272: Software Engineering

Instructor: Tevfik Bultan

Lectures 3 and 4: Alloy and Alloy Analyzer

Object Oriented Modeling with UML

- UML is an object oriented modeling language

- UML allows software developers to specify object oriented designs at a high level of abstraction
  - UML models represent a higher level of abstraction compared to object oriented programs
  - They can be used in documenting the software design before the software is implemented

- UML models do not have a formal semantics
  - This is a major problem since software developers use UML models to document and communicate the design

Object Oriented Modeling with UML+OCL

- Object Constraint Language (OCL) allows software developers to make UML models more precise

- OCL expressions and constraints can be used to reduce the imprecision in UML designs

- One can augment a UML models with constraints written in OCL
  - OCL constraints can be used to write contracts for UML classes
  - Similar to design by contract assertions written as annotations for object oriented programs

- OCL expressions have formal syntax and semantics

Object Oriented Modeling with Alloy

- Alloy is another object oriented modeling language
- Alloy has formal syntax and semantics
- Alloy specifications can be written in ASCII
- Alloy also has a visual language similar to UML class diagrams
- Alloy has a constraint analyzer which can be used to automatically analyze properties of Alloy models

Alloy: A Modeling Language

- Alloy is a formal modeling language

- Alloy has formal syntax and semantics

- Alloy specifications are written in ASCII
  - There is also a visual representation (similar to UML class diagrams and entity-relationship diagrams) but the visual representation does not have the expressiveness of the whole language

- Alloy has a verification tool called Alloy Analyzer which can be used to automatically analyze properties of Alloy models

Alloy: A Modeling Language

- Alloy targets formal specification of object oriented data models

- It can be used for data modeling in general
  - It is good at specifying classes objects, the associations among them, and constraints on those associations

- It is most similar to UML class diagrams combined with OCL (Object Constraint Language)
  - However, it has a simpler and cleaner semantics than UML/OCL and it is also supported by a verification tool (Alloy Analyzer)

Alloy Analyzer

- Alloy Analyzer is a verification tool that analyzes Alloy specifications

- It uses bounded verification
  - It limits the number of objects in each class to a fixed number and checks assertions about the specification within that bound

- It uses a SAT-solver to answer verification queries
  - It converts verification queries to satisfiability of Boolean logic formulas and calls a SAT solver to answer them

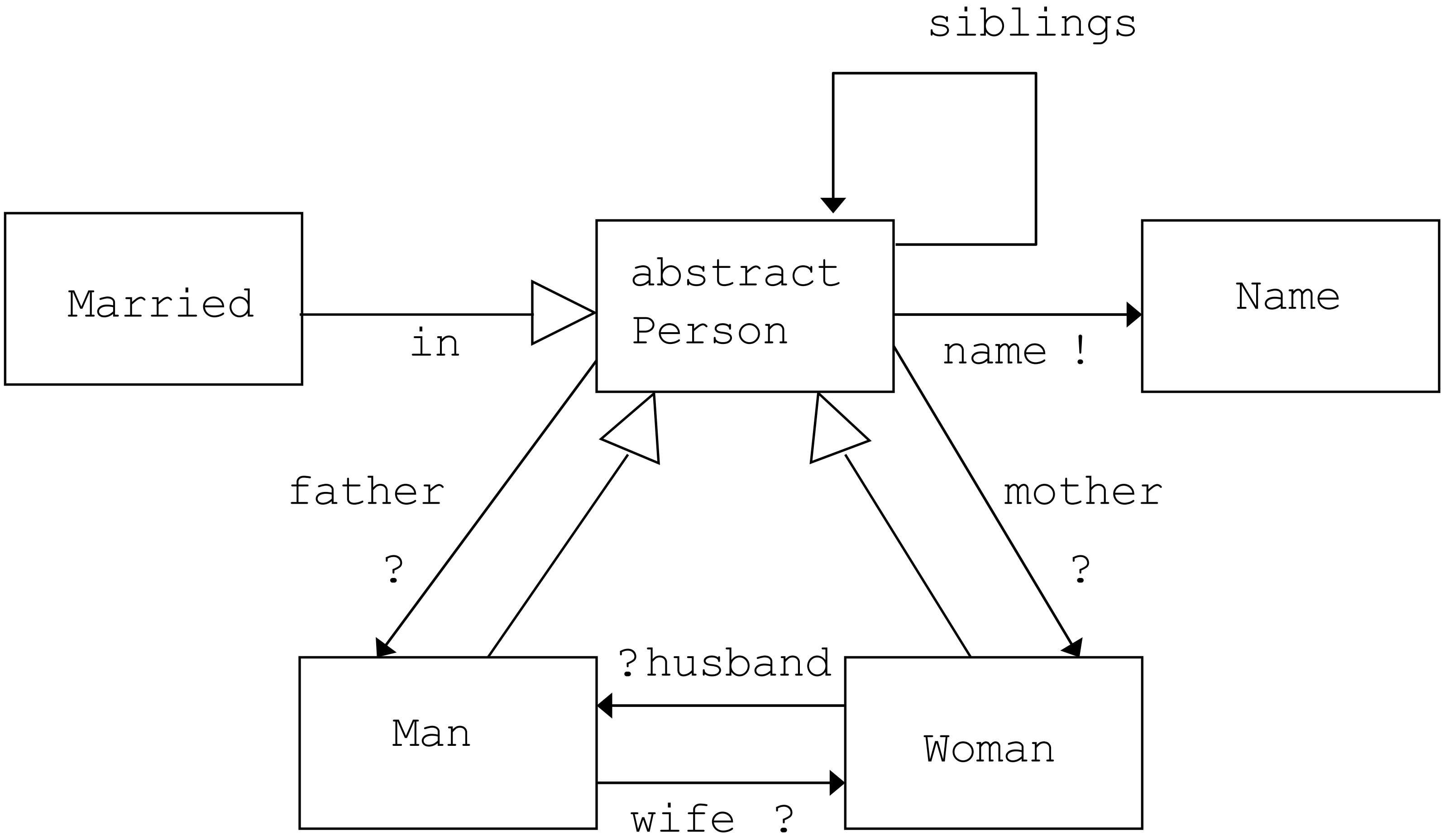
Alloy and Alloy Analyzer

- Alloy and Alloy Analyzer were developed by Daniel Jackson's group at MIT

- References
  - "Alloy: A Lightweight Object Modeling Notation"
    Daniel Jackson, ACM Transactions on Software Engineering and Methodology (TOSEM), Volume 11, Issue 2 (April 2002), pp. 256-290.
  - "Software Abstractions: Logic, Language and Analysis" by Daniel Jackson. MIT Press, 2006.

- Unfortunately, the TOSEM paper is based on the old syntax of Alloy
  - The syntax of the Alloy language is different in the more recent versions of the tool
  - Documentation about the current version of Alloy is available here: http://alloy.mit.edu/
  - My slides are based on the following tutorial
    http://alloy.mit.edu/alloy/tutorials/online

An Alloy Object Model for a Family Tree



Basics of Alloy Semantics

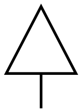
- Each box denotes a set of objects (atoms)
  - Corresponds to an object class in UML/OCL
  - In Alloy these are called signatures

- An object is an abstract, atomic and unchanging entity

- The state of the model is determined by
  - the relationships among objects and
  - the membership of objects in sets
  - these can change in time

Subclasses are subsets

- An arrow with unfilled head denotes a subset
  - `Man`, `Woman`, `Married` are subsets of `Person`
  - This corresponds to sub-classes in UML/OCL



- The keyword **`extends`** indicates disjoint subsets
  - This is the default, if a subset is not labeled, then it is assumed to extend
  - `Man` and `Woman` are disjoint sets (their intersection is empty)
    - There is no `Person` who is a `Woman` and a `Man`
- The keyword **`in`** indicates subsets, not necessarily disjoint from each other (or other subsets that extend)
  - `Married` and `Man` are not disjoint
  - `Married` and `Woman` are not disjoint

Signatures

- In Alloy sets of atoms such as `Man`, `Woman`, `Married`, `Person` are called **signatures**
  - *Signatures correspond to object classes*
- A signature that is not subset of another signature is a top-level signature
- Top-level signatures are implicitly disjoint
  - `Person` and `Name` are top-level signatures
    - They represent disjoint sets of objects
- Extensions of a signature are also disjoint
  - `Man` and `Woman` are disjoint sets
- An abstract signature has no elements except those belonging to its extensions
  - There is no `Person` who is not a `Man` or a `Woman`

Class associations are relations

- Arrows with a small filled arrow head denote relations ↑
- For example, `name` is a relation that maps `Person` to `Name`
- Relations are expressed as fields of signatures
  - These correspond to associations in UML-OCL
  - They express relations between object classes

Multiplicity

- Markings at the ends of relation arrows denote the multiplicity constraints
  - `*` means zero or more (default, keyword `set`)
  - `?` means zero or one (keyword `lone`)
  - `!` means exactly one (keyword `one`)
  - `+` means one or more (keyword `some`)
  - If there is no marking, the multiplicity is `*`

- `name` maps each `Person` to exactly one `Name` (based on the mark at the `Name` end of the arrow denoting the `name` relationship)

- `name` maps zero or more members of `Person` to each `Name` (based on the omission of the mark at the `Person` end)

Textual Representation

- Alloy is a textual language
  - The graphical notation is just a useful way of visualizing the specifications but it is not how you write an Alloy model

- The textual representation represents the Alloy model completely
  - i.e., the graphical representation is redundant, it can be used to visualize a model but it is not used to specify a model

Alloy Object Model for a Family Tree

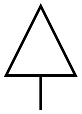
```
module language/Family
sig Name { }
abstract sig Person {
  name: one Name,
  siblings: Person,
  father: lone Man,
  mother: lone Woman
  }
sig Man extends Person {
  wife: lone Woman
  }
sig Woman extends Person {
  husband: lone Man
  }
sig Married in Person {
  }
```

Signatures

- Textual representation starts with **`sig`** declarations defining the signatures (sets of atoms)
  - You can think of signatures as object classes, each signature represents a set of objects

- Multiplicity:

`set`	zero or more
`one`	exactly one
`lone`	zero or one
`some`	one or more

- **`extends`** and **`in`** are used to denote which signature is subset of which other signature
  - Corresponding to arrow with unfilled head 
  - **`extends`** denotes disjoint subsets

Signatures

```
sig A {}
```

set of atoms A

```
sig A {}
sig B {}
```

intersection

disjoint sets A and B. As an Alloy expression we can write: `no A & B`

(Alloy expressions are discussed in later slides)

```
sig A, B {}
```

same as above

subset

```
sig B extends A {}
```

set B is a subset of A. As an Alloy epxression: `B in A`

logical and

```
sig B extends A {}
sig C extends A {}
```

B and C are disjoint subsets of A: `B in A && C in A && no B & C`

```
sig B, C extends A {}
```

same as above

Signatures

```
abstract sig A {}
sig B extends A {}
sig C extends A {}
```

union

A partitioned by disjoint subsets B and C: `no B & C && A = (B + C)`

```
sig B in A {}
```

B is a subset of A, not necessarily disjoint from any other set

```
sig C in A + B {}
```

C is a subset of the union of A and B: `C in A + B`

```
one sig A {}
lone sig B {}
some sig C {}
```

A is a singleton set
B is a singleton or empty
C is a non-empty set

Fields are Relations

- The fields define relations among the signatures
  - Similar to a field in an object class that establishes a relation between objects of two classes
  - Similar to associations in UML/OCL

- Visual representation of a field is an arrow with a small filled arrow head ↑

Fields Are Relations

`sig` `A {f: e}`

f is a binary relation with domain A and range given by expression e

each element of A is associated with exactly one element from e

(i.e., the default cardinality is one)

```
all a: A | a.f: one e
```

```
sig A {
    f1: one e1,
    f2: lone e2,
    f3: some e3,
    f4: set e4
    }
```

Multiplicities correspond to the following constraint, where `m` could be `one`, `lone`, `some`, or `set`

```
all a: A | a.f : m e
```

Fields

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

two fields with the same constraint

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

a field can declare a ternary relation, each tuple in the relation f has three elements (one from A, one from e1 and one from e2), m and n denote the cardinalities of the sets

```
all a: A | a.f : e1 m -> n e2
```

```
sig AdressBook {
    names: set Name,
    addrs: names -> Addr
  }
```

In definition of one field you can use another field defined earlier (these are called dependent fields)

```
(all b: AddressBook | b.addrs: b.names -> Addr)
```

Alloy Object Model for a Family Tree

```
module  language/Family
sig Name  { }
abstract  sig Person  {
  name:  one  Name,
  siblings:  Person,
  father:  lone  Man,
  mother:  lone  Woman
  }
sig Man extends  Person  {
  wife:  lone  Woman
  }
sig Woman extends  Person  {
  husband:  lone Man
  }
sig Married  extends  Person  {
  }
fact  {
  no p:  Person  | p  in  p.^(mother  +  father)
  wife  =  ~husband
}
```

transitive closure

transpose

Facts

- After the signatures and their fields, facts are used to express constraints that are assumed to always hold

- Facts are not assertions, they are constraints that restrict the model
  - Facts are part of our specification of the system
  - Any configuration that is an instance of the specification has to satisfy all the facts

Facts

`fact { F }`

`fact f { F }`

Facts can be written as separate paragraphs and can be named.

`Sig A { ... }{ F }`

Facts about a signature can be written immediately after the signature

Signature facts are implicitly quantified over the elements of the signature

It is equivalent to:

`fact {all a: A | F’}`

where any field of A in F is replaced with a.field in F’

Facts

```
sig Host {}

sig Link {from, to: Host}

fact {all x: Link | x.from != x.to}
```

no links from a host to itself

```
fact noSelfLinks {all x: Link | x.from != x.to}
```

same as above

```
sig Link {from, to: Host} {from != to}
```

same as above, with implicit 'this.'

Functions

```
fun f[x1: e1, ..., xn: en] : e { E }
```

- A function is a named expression with zero or more arguments
  - When it is used, the arguments are replaced with the instantiating expressions

```
fun grandpas[p: Person] : set Person {
  p.(mother + father).father
  }
```

Predicates

```
pred p[x1: e1, ..., xn: en] { F }
```

- A predicate is a named constraint with zero or more arguments
  - When it is used, the arguments are replaced with the instantiating expressions

```
fun grandpas[p: Person] : set Person {
  let parent = mother + father + father.wife +
  mother.husband | p.parent.parent & Man
  }

pred ownGrandpa[p: Person] {
  p in grandpas[p]
  }
```

Assertions

```
assert a { F }
```

Assertions are constraints that were intended to follow from facts of the model

You can use Alloy analyzer to check the assertions

```
sig Node {
  children:  set Node
  }
one sig Root extends Node {}
fact {
```

reflexive transitive closure

```
  Node in Root.*children
  }
// invalid assertion:
assert someParent {
  all n: Node | some children.n
  }
// valid assertion:
assert someParent {
  all n: Node - Root | some children.n
  }
```

Assertions

- In Alloy, assertions are used to specify properties about the specification
- Assertions state the properties that we expect to hold
- After stating an assertion we can check if it holds using the Alloy analyzer (within a given scope)

Check command

```
assert a { F }
check a scope
```

- Assert instructs Alloy analyzer to search for counterexample to assertion within scope
  - Looking for counter-example means looking for a solution to
    `M && !F` where M is the formula representing the model

```
check a
```

top-level sigs bound by 3

```
check a for default
```

top-level sigs bound by default

```
check a for default but list
```

default overridden by bounds in list

```
check a for list
```

sigs bound in list

Check Command

```
abstract sig Person {}
sig Man extends Person {}
sig Woman extends Person {}
sig Grandpa extends Man {}
check a
check a for 4
check a for 4 but 3 Woman
check a for 4 but 3 Man, 5 Woman
check a for 4 Person
check a for 4 Person, 3 Woman
check a for 3 Man, 4 Woman
check a for 3 Man, 4 Woman, 2 Grandpa
```

Check Example

```
fact {
  no p: Person | p in p.^(mother + father)
  no (wife + husband) & ^(mother + father)
  wife = ~husband
  }
assert noSelfFather {
  no m: Man | m = m.father
  }
check noSelfFather
```

Run Command

```
pred p[x: X, y: Y, ...] { F }
run p scope
```

Instructs analyzer to search for instance of a predicate within scope

If the model is represented with formula M, run finds solution to

```
M && (some x: X, y: Y, ... | F)
```

```
fun f[x: X, y: Y, ...] : R { E }
run f scope
```

Instructs analyzer to search for instance of function within scope

If model is represented with formula M, run finds solution to

```
M && (some x: X, y: Y, ..., result: R | result = E)
```

Alloy Object Model for a Family Tree

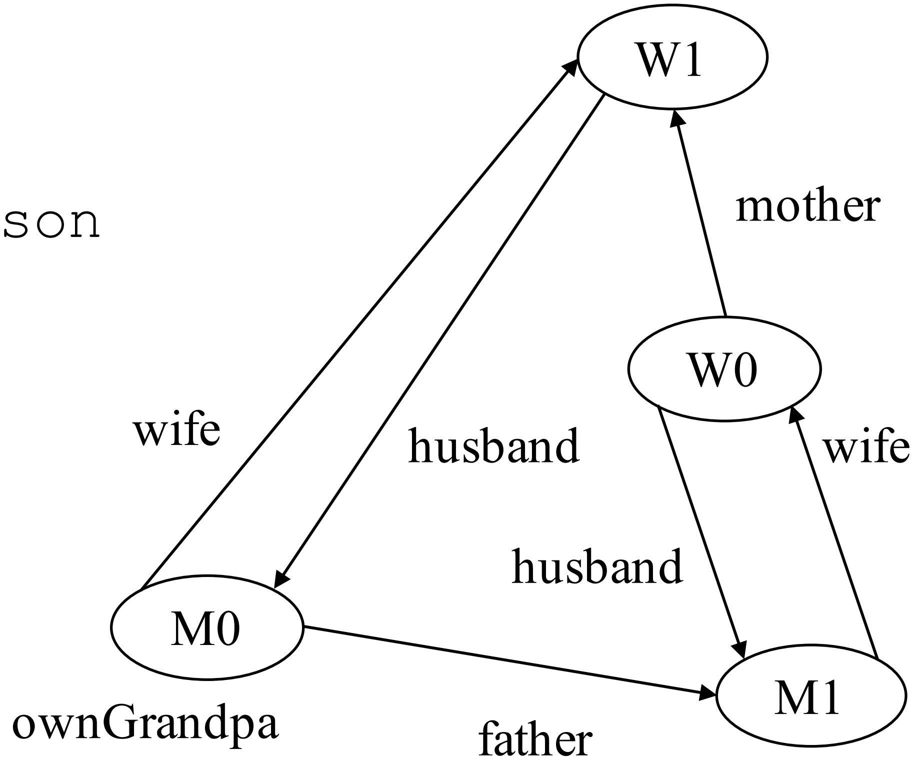
```
module language/Family
sig Name { }
abstract sig Person {
  name: one Name,
  siblings: Person,
  father: lone Man,
  mother: lone Woman
  }
sig Man extends Person {
  wife: lone Woman
  }
sig Woman extends Person {
  husband: lone Man
  }
sig Married extends Person {
  }
fact {
  no p: Person | p in p.^(mother + father)
  no (wife + husband) & ^(mother + father)
  wife = ~husband
}
```

Predicate Simulation

```
fun grandpas[p: Person] : set Person {
  let parent = mother + father + father.wife +
  mother.husband | p.parent.parent & Man
  }
pred ownGrandpa[p: Person] {
  p in grandpas[p]
  }
run ownGrandpa for 4 Person
```

Predicate Simulation

```
fun grandpas[p: Person] : set Person {
  let parent = mother + father + father.wife +
  mother.husband | p.parent.parent & Man
  }
pred ownGrandpa[p: Person] {
  p in grandpas[p]
  }
run ownGrandpa for 4 Person
```



Alloy Expressions

- Expressions in Alloy are expressions in Alloy’s logic

- atoms are Alloy's primitive entities
  - indivisible, immutable, uninterpreted

- relations associate atoms with one another
  - set of tuples, tuples are sequences of atoms

- every value in Alloy logic is a relation!
  - relations, sets, scalars are all the same thing

Everything is a relation

sets are unary (1 column) relations

```
Person = {(P0), (P1), (P2)}
Name = {(N0), (N1), (N2), (N3)}
```

scalars are singleton sets

```
myName = {(N1)}
yourName = {(N2)}
```

binary relation

```
name = {(P0, N0), (P1, N0), (P2, N2)}
```

Alloy also allows relations with higher arity (like ternary relations)

Constants

`none`	*empty set*
`univ`	*universal set*
`iden`	*identity relation*

```
Person = {(P0), (P1), (P2)}
Name = {(N0), (N1), (N2), (N3)}
none = {}
univ = {(P0), (P1), (P2), (N0), (N1), (N2), (N3)}
iden = {(P0, P0),(P1, P1), (P2, P2), (N0, N0), (N1,
  N1), (N2, N2),(N3,N3) }
```

Set Declarations

`x: m e` x is a subset of e and its cardinality (size) is restricted to be m

`m` can be:

`set`	any number
`one`	exactly one (default)
`lone`	zero or one
`some`	one or more

`x: e` is equivalent to `x: one e`

`SomePeople: set Person`

SomePeople is a subset of the set Person

Set Operators

+	*union*
&	*intersection*
–	*difference*
`in`	*subset*
=	*equality*

Product Operator

`->` *cross product*

```
Person = {(P0), (P1)}
Name = {(N0), (N1)}
Address = {(A0)}

Person -> Name =
    {(P0, N0), (P0, N1), (P1, N0), (P1, N1)}

Person -> Name -> Adress =
  {(P0, N0, A0), (P0, N1, A0), (P1, N0, A0),
   (P1, N1, A0)}
```

Relation Declarations with Multiplicity

`r: A m -> n B` *cross product with multiplicity constraints*

m and n can be one, lone, some, set

`r: A -> B` is equivalent to (default multiplicity is set)

```
r: A set -> set B
```

`r: A m -> n B` is equivalent to:

```
r: A -> B
all a: A | n a.r
all b: B | m r.b
```

Relation Declarations with Multiplicity

`r: A -> one B`

r is a function with domain A

`r: A one -> B`

r is an injective relation with range B

`r: A -> lone B`

r is a function that is partial over the domain A

`r: A one -> one B`

r is an injective function with domain A and range B (a bijection from A to B)

`r: A some -> some B`

r is a relation with domain A and range B

Relational Join (aka navigation)

`p.q`

dot is the relational join operator

Given two tuples (p_1, …, p_n) in p and (q_1, …, q_m) in q where $p_n = q_1$

p.q contains the tuple (p_1, …, p_{n-1}, q_2,…,q_m)

```
{(N0)}.{(N0,D0)} = {(D0)}
{(N0)}.{(N1,D0)} = {}
{(N0)}.{(N0,D0),(N0,D1)}} = {(D0),(D1)}
{(N0),(N1)}.{(N0,D0),(N1,D1),(N2,D3)}} = {(D0),(D1)}
{(N0, A0)}.{(A0, D0)} = {(N0, D0)}
```

Box join

`[]`

box join, box join can be defined using dot join

```
e1[e2]  = e2.e1

a.b.c[d]  = d.(a.b.c)
```

Unary operations on relations

~	transpose
^	transitive closure
*	reflexive transitive closure

these apply only to binary relations

```
^r = r + r.r + r.r.r + ...

*r = iden + ^r

wife = {(M0,W1), (M1, W2)}
~wife = husband = {(W1,M0), (W2, M1)}
```

Relation domain, range, restriction

`domain`	returns the domain of a relation
`range`	returns the range of a relation
`<:`	domain restriction (restricts the domain of a relation)
`:>`	range restriction (restricts the range of a relation)

```
name = {(P0,N1), (P1,N2), (P3,N4), (P4, N2)}
domain(name) = {(P0), (P1), (P3), (P4)}
range(name) = {(N1), (N2), (N4)}

somePeople = {(P0), (P1)}
someNames = {(N2), (N4)}

name :> someNames = {(P1,N2), (P3,N4), (P4,N2)}

somePeople <: name= {(P0,N1), (P1,N2)}
```

Relation override

`++` override

```
p ++ q = p - (domain(q) <: p) + q
```

```
m' = m ++ (k -> v)
```

update map m with key-value pair (k, v)

Boolean operators

`!`	`not`	negation
`&&`	`and`	conjunction
`||`	`or`	disjunction
`=>`	`implies`	implication
	`else`	alternative
`<=>`	`iff`	bi-implication

four equivalent constraints:

```
F => G else H
F implies G else H
(F && G) || ((!F) && H)
(F and G) or ((not F) and H)
```

Quantifiers

`all x: e | F`

`all x: e1, y: e2 | F`

`all x, y: e | F`

`all disj x, y: e | F` F holds on distinct x and y

`all`	F holds for every x in e
`some`	F holds for at least one x in e
`no`	F holds for no x in e
`lone`	F holds for at most one x in e
`one`	F holds for exactly one x in e

A File System Model in Alloy

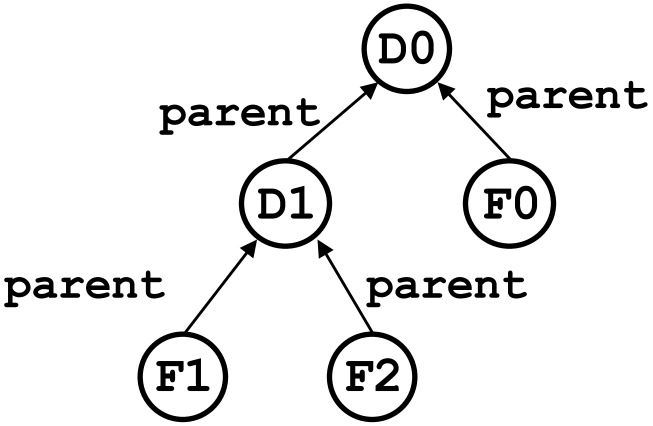
```
// File system objects
abstract sig FSObject { }
sig File, Dir extends FSObject { }

// A File System
sig FileSystem {
  live: set FSObject,
  root: Dir & live,
  parent: (live - root) -> one (Dir & live),
  contents: Dir -> FSObject
}
{
  // live objects are reachable from the root
  live in root.*contents
  // parent is the inverse of contents
  parent = ~contents
}
```

An Instance of the File System Specification

```
FileSystem = {(FS0)}
FSObject = {(F0), (F1), (F2), (F4), (D0), (D1)}
File = {(F0), (F1), (F2), (F4)}
Dir = {(D0), (D1)}

live = {(FS0,F0),(FS0,F1),(FS0,F2),(FS0,D0),(FS0,D1)}
root = {(FS0,D0)}
parent = {(FS0,F0,D0),(FS0,D1,D0),
(FS0,F1,D1),(FS0,F2,D1)}
contents = {(FS0,D0,F0),(FS0,D0,D1),
(FS0,D1,F1),(FS0,D1,F2)}
```



A File System Model in Alloy

```
// Move x to directory d
pred move [fs, fs': FileSystem, x: FSObject, d: Dir]{
  // precondition
  (x + d) in fs.live
  // postcondition
  fs'.parent = fs.parent - x->(x.(fs.parent)) + x->d
}
```

File System Model in Alloy

```
// Delete the file or empty directory x
pred remove [fs, fs': FileSystem, x: FSObject] {
  x in (fs.live - fs.root)
  fs'.root = fs.root
  fs'.parent = fs.parent - x->(x.(fs.parent))
}

// Recursively delete the directory x
pred removeAll [fs, fs': FileSystem, x: FSObject] {
  x in (fs.live - fs.root)
  fs'.root = fs.root
  let subtree = x.*(fs.contents) |
   fs'.parent = fs.parent - subtree->(subtree.(fs.parent))
}
```

File System Model in Alloy

```
// Moving doesn't add or delete any file system objects
moveOkay: check {
  all fs, fs': FileSystem, x: FSObject, d:Dir |
    move[fs, fs', x, d] => fs'.live = fs.live
} for 5

// remove removes exactly the specified file or directory
removeOkay: check {
  all fs, fs': FileSystem, x: FSObject |
    remove[fs, fs', x] => fs'.live = fs.live - x
} for 5
```

File System Model in Alloy

```
// removeAll removes exactly the specified subtree
removeAllOkay: check {
  all fs, fs': FileSystem, d: Dir |
    removeAll[fs, fs', d] =>
      fs'.live = fs.live - d.*(fs.contents)
} for 5

// remove and removeAll has the same effects on files
removeAllSame: check {
  all fs, fs1, fs2: FileSystem, f: File |
    remove[fs, fs1, f] && removeAll[fs, fs2, f] =>
      fs1.live = fs2.live
} for 5
```

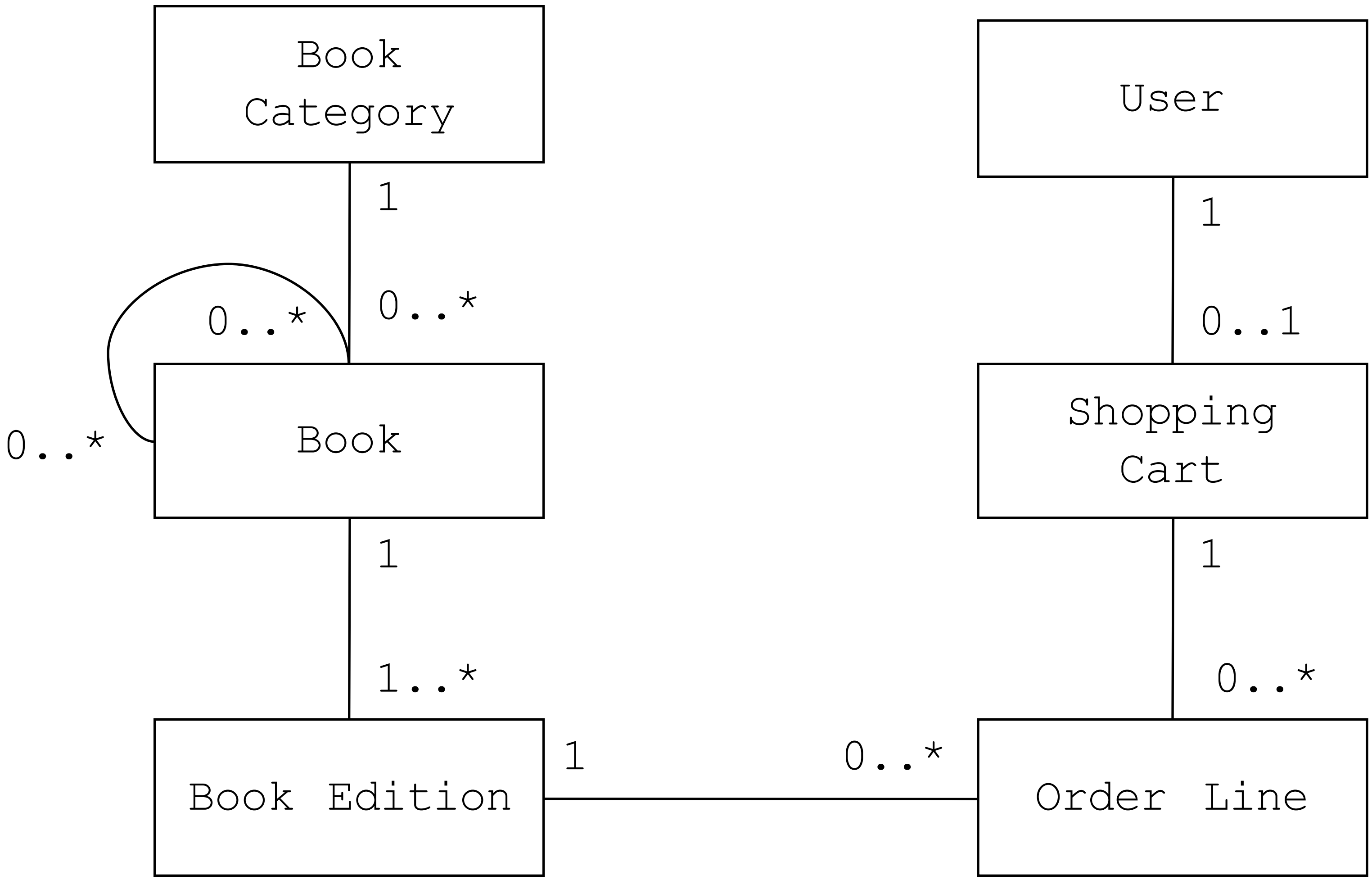
Data Modeling with Alloy

- A natural way to represent the data model for a web application is to use entity-relationship diagrams or UML class diagrams

- Entity-relationship diagrams and UML class diagrams can be converted to Alloy specifications

- Once we write the data model in Alloy we can check assertions about the data model

A Book Store Data Model in UML



Alloy Specification of Book Store Data Model

```
sig BookCategory {
       books:  set Book
}
sig Book {
   category: one BookCategory,
   edition: set BookEdition,
   similar: set Book
}
sig BookEdition {
   book: one Book
}
sig OrderLine {
   order: one BookEdition
}
sig ShoppingCart {
  contents: set OrderLine
}
sig User {
  cart: lone ShoppingCart
}
```

Alloy Specification (Cont.)

```
fact {
  books = ~category
  book = ~edition
  all e1, e2: BookEdition | e1 != e2 => e1.book != e2.book
  all b1, b2: Book | b1 in b2.similar => b1.category = b2.category
  all u1, u2: User | u1.cart = u2.cart => u1 = u2
  all o:OrderLine, c1, c2:ShoppingCart |
               (o in c1.contents && o in c2.contents) => c1 = c2
}

pred addCart[u, u’ : User, o : OrderLine]{
  !(o in u.cart.contents)
  u'.cart.contents = u.cart.contents + o
}

pred removeCart[u, u’ : User, o : OrderLine]{
  o in u.cart.contents
  u'.cart.contents = u.cart.contents - o
}
```

Checking the Alloy Specification

```
assert category {
  all b1, b2 : Book | b1.category != b2.category => b1 !in b2.similar
}

assert category1 {
  no b: Book,  e1, e2:BookEdition | e1 != e2 && e1.book=b && e2.book=b
}

run addCart

run removeCart

run emptyCart

check category

check category1
```

Alloy Kernel

- Alloy is based on a small kernel language
- The language as a whole is defined by the translation to the kernel
- It is easier to define and understand the formal syntax and semantics of the kernel language

Alloy Kernel Syntax

```
formula ::=                     formula syntax
  elemFormula                   elementary formulas
  | compFormula                 compound formulas
  | quantFormula                quantified formulas

elemFormula ::=
  expr in expr                  subset
  expr = expr                   equality

compFormula ::=
  not formula                   negation (not)
  formula and formula           conjunction (and)

quantFormula ::=
 all var : expr | formula  universal quantification
```

```
expr ::=                expression syntax
  rel                   relation
  | var                 quantified variable
  | none                empty set
  | expr binop expr
  | unop expr

binop ::=               binary operators
  +                     union
  | &                   intersection
  | -                   difference
  | .                   join
  | ->                  product

unop ::=                unary operators
  ~                     transpose
  | ^                   transitive closure
```

Alloy Kernel Semantics

- Alloy kernel semantics is defined using denotational semantics

- There are two meaning functions in the semantic definitions
  - M: which interprets a formula as true or false
    - M: Formula, Instance → Boolean
  - E: which interprets an expression as a relation value
    - E: Expression, Instance → RelationValue

- Interpretation is given with respect to an instance that assigns a relational value to each declared relation

- Meaning functions take a formula or an expression and the instance as arguments and return a Boolean value or a relation value

Alloy Kernel Semantics

- To handle the sets and relations in a uniform way Alloy semantics encodes sets also as relations
- Set {x1, x2, ...} is represented as a relation {(unit,x1), (unit,x2), ...}
- Scalar types are singleton sets, i.e., a scalar x1 is represented as {x} which is actually represented as the relation {(unit,x1)}

Alloy Kernel Semantics

M: Formula, Instance → Boolean

Formula Semantics:

$M[p \text{ in } q]i = E[p]i \subseteq E[q]i$

$M[p = q]i = (E[p]i = E[q]i)$

$M[\ !f\ ]i = \neg M[f]i$

$M[f \text{ and } g]i = M[f]i \wedge M[g]i$

$M[\text{all } x{:}e \mid f]i = \wedge\{M[f](i \oplus x \rightarrow v) \mid v \subseteq E[e]i \ \wedge \#v = 1\}$

$i \oplus x \rightarrow v$ is the instance generated by extending i with the binding of variable x to the value v

#v denotes the cardinality of v

Alloy Kernel Semantics

E: Expression, Instance → RelationValue

Expression Semantics:

$E[\text{none}]i = \varnothing$

$E[p+q]i = E[p]i \cup E[q]i$

$E[p\&q]i = E[p]i \cap E[q]i$

$E[p-q]i = E[p]i \setminus E[q]i$

$$E[p.q]i = \{(p_1, \ldots, p_{n-1}, q_2, \ldots, q_m) \mid (p_1, \ldots, p_n) \in E[p]i \wedge (q_1, \ldots, q_m) \in E[q]i \wedge p_n = q_1\}$$

$E[{\sim}p]i = \{(y,x) \mid (x,y) \in E[p]i\}$

$E[\hat{}p]i = \{(x,y) \mid \exists p_1, \ldots \exists p_n, n \geq 0 \mid (x,p_1), (p_1,p_2), \ldots (p_n,y) \in E[p]i\}$

Analyzing Specifications

- Possible problems with a specification
  - The specification is over-constrained: There is no model for the specification
  - The specification is under-constrained: The specification allows some unintended behaviors
- Alloy analyzer has automated support for finding both over-constraint and under-constraint errors

Analyzing Specifications

- Remember that the Alloy specifications define formulas and given an environment (i.e., bindings to the variables in the specification) the semantics of Alloy maps a formula to true or false
- An environment for which a formula evaluates to true is called a model (or instance or solution) of the formula
- If a formula has at least one model then the formula is ***consistent*** (i.e., ***satisfiable***)
- If every (well-formed) environment is a model of the formula, then the formula is ***valid***
- The negation of a valid formula is inconsistent

Analyzing Specifications

- Given a assertion we can check it as follows:
  - Negate the assertion and conjunct it with the rest of the specification
  - Look for a model for the resulting formula, if there exists such a model (i.e., the negation of the formula is consistent) then we call such a model a *counterexample*

- Bad news
  - Validity and consistency checking for Alloy is undecidable
    - The domains are not restricted to be finite, they can be infinite, and there is quantification

Analyzing Specifications

- Alloy analyzer provides two types of analysis:
  - *Simulation,* in which consistency of an invariant or an operation is demonstrated by generating an environment that models it
    - Simulations can be used to check over-constraint errors: To make sure that the constraints in the specification is so restrictive that there is no environment which satisfies them
    - The `run` command in Alloy analyzer corresponds to simulation
  - *Checking,* in which a consequence of the specification is tested by attempting to generate a counter-example
    - The `check` command in Alloy analyzer corresponds to checking
- Simulation is for determining consistency (i.e., satisfiability) and Checking is for determining validity
  - And these problems are undecidable for Alloy specifications

Trivial Example

- Consider checking the theorem

```
all x:X | some y:Y | x.r = y
```

- To check this formula we formulate its negation as a problem

```
r: X -> Y
!all x:X | some y:Y | x.r = y
```

- The Alloy analyzer will generate an environment such as

```
X = {X0, X1}
Y = {Y0, Y1}
r = {(X0, Y0), (X0, Y1)}
x = {X1}
```

which is a model for the negated formula. Hence this environment is a counterexample to the claim that the original formula is valid

The value X1 for the quantified variable x is called a Skolem constant and it acts as a witness to the to the invalidity of the original formula

Sidestepping Undecidability

- Alloy analyzer restricts the simulation and checking operations to a finite scope
  - where a scope gives a finite bound on the sizes of the domains in the specification (which makes everything else in the specification also finite)

- Here is another way to put it:
  - Alloy analyzer rephrases the consistency problem as: Does there exist an environment within the given scope that is a model for the formula
  - Alloy analyzer rephrases the validity problem as: Are all the well-formed environments within the scope a model for the formula
- Validity and consistency problem within a finite scope are decidable problems
  - Simple algorithm: just enumerate all the environments and evaluate the formula on all environments using the semantic function

Simulation: Consistency within a Scope

- If the Alloy analyzer finds a model within a given scope then we know that the formula is consistent!

- On the other hand, if the Alloy analyzer cannot find a model within a given scope does not prove that the formula is inconsistent
  - General problem is is undecidable

- However, the fact that there is no model within a given scope shows that the formula might be inconsistent
  - which would prompt the designer to look at the specification to understand why the formula is inconsistent within that scope

Checking: Validity within a given Scope

- If the formula is not valid within a given scope then we are sure that the formula is not valid
  - Alloy analyzer would generate a counter-example and the designer can look at this counter-example to figure out the problem with the specification.

- On the other hand, the fact that Alloy analyzer shows that a formula is valid within a given scope does not prove that the formula is valid in general
  - Again, the problem is undecidable

- However, the fact that the formula is valid within a given scope gives the designer a lot of confidence about the specification

Alloy Analyzer

- Alloy analyzer converts the simulation and checking queries to boolean satisfiability problems (SAT) and uses a SAT solver to solve the satisfiability problem
- Here are the steps of analysis steps for the Alloy analyzer:
  1. Conversion to negation normal form and skolemization
  2. Formula is translated for a chosen scope to a boolean formula along with a mapping between relational variables and the boolean variables used to encode them. This boolean formula is constructed so that it has a model exactly when the relational formula has a model in the given scope
  3. The boolean formula is converted to a conjunctive normal form, (the preferred input format for most SAT solvers)
  4. The boolean formula is presented to the SAT solver
  5. If the solver finds a model, a model of the relational formula is then reconstructed from it using the mapping produced in step 2

Translation Overview

- In negation normal form only elementary formulas are negated
  - To convert to negation normal form push negations inwards using de Morgan’s laws

- Skolemization eliminates existentially quantified variables.
  - If the existential quantification is not within a universal quantification the quantified variable is replaced with a constant and an additional constraint that such a constant exists
  - If the existential quantification is within a universal quantification the existentially quantified variable is replaced with a function

Translation Overview

- For example

```
!all x: X | some y: Y | x.r=y
```

is converted to

```
some x: X | all y: Y | !x.r=y
```

which is converted to the problem

```
r: X->Y
x: X
all y:Y| !x.r=y
some z:X | z=x
```

Translation Overview

- For example

  `all x: X | some y: Y | x.r=y`

  is converted to

  `all x: X | x.r=y[x]`

  by replacing y with the function

  `y: X->one Y`

- This method generalizes to arbitrary number of universal quantifiers by creating functions indexed by as many types as necessary

Translation Overview

- Once a scope is fixed a value of a relation from S to T can be represented as a bit matrix with a 1 in the ith row of jth column when the ith atom in S is related to the jth atom in T and 0 otherwise
  - Such matrices encode all possible relations from S to T

- Hence, collection of possible values of a relation can be expressed by a matrix of boolean variables

- Any constraint on a relation can be expressed as a formula in these boolean variables and a relational formula as a whole can be similarly expressed by introducing boolean variables for each relational variables

Translation Overview

- For example

  `all y: Y | !x.r=y`

  using a scope of 2 would be translated as follows
- First let’s look at the negation of the formula

  `some y: Y | x.r=y`

- Generate a vector [x0 x1] for x and a matrix [r00 r01, r10 r11] for r

- The expression x.r corresponds to the vector

  $[x0 \wedge r00 \vee x1 \wedge r10 \quad x0 \wedge r01 \vee x1 \wedge r11]$

Translation Overview

- Given,

$$x.r \equiv [x0 \land r00 \lor x1 \land r10 \quad x0 \land r01 \lor x1 \land r11]$$

and $y \equiv [y0 \quad y1]$, we get

$$x.r = y \equiv$$

$$(y0 \leftrightarrow (x0 \land r00 \lor x1 \land r10)) \land (y1 \leftrightarrow (x0 \land r01 \lor x1 \land r11))$$

$$\land (y0 \land \neg y1 \lor \neg y0 \land y1)$$

- Then the boolean logic translation for `some y: Y | x.r=y` is

$$\text{true} \leftrightarrow (x0 \land r00 \lor x1 \land r10) \land \text{false} \leftrightarrow (x0 \land r01 \lor x1 \land r11)$$

$$\lor \ \text{false} \leftrightarrow (x0 \land r00 \lor x1 \land r10) \land \text{true} \leftrightarrow (x0 \land r01 \lor x1 \land r11)$$

$$\equiv (x0 \land r00 \lor x1 \land r10) \land \neg (x0 \land r01 \lor x1 \land r11)$$

$$\lor \neg (x0 \land r00 \lor x1 \land r10) \land (x0 \land r01 \lor x1 \land r11)$$

Translation Overview

- Hence, the formula `some y: Y | x.r=y` is satisfiable within a scope of 2 if and only if the following boolean logic formula is satisfiable

$$(x0 \wedge r00 \vee x1 \wedge r10) \wedge \neg (x0 \wedge r01 \vee x1 \wedge r11)$$
$$\vee \neg (x0 \wedge r00 \vee x1 \wedge r10) \wedge (x0 \wedge r01 \vee x1 \wedge r11)$$

- Note that we can also generate the boolean logic formula for checking the satisfiability of

`all y: Y | !x.r=y ≡ ¬ (some y: Y | x.r=y)`

within the scope of 2 by negating the boolean logic formula above:

$$\neg((x0 \wedge r00 \vee x1 \wedge r10) \wedge \neg (x0 \wedge r01 \vee x1 \wedge r11)$$
$$\vee \neg (x0 \wedge r00 \vee x1 \wedge r10) \wedge (x0 \wedge r01 \vee x1 \wedge r11))$$

Translation Overview

- The generated boolean satisfiability problem (SAT) is an NP-complete problem

- Alloy analyzer implements an efficient translation in the sense that the problem instance presented to the SAT solver is as small as possible
  - It will take the SAT solver exponential time in the worst case to solve the boolean satisfiability problem