name: total function

- Potential instances:



Multiplicities and Ternary Relations

- `sig S {f: T -> one V}`
  - For each element s of S, the following holds:
    Within s.S,
    f maps each T-element to to exactly one V-element
- Potential instances:



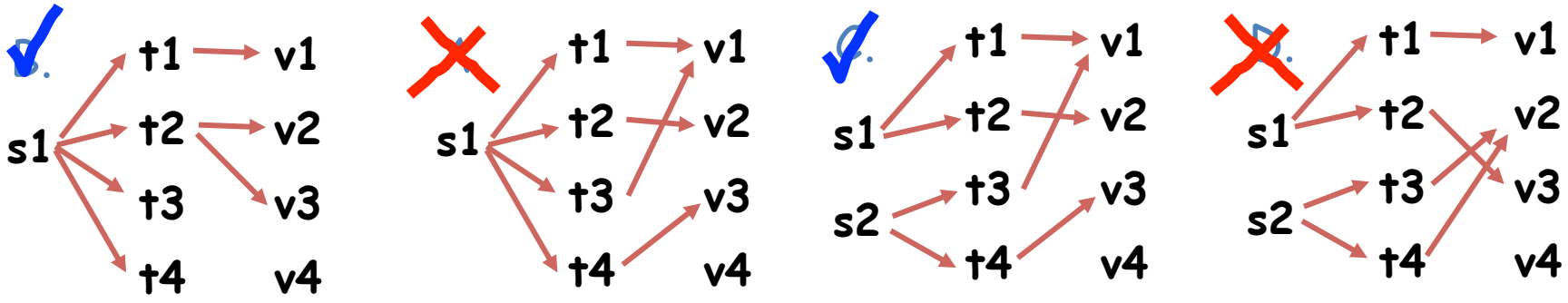
Multiplicities and Ternary Relations

- `sig S {f: T lone -> V}`
  - For each element s of S, the following holds:

    Within s.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}`
  - …

Cardinality Constraints

- Multiplicities can also be applied to whole 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

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)}
```

Person can be their 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)}
```

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)}
```

Self-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)}
```

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)}
```

"where is (Man2,Man1) ?"

Child-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)}
```

Asymmetric marriage?

```
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 match our domain knowledge
  - We can add constraints to enrich the model
- Underconstrained models are common early in the development process
  - AA gives us 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 be their own parent (or more generally, their own ancestor)
  - No person can have more than one father or mother
  - A person's siblings are those with the same parents

Adding Constraints

- We'd like to enforce the following social constraints
  - The spouse relation is symmetric
  - A man's wife cannot be one of his siblings

Predefined Sets

- There are three constants:
  - `none` : empty set
  - `univ` : universal set
  - `ident` : identity
- Example. For a model with just the two sets:

```
Man = {(M0),(M1),(M2)}
Woman = {(W0),(W1)}
```

the constants have the values

```
none = {}
univ = {(M0),(M1),(M2),(W0),(W1)}
ident ={(M0,M0),(M1,M1),(M2,M2),(W0,W0),(W1,W1)}
```

Quantifiers

- Alloy includes 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 fails for every x in S
  - `lone x: S | F` F holds for at most 1 x in S
  - `one x: S | F` F holds for exactly 1 x in S

Everything is a Set in Alloy

- There are no scalars
  - We never speak directly about elements (or tuples) of relations
  - Instead, we always use singleton relations:

```
let matt = one Person
```

- When we have quantification:

```
all x : S | … x …
```

`x = {t}` for some element `t` of `S`

Set Operators

- Set operators
  - `+` : union
  - `&` : intersection
  - `-` : difference
  - `in` : subset
  - `=` : equality

- Ex: Married men,

```
Married & Man
```

Relational Operators

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

Arrow Product

- `p -> q`
  - p and q are two relations
  - p->q is the relation you get by taking every combination of a tuple from p and a tuple from q and concatenating them.

- Examples:

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

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

Transpose

- `~ p`
  - take the mirror image of the relation p,
    i.e. reverse the order of atoms in each tuple.

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

- How would you use `~` to express the parents relation ?

`~children`

How to join tuples ?

- p.q : What is the join of theses two tuples ?
  - $(s_1,...,s_m)$
  - $(t_1,...,t_m)$

  If $s_m \neq t_1$ then the result is empty
  If $s_m = t_1$ then the result is: $(s_1,...,s_{m-1},t_2,...,t_m)$

- Examples :
  - {(a,b)}.{(a,c)} = {}
  - {(a,b)}.{(b,c)} = {(a,c)}

- What about {(a)} . {(a)} ? Not defined !

p.s is defined iff p and s are **not** both unary relations

How to join relations ?

- `p.q`
  - p and q are two relations that are not both unary
  - `p.q` is the relation you get by taking every combination of a tuple from p and a tuple from q and adding their join, if it exists.

Exercises

- What's the result of these join applications?
  - `{(a,b)}.{(c)}`
  - `{(a)}.{(a,b)}`
  - `{(a,b)}.{(b)}`
  - `{(a)}.{(a,b,c)}`
  - `{(a,b,c)}.{(c)}`
  - `{(a,b)}.{(a,b,c)}`
  - `{(a,b,c,d)}.{(d,e,f)}`
  - `{(a)}.{(b)}`

Examples:

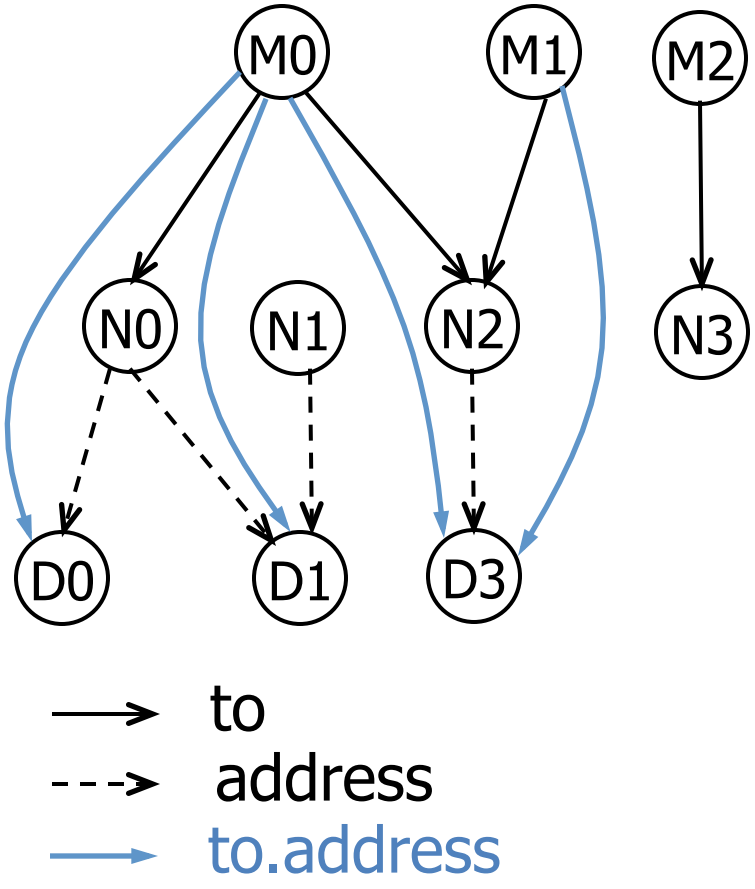
to maps a message to the name it's intended to be send to

address maps names to addresses

- `to = {(M0,N0),(M0,N2)`
  `(M1,N2),(M2,N3)}`
- `address = {(N0,D0),`
  `(N0,D1),(N1,D1),(N2,D3)}`

to.address maps a message to the addresses it should be sent to

- `to.address = {(M0,D0),`
  `(M0,D1),(M0,D3),(M1,D3)}`



Exercises

- Given a relation `addr` of arity 4 that contains the tuple `b->n->a->t` when book `b` maps name `n` to address `a` at time `t`, and a book `b` and a time `t`:
  - `addr = {(B0,N0,D0,T0),(B0,N0,D1,T1), (B0,N1,D2,T0),(B0,N1,D2,T1),(B1,N2,D3,T0), (B1,N2,D4,T1)}`
  - `t = {(T1)}` `b = {(B0)}`

♪

The expression `b.addr.t` is the name-address mapping of book `b` at time `t`. What is the value of `b.addr.t` ?

- When p is a binary relation and q is a ternary relation, what is the arity of the relation p.q ?
- Join is not associative, why ?

  (i.e. (p.q).r and p.(q.r) are not always equivalent)

Example: Family Structure

- How would you use join to find Matt's children or grandchildren ?
  - `matt.children` // Matt's children
  - `matt.children.children` // Matt's grandchildren

- What if we want to find Matt's descendants?

Box Join

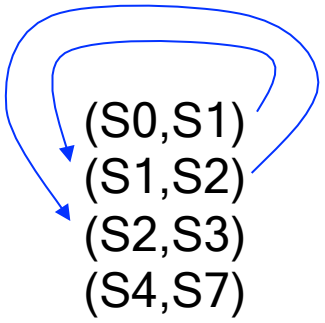
- `p[q]`
  - Semantically identical to dot join, but takes its arguments in different order

`p[q] ≡ q.p`

- Example: Matt's children or grandchildren ?
  - `children[matt]` // Matt's children
  - `children.children[matt]` // Matt's grandchildren
  - `children[children[matt]]` // Matt's grandchildren

Transitive Closure

- ^{r}
  - Intuitively, the transitive closure of a relation r: S x S is what you get when you keep navigating through r until you can't go any farther.



r

(S0,S1)
(S1,S2)
(S2,S3)
(S4,S7)
(S0,S2)
(S0,S3)
(S1,S3)

^{r}

- $^r = r + r.r + r.r.r + \ldots$

Example: Family Structure

- What if we want to find Matt's ancestors or descendants ?
  - `matt.^children` // Matt's descendants
  - `matt.^(~children)` // Matt's ancestors

- How would you express the constraint "*No person can be their own ancestor*"

```
no p: Person | p in p.^(~children)
```

Reflexive-transitive closure

- `*r = ^r + iden`

Domain and image Restrictions

- The restriction operators are used to filter relations to a given domain or image
- If s is a set and r is a relation then
  - s <: r contains tuples of r starting with an element in s
  - r :> s contains tuples of r ending with an element in s
- Example:
  - `Man = {(M0),(M1),(M2),(M3)}`
  - `Woman = {(W0),(W1)}`
  - `children = {(M0,M1),(M0,M2),(M3,W0),(W1,M1)}`
  - `Man <: children = {(M0,M1),(M0,M2),(M3,W0)}`
    `// father-child`
  - `children :> Man = {(M0,M1),(M0,M2),(W1,M1)}`
    `// parent-son`

Override

- p ++ q
  - p and q are two relations of arity two or more
  - the result is like the union between p and q except that tuples of q can replace tuples of p. Any tuple in p that matches a tuple in q starting with the same element is dropped.
  - `p ++ q = p - (domain(q) <: p) + q`
- Example
  - `oldAddr = {(N0,D0),(N1,D1),(N1,D2)}`
  - `newAddr = {(N1,D4),(N3,D3)}`
  - `oldAddr ++ newAddr = {(N0,D0),(N1,D4),(N3,D3)}`

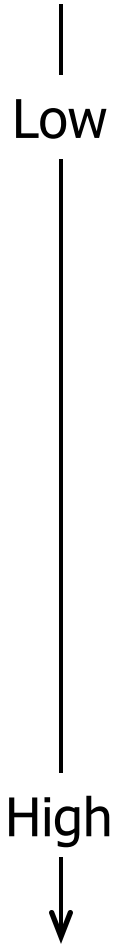
Logical Operators

- The usual logical operators are available
  - `not` `!` negation
  - `and` `&&` conjunction
  - `or` `||` disjunction
  - `implies` `=>` implication
  - `else` alternative
  - `<=>` iff

- Example:
  - `a != b` is equivalent to `not a = b`

Operator Precedence

- ||
- <=>
- =>
- &&
- !
- = != in
- + -
- ++
- &
- ->
- <:
- :>
- []
- .
- ~ * ^



Example: Family Structure

- How would you express the constraint "*No person can have more than one father and mother*" ?

Example: Family Structure

- How would you express the constraint "*No person can have more than one father and mother*" ?

```
all p: Person |
 (lone (p.parents & Man)) and
 (lone (p.parents & Woman))
```

- This is an example of a negative constraint that is easier to state positively (to make use of the `lone` operator).

Set Comprehension

`{ x : S | F }`

– the set of values drawn from set S for which F holds

- How would use the comprehension notation to specify the set of people that have the same parents as Matt?

```
{ q: Person | q.parents = matt.parents }
```

Example: Family Structure

- How would you express the constraint "*A person P's siblings are those people, other than P, with the same parents as P*"

Example: Family Structure

- How would you express the constraint "*A person P's siblings are those people, other than P, with the same parents as P*"

```
all p: Person |
  p.siblings =
    {q: Person | p.parents = q.parents} - p
```

Example: Family Structure

- *Each married man (woman) has a wife (husband)*

- *A spouse can't be a sibling*

Example: Family Structure

- *Every married man (woman) has a wife (husband)*

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

- *A spouse can't be a sibling*

```
no p: Married |
    p.spouse in p.siblings
```

Let

- You can factor expressions out:

```
let x = e | A
```

  - Each occurrence of the variable `x` will be replaced by the expression `e` in `A`

- Example: *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)
```

Facts

- Additional constraints on signatures and fields are expressed in Alloy as facts

- AA looks for instances of a model that also satisfy all its fact constraints

Example Facts

Family Structure:

```
-- No person can be their own ancestor


-- At most one father and mother


-- P's siblings are persons with same parents excluding P
```

Example Facts

Family Structure:

```
-- No person can be their own ancestor
fact selfAncestor {
   no p: Person | p in p.^parents
}

-- At most one father and mother
fact loneParents {
   all p: Person |   lone (p.parents & Man)   and
                     lone (p.parents & Woman)
}

-- P's siblings are persons with same parents excluding P
fact siblingsDefinition {
  all p: Person |
    p.siblings = {q: Person | p.parents = q.parents} - p
}
```

Example Facts

Family Structure:

```
fact social {
   -- Every married man (woman) has a wife (husband)




   -- A spouse can't be a sibling

   -- A person can't be married to a blood relative

}
```

Example Facts

Family Structure:

```
fact social {
   -- Every married man (woman) has a wife (husband)
   all p: Married |
     let s = p.spouse |
       (p in Man => s in Woman) and
       (p in Woman => s in Man)

   -- A spouse can't be a sibling
   no p: Married | p.spouse in p.siblings

   -- A person can't be married to a blood relative
   no p: Married |
     some (p.*parents & (p.spouse).*parents)
}
```

Run Command

- Used to ask AA to generate an instance of the model
- May include conditions
  - Used to guide AA to pick model instances with certain characteristics
  - E.g., force certain sets and relations to be non-empty
  - In this case, not part of the "true" specification

Run Command

- To analyze a model, you add a run command and instruct AA to execute it.
  - the run command

    tells the tool to search for an instance of the model
  - you may also give a scope

    bounds the size of instances that will be considered

- AA executes only the first run command in a file

Scope

- Limits the size of instances considered to make instance finding feasible
- Represents the maximum number of tuples in each top-level signature
- Default value = 3

Run Conditions

- We can use *condition* schemas to encode "realism constraints" to e.g.,
  - Force generated models to include at least one married person, or one married man, etc.
- Later on we'll see that *condition* schemas can be used to implement "constraint macros" – parameterized macros that can be called from other schemas.
  - This allows common constraints to be shared

Run Example

Family Structure:

```
-- The simplest run command
-- The scope is 3
run {}

-- The scope is 4
run {} for 5

-- With conditions, forcing each set to be populated
-- Set the scope to 2
run {some Man && some Woman && some Married} for 2

-- Other scenarios
run {some Woman && no Man} for 7
run {some Man && some Married && no Woman}
```

Exercises

- Load `family-2.als`
- Execute it
- Analyze the metamodel
- Look at the generated instance
- Does it look correct?
- What if anything would you change about it?

Empty Instances

- The analyzer's algorithms prefer smaller instances
  - Often it produces empty or otherwise trivial instances
  - It is useful to know that these instances satisfy the constraints (since you may not want them)
- Usually, they do not illustrate the interesting behaviors that are possible

Exercises

- Load family-3.als
- Execute it
- Look at the generated instance
- Does it look correct?
- How can you produce
  - two married couples?
  - a non empty married relation and a non-empty siblings relation ?

Assertions

- Often we believe that our model entails certain constraints that are not directly expressed
  - e.g., some A & (A in B) entails some B
- We can define these additional constraints as assertions and use the analyzer to check if they hold
  - e.g., assert a1 { some B }
    check a1

Assertions

- If the constraint in an assertion does not hold, the analyzer will produce a counterexample instance.

- If you expect the constraint to hold but it does not, you can either
  - more it into a fact or
  - refine your model until the assertion holds

Assertions

- No person has a parent that is also a sibling.

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

- A person's siblings are his/her siblings' siblings.

```
assert a2 { all p: Person |
              p.siblings = p.siblings.siblings }
```

- No person shares a common ancestor with his/her spouse (i.e., spouse isn't related by blood).

```
assert a3 { no p: Married |
              some (p.^parents & p.spouse.^parents) }
```

Assertion Scopes

- You can specify a scope explicitly for any signature, but:
  - If a signature has been given a bound
  - Then the bound of its supersignature or any other extension of the same supersignature can be determined

Example Scope

```
abstract sig Object {}
sig Directory extends Object {}
sig File extend Object {}
sig Alias extend File {}
```

We consider an assertion A.

- well-formed:

```
check A for 5 Object
check A for 4 Directory, 3 File
check A for 5 Object, 3 Directory
check A for 3 Directory, 3 Alias, 5 File
```

- ill-formed because it leaves the bound of File unspecified

```
check A for 3 Directory, 3 Alias
```

Example Scope

```
abstract sig Object {}
sig Directory extends Object {}
sig File extends Object {}
sig Alias extends File {}
```

- `check A for 5` [or] `run {} for 5`
  - places a bound of 5 on each top-level signature (in this case just `Object`)
- `check A for 5 but 3 Directory`
  - additionally places a bound of 3 on `Directory`, and a bound of 2 on `File` by implication
- `check A for exactly 3 Directory, exactly 3 Alias, 5 File`
  - limits `File` to at most 5 tuples, but requires that `Directory` and `Alias` have exactly 3 tuples each

Scope

- Size determined in a signature declaration has priority on size determined in scope
- Example:

```
abstract sig Color {}
one sig red, yellow, green extends color {}
sig Pixel {color: one Color}

check A for 2
```

  limits the signature `Pixel` to 2 elements, but assigns a size of exactly 3 to `Color`

Exercises

- Load family-4.als
- Execute it
- Look at the generated counter-examples
- Why is SiblingsSibling false?
- Why is NoIncest false?

Problems with Assertions

```
Analyzing SiblingSiblings ...
Scopes: Person(3)
Counterexample found:

  Person = {M,W0,W1}
  Man = {M}
  Woman = {W0,W1}
  Married = {M,W1}

  children = {(W0,W1)}
  siblings = {(M,W0),(W0,M)}
  spouse = {(M,W1),(W1,M)}
```

M.siblings = {W0}
M.siblings.siblings = {M}

Problems with Assertions

```
Analyzing NoIncest ...
Scopes: Person(3)
Counterexample found:

  Person = {M0,M1,W}
  Man = {M0,M1}
  Woman = {W}
  Married = {M1,W}

  children = {(M0,W),(W,M1)}
  siblings = {}
  spouse = {(M1,W),(W,M1)}
```

(M0 is an Ancestor of M1
and
M0 is an ancestor of W)
and
M1 and W are married

Exercises

- Fix the specification
  - If the model is underconstrained, add appropriate constraints
  - If the assertion is not correct, modify it
- Demonstrate that your fixes yield no counter-examples
  - Does varying the scope make a difference?
  - Does this mean that the assertions hold for all models?

Exercises

- Express the notion of “blood relative” (share common ancestor) as a condition parameterized on two singleton sets p and q that holds when p and q have a common ancestor.
- Add an extra group of invariants that add common social constraints on the husband/wife and parent relations
  - A person can’t have children with a blood relative
  - A person can’t be married to a blood relative.

Predicates and Functions

- Can be used as “macros”
  - Can be named and reused in different contexts (facts, assertions and conditions of run)
  - Can be parameterized
  - Used to factor out common patterns
- Predicates are good for:
  - Constraints you don’t want to record as fact
  - Constraints you want to reuse in different contexts
- Functions are good for
  - Expressions you want to reuse in different contexts

Functions

- A named expression, with zero or more arguments and an expression for the result
- Examples:
  - The parents relation

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

  - Sisters

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

  - No person can be their own ancestors or sisters

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

Predicates

- A named constraint, with zero or more arguments
- Predicates are NOT included when analyzing other schemas (e.g., facts or assertions) unless they are referenced by name in the schemas being analyzed
- Example:
  - Two persons are blood relatives iff they have a common ancestor

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

  - A person can't be married to a blood relative

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

Predicate or Fact ?

- Predicates are (parametrized) definitions of constraints
- Facts are assumed constraints
- Note:
  - You can package constraints as predicates and then include the predicates in facts

Exercises

- Define a predicate that characterizes the notion of "in-law" for the family example
- Write an fact stating that a person is an in-law of their in-laws
- Add these to the family example and run it through AA
- Can you express this same notion in another way in the Alloy model?
  - Do so and run it through AA
  - Which approach is better? Why?

Exercises

- Add an assertion stating that a person has no married in-laws
- What is the minimum scope for set Person for which ACA can find a counterexample?
- How would you use ACA to demonstrate that your answer is truly the minimum scope?
- Demonstrate it!

Acknowledgements

- The family structure example is based on an example by Daniel Jackson distributed with the Alloy Analyzer.