Alloy

(Based on [Daniel Jackson 2006] and Alloy MIT Website)

Jen-Feng Shih

SVVRL
Dept. of Information Management
National Taiwan University

December 10, 2009

Outline

What is Alloy

- Alloy is a structural modelling language based on first-order logic, for expressing complex structural constraints and behaviour.
- The Alloy Analyzer is a constraint solver that provides fully automatic simulation and checking.
- Developed by the Software Design Group at MIT.

How is Alloy Related to Z and OCL

- Alloy can be viewed as a subset of Z.
- Unlike Z, Alloy is first order, which makes it analyzable (but also less expressive).
- Alloy is a pure ASCII notation and doesn't require special typesetting tools.
- Alloy is similar to OCL, the Object Language of UML, but it has a more conventional syntax and a simpler semantics, and is designed for automatic analysis.

Alloy = Logic + Language + Analysis

- Logic
  - first order logic + relational calculus
- Language
  - syntax for structuring specifications in the logic
- Analysis
  - bounded exhaustive search for counterexample to a claimed property using SAT

Example

- A birthday book...
  - Associates birthday with shorter names that are more convenient to use.
  - alias: a nickname.
  - group: an entire set of friends.

Outline

Three Logics in One

- Predicate calculus style
  Relation names are used as predicates and tuples formed from quantified variables.

  all n: Name, d, d': Date |
  n -> d **in** birthday **and** n -> d' **in** birthday **implies** d = d'

- Navigation expression style (the most expressive)
  Expressions denote sets, which are formed by "navigating" from quantified variables along relations.

  all n: Name | **lone** n.birthday

- Relational calculus style
  Expressions denote relations, and there are no quantifiers at all.

  no ~birthday.birthday - **iden**

Atoms and Relations

- Atoms are Alloy's primitive entities
  - indivisible, immutable, uninterpreted
- Relations associate atoms with one another
  - consists of a set of tuples, each tuple being a sequence of atoms
  - all relations are first-order, relations cannot contain relations
- Every value in Alloy logic is a relation
  - relations, sets, scalars all the same thing

Everything Is a Relation

- *Sets* are unary relations

$$\begin{aligned} \text{Name} &= \{(\text{N0}), (\text{N1}), (\text{N2})\} \\ \text{Date} &= \{(\text{D0}), (\text{D1}), (\text{D2})\} \\ \text{Book} &= \{(\text{B0}), (\text{B1})\} \end{aligned}$$

- *Scalars* are singleton sets (unary relation with only one tuple)

$$\begin{aligned} \text{myName} &= \{(\text{N0})\} \\ \text{yourName} &= \{(\text{N2})\} \\ \text{myBook} &= \{(\text{B0})\} \end{aligned}$$

- Binary relation

$$\text{name} = \{(\text{B0}, \text{N0}), (\text{B1}, \text{N0}), (\text{B2}, \text{N2})\}$$

- Ternary relation

$$\begin{aligned} \text{birthdays} = \{&(\text{B0}, \text{N0}, \text{D0}), (\text{B0}, \text{N1}, \text{D1}), \\ &(\text{B1}, \text{N1}, \text{D2}), (\text{B1}, \text{N2}, \text{D2})\} \end{aligned}$$

Constants

none	empty set
univ	universal set
iden	identity

Example

Name = {(N0), (N1), (N2)}
Date = {(D0), (D1)}

none = {}
univ = {(N0), (N1), (N2), (D0), (D1)}
iden = {(N0, N0), (N1, N1), (N2, N2), (D0, D0), (D1, D1)}

Set Operators

+	union
&	intersection
-	difference
in	subset
=	equality

Example

Name = {(N0), (N1), (N2)}
Alias = {(N1), (N2)}
Group = {(N0)}
RecentlyUsed = {(N0), (N2)}

Alias + Group = {(N0), (N1), (N2)}
Alias & RecentlyUsed = {(N2)}
Name - RecentlyUsed = {(N1)}
RecentlyUsed **in** Alias = false
RecentlyUsed **in** Name = true
Name = Group + Alias = true

Product Operator

-> cross product

Example

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

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

Relational Join

$$\texttt{p.q} \equiv \boxed{\begin{matrix}(a,\ b)\\(a,\ c)\\(b,\ d)\end{matrix}} \; . \; \boxed{\begin{matrix}(a,\ d,\ c)\\(b,\ c,\ c)\\(c,\ c,\ c)\\(b,\ a,\ d)\end{matrix}} = \boxed{\begin{matrix}(a,\ c,\ c)\\(a,\ a,\ d)\end{matrix}}$$

$$\texttt{x.f} \equiv \boxed{(c)} \; . \; \boxed{\begin{matrix}(a,\ b)\\(b,\ d)\\(c,\ a)\\(d,\ a)\end{matrix}} = \boxed{(a)}$$

Join Operators

.	dot join
[]	box join

```
e1[e2]      = e2.e1
a.b.c[d]    = d.(a.b.c)
```

Example

Book = {(B0)}
Name = {(N0), (N1), (N2)}
Date = {(D0), (D1), (D2)}

myName = {(N1)}
myBirth = {(D0)}

birthday = {(B0, N0, D0), (B0, N1, D0), (B0, N2, D2)}

Book.birthday = {(N0, D0), (N1, D0), (N2, D2)}
Book.birthday[myName] = {(D0)}
Book.birthday.myName = {}

Unary Operators

~	transpose
^	transitive closure
*	reflexive transitive closure
	(apply only to binary relations)

^r = r + r.r + r.r.r + ...
*r = **iden** + ^r

Example

Node = {(N0), (N1), (N2), (N3)}
first = {(N0)}   next = {(N0, N1), (N1, N2), (N2, N3)}

~next = {(N1, N0), (N2, N1), (N3, N2)}
^next = {(N0, N1), (N0, N2), (N0, N3),
(N1, N2), (N1, N3), (N2, N3)}
*next = {(N0, N0), (N0, N1), (N0, N2), (N0, N3), (N1, N1),
(N1, N2), (N1, N3), (N2, N2), (N2, N3), (N3, N3)}

first.^next = {(N1), (N2), (N3)}
first.*next = Node

Restriction and Override

<:	domain restriction
:>	range restriction
++	override

$$p \text{ ++ } q = p - (\mathrm{domain}[q] <: p) + q$$

Example

Name = {(N0), (N1), (N2)}
Alias = {(N0), (N1)} Date = {(D0)}
birthday = {(N0, N1), (N1, N2), (N2, D0)}

birthday :> Date = {(N2, D0)}
Alias <: birthday = {(N0, N1), (N1, N2)} = birthday :> Name
birthday :> Alias = {(N0, N1)}

birthday' = {(N0, N1), (N1, D0)}
birthday ++ birthday' = {(N0, N1), (N1, D0), (N2, D0)}

Boolean Operators

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

Example

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	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

all x: e | F
all x: e1, y: e2 | F
all x, y: e | F
all disj x, y: e | F

Example

some n: Name, d: Date | d **in** n.birthday
some name maps to some birthday - birthday book not empty

no n: Name | n **in** n.^birthday
no name can be reached by lookups from itself - birthday book acyclic

all n: Name | **lone** d: Date | d **in** n.birthday
every name maps to at most one birthday - birthday book is functional

all n: Name | **no disj** d, d': Date | (d + d') **in** n.birthday
no name maps to two or more distinct birthday - same as above

Quantified Expressions

some e	e has *at least one* tuple
no e	e has *no* tuples
lone e	e has *at most one* tuple
one e	e has *exactly one* tuple

Example

some Name

set of names is not empty

some birthday

birthday book is not empty - it has a tuple

no (birthday.Date - Name)

nothing is mapped to birthday except names

all n: Name | **lone** n.birthday

every name maps to at most one birthday

Let Expressions and Constraints

let x = e | A
f **implies** e1 **else** e2

A can be a constraint or an expression.
if f **then** e1 **else** e2

Example

Four equivalent constraints:

all n: Name | (**some** n.lunarBirthday
implies n.birthday = n.lunarBirthday **else** n.birthday = n.solarBirthday)

all n: Name | **let** l = n.lunarBirthday, d = n.birthday |
(**some** l **implies** d = l **else** d = n.solarBirthday)

all n: Name | **let** l = n.lunarBirthday |
n.birthday = (**some** l **implies** l **else** n.solarBirthday)

all n: Name | n.birthday =
(**let** l = n.lunarBirthday | (**some** l **implies** l **else** n.solarBirthday))

Comprehensions

{x1: e1, x2: e2, ..., xn: en | F}

Example

{n: Name | **no** n.^birthday & Date}

set of names that don't resolve to any actual birthdays

{n: Name, D: Date | n -> D **in** ^birthday}

binary relation mapping names to reachable birthday

Declarations

relation-name : expression

- almost the same as the meaning of a subset constraint x **in** e

Example

birthday: Name->Date

a signal birthday book, maps names to birthdays

birth: Book->Name->Date

a collection of birthday books, maps books to names to birthday

birthday: Name->(Name + Date)

a multilevel birthday book maps names to names and birthday

Set Multiplicities

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

x: *m* e
x: e <=> x: **one** e

Example

RecentlyUsed: **set** Name

RecentlyUsed is a subset of the set Name

myBirthday: Date

myBirthday is a singleton subset of Date

myName: **lone** Name

myName is either empty or a singleton subset of Name

theirBirthday: **some** Date

theirBirthday is a nonempty subset of Date

Relation Multiplicities

r: A *m* -> *n* B

- r: A *m* -> *n* B <=> ((**all** a: A | *n* a.r) **and** (**all** b: B | *m* r.b))
- r: A -> B <=> r: A **set** -> **set** B
- r: A -> (B *m* -> *n* C) <=> **all** a: A | a.r: B *m* -> *n* C
- r: (A *m* -> *n* B) -> C <=> **all** c: C | r.c: A *m* -> *n* B

Example

birthday: Name -> **lone** Date

each name refers to at most one birthday

members: Group **lone** -> **some** Addr

address belongs to at most one group name and group contains at least one address

Outline

"I'm My Own Grandpa" in Alloy

```
module grandpa /*module header*/
abstract sig Person { /*signature declarations*/
   father: lone Man,
   mother: lone Woman
}
sig Man extends Person {
   wife: lone Woman
}
sig Woman extends Person {
   husband: lone Man
}

fact { /*constriant paragraphs*/
   no p: Person | p in p.^(mother + father)
   wife = ~husband
}
```

"I'm My Own Grandpa" in Alloy (Cont'd)

```
assert noSelfFather { /*assertions*/
  no m: Man | m = m.father
}
check noSelfFather /*commands*/

fun grandpas[p: Person] : set Person { /*constriant paragraphs*/
  p.(mother + father).father
}
pred ownGrandpa[p: Person] { /*constriant paragraphs*/
  p in grandpas[p]
}
run ownGrandpa for 4 Person /*commands*/
```

Signatures

sig A {}
set of atoms A

sig A {}
sig B {}
disjoint sets A and B (no A & B)

sig A, B {}
same as above

sig B **extends** A {}
set B is a subset of A (B in A)

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 (Cont'd)

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

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

Field Declarations

sig A {f: e}
f is a binary relation with domain A and range given by expression e
f is constrained to be a function: (f: A -> **one** e) or (**all** a: A | a.f: e)

sig A { f1: **one** e1, f2: **lone** e2, f3: **some** e3, f4: **set** e4 }
(all a: A | a.fn : m e)

sig A {f, g: e}
two fields with same constraints

sig A {f: e1 m -> n e2}
(f: A -> (e1 m -> n e2)) or (**all** a: A | a.f : e1 m -> n e2)

```
sig Book {
   names: set Name,
   birthday: names -> Date
}
```

dependent fields (**all** b: Book | b.birthday: b.names -> Date)

grandpa: field

```
abstract sig Person {
    father: lone Man,
    mother: lone Woman
}
sig Man extends Person {
    wife: lone Woman
}
sig Woman extends Person {
    husband: lone Man
}
```

- fathers are men and everyone has at most one
- mothers are women and everyone has at most one
- wives are women and every man has at most one
- husbands are men and every woman has at most one

Facts

fact { F }
fact f { F }
sig S { ... }{ F }

- facts introduce constraints that are assumed to always hold

Example

```
fact {
  no p: Person |
    p in p.^(mother + father)
  wife = ~husband
}
```

Functions

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

- functions are named expression with declaration parameters and a declaration expression as a result invoked by providing an expression for each parameter

Example

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

Predicates

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

- named formula with declaration parameters

Example

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

"Receiver" Syntax

fun f[x: X, y: Y, ...] : Z {...x...}
fun X.f[y:Y, ...] : Z {...this...}

pred p[x: X, y: Y, ...] {...x...}
pred X.p[y:Y, ...] {...this...}

- Whether or not the predicate or function is declared in this way, it can be used in the form

  x.p[y, ...]

  where *x* is taken as the first argument, *y* as the second, and so on.

Example

```
fun Person.grandpas : set Person {
  this.(mother + father).father
}

pred Person.ownGrandpa {
  this in this.grandpas
}
```

Assertions and Check Command

assert a { F }

- constraint intended to follow from facts of the model

check a *scope*

- instructs analyzer to search for counterexample to assertion within scope
- if model has facts M, finds solution to $M \&\& !F$

Example

```
fact {
  no p: Person | p in p.^(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 predicate within scope
- if model has facts M, finds solution to
  $M \ \&\& \ (some\ x : X,\ y : Y,\ ... \mid F)$

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

- instructs analyzer to search for instance of function within scope
- if model has facts M, finds solution to
  $M \ \&\& \ (some\ x : X,\ y : Y,\ ...,\ result : R \mid result = E)$

grandpa: predicate simulation

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

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

run ownGrandpa for 4 Person
```

- command instructs analyzer to search for configuration with at most 4 people in which a man is his own grandfather

Types and Type Checking

- Alloy's type system has two functions.
  - It allows the analyzer to catch errors before any serious analysis is performed.
  - It is used to resolve overloading.
- A *basic type* is introduced for each top-level signature and for each extension signature.
  - A signature that is declared independently of any other is a *top-level* signature.
- When signature *A1* extends signature *A*, the type associated with *A1* is a *subtype* of the type associated with *A*.
- A subset signature acquired its parent's type.
  - If declared as a subset of a union of signatures, its type is the union of the types of its parents.
- Two basic type are said to *overlap* if one is a subtype of the other.

Types and Type Checking (Cont'd)

- Every expression has a *relational type*, consisting of a union of products:

  $$A_1\text{->}B_1\text{->}\ldots \;+\; A_2\text{->}B_2\text{->}\ldots \;+\; \ldots$$

  where each of the A_i, B_i, and so on, is a basic type.
  - A binary relation's type, for example, will look like this:

    $$A_1\text{->}B_1 \;+\; A_2\text{->}B_2 \;+\; \ldots$$

    and a set's type like this:

    $$A_1 \;+\; A_2 \;+\; \ldots$$

  - The type of an expression is itself just an Alloy expression.
- Types are inferred automatically so that the value of the type always contains the value of the expressions. It's an *overapproximation*.
  - If two types have an empty intersection, the expressions they were obtained from must also have an empty intersection.

Types and Type Checking (Cont'd)

- There are two kinds of type error.
  - It is illegal to form expressions that would give relations of mixed arity.
  - An expression is illegal if it can be shown, from the declarations alone, to be redundant, or to contain a redundant subexpression.
- The subtype hierarchy is used primarily to determine whether types are disjoint.
- The typing of an expression of the form $s.f$ where s is a set and f is a relation only requires s and the domain of r to overlap.
  - The case that two types are disjoint is rejected, because it always results in the empty set.
- Type checking is sound.
  - When checking an intersection expression, for example, if the resulting type is empty, the relation represented by the expression must be empty.

Types and Type Checking (Cont'd)

- A signature defines a local namespace for its declarations, so you can use the same field name in different signatures, and each occurence will refer to a different field.
- When a field name appears that could refer to multiple fields, the types of the candidate fields are used to determine which field is meant.
- If more than one field is possible, an error is reported.

Example

sig Object, Block {}
sig Directory **extends** Object {contents: **set** Object}
sig File **extends** Object {contents: **set** Block}
all f: File | **some** f.contents
// The occurrence of the field name *contents* in the constraint is trivially resolved.
all o: Object | **some** o.contents
// The occurrence of the field name *contents* in the constraint is not resolved, and the constraint is rejected.

Outline

The Alloy Analyzer

- The Alloy Analyzer is a 'model finder'.
- Given a logical formula (in Alloy), it attempts to find a model that makes the formula true.
  - A model is a binding of the variables to values.
- For simulation, the formula will be some part of the system description.
  - If it's a state invariant INV, models of INV will be states that satisfy the invariant.
  - If it's an operation OP, with variables representing the before and after states, models of OP will be legal state transitions.
- For checking, the formula is a negation, usually of an implication.
  - To check that the system described by the property SYS has a property PROP, you would assert (SYS implies PROP).
  - The Alloy Analyzer negates the assertion, and looks for a model of (SYS and not PROP), which, if found, will be a counterexample to the claim.

The Small Scope Hypothesis

- Simulation is for determining consistency (i.e., satisfiability) and Checking is for determining validity And these problems are undecidable for Alloy specifications.
- Alloy analyzer restricts the simulation and checking operations to a finite scope.
- Validity and consistency problem within a finite scope are decidable problems.
- *Most bugs have small counterexample.*
- If an assertion is invalid, it probably has a small counterexample.

How Does It Work

- The Alloy Analyzer is essentially a compiler.
- It translates the problem to be analyzed into a (usually huge) boolean formula.
- Think about a particular value of a binary relation r from a set A to a set B:
  - The value can be represented as an adjacency matrix of 0's and 1's, with a 1 in row i and column j when the *ith* element of A is mapped to the *jth* element of B.
  - So the space of all possible values of r can be represented by a matrix of boolean *variables*.
  - The dimensions of these matrices are determined by the scope; if the scope bounds A by 3 and B by 4, r will be a 3×4 matrix containing 12 boolean variables, and having 2^{12} possible values.

How Does It Work (Cont'd)

- Now, for each relational expression, a matrix is created whose elements are boolean expressions.
  - For example, the expression corresponding to $p + q$ for binary relations p and q would have the expression $p_{i,j} \vee q_{i,j}$ in row i and column j.
- For each relational formula, a boolean formula is created.
  - For example, the formula corresponding to *pinq* would be the conjunction of $p_{i,j} \Rightarrow q_{i,j}$ over all values of i and j.
- The resulting formula is handed to a SAT solver, and the solution is translated back by the Alloy Analyzer into the language of the model.
- All problems are solved within a user-specified scope that bounds the size of the domains, and thus makes the problem finite (and reducable to a boolean formula).
- Alloy analyzer implements an efficient translation in the sense that the problem instance presented to the SAT solver is as small as possible.

Different from Model Checkers

- The Alloy Analyzer is designed for analyzing state machines with operations over complex states.
- Model checkers are designed for analyzing state machines that are composed of several state machines running in parallel, each with relatively simple state.
- Alloy allows structural constraints on the state to be described very directly (with sets and relations), whereas most model checking languages provide only relatively low-level data types (such as arrays and records).
- Model checkers do a temporal analysis that compares a state machine to another machine or a temporal logic formula.