Alloy

(Based on [Jackson 2006])

Yih-Kuen Tsay
(with help from Yi-Wen Chang)

Dept. of Information Management
National Taiwan University

Outline

About Alloy

Logic

Language

Analysis

The Alloy Philosophy

- The core of software development is the design of abstractions.
- An abstraction is an idea reduced to its essential form.
- You carefully design the abstractions and then develop their embodiments in code.
- To find flaws early, the abstractions should be made precise and unambiguous using formal specification.
- To be practically useful, the formal notation should be based on a small core of simple and robust concepts.
- It is even more important to adopt a fully automatic analysis that provides immediate feedbacks.
- The insist on full automation, according to the originator, was inspired by the success of model checking.

What Is Alloy?

- The Alloy approach consists of a modeling language and an automatic analyzer.
- The language, Alloy, is a structural modelling language based on first-order logic, for expressing complex structural constraints and behaviors.
- The Alloy Analyzer is a constraint solver that provides fully automatic simulation and checking.
- The approach is developed by the Software Design Group of Daniel Jackson at MIT.
- Jackson boasts the approach to be "lightweight formal methods".

Contrast with OCL

- Like OCL, Alloy has a pure ASCII notation and does not require special typesetting tools.
- As a modeling language, Alloy is similar to OCL, but it has a more conventional syntax and a simpler semantics.
- Unlike OCL, Alloy is designed for fully automatic analysis.

Alloy = Logic + Language + Analysis

- Logic
  - the core that provides the fundamental concepts
  - first-order logic + relational calculus
- Language
  - syntax for structuring specifications in the logic
- Analysis
  - bounded search by constraint solving
  - simulation: finding instances of states or executions that satisfy a given property
  - checking: finding a counterexample to a given property

Example

- An address book for an email client
  - associates email addresses with shorter names that are more convenient to use.
  - alias: a nickname that can be used in place of the person's address
  - group: an entire set of correspondents
- Sample models under "book/chapter2" in the Alloy Analyzer

Outline

Three Logics in One

- Predicate calculus style
  Two kinds of expression: relation names, which are used as predicates, and tuples formed from quantified variables.

  all n: Name, d, d': Address |
  n -> d **in** address **and** n -> d' **in** address **implies** d = d'

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

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

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

  no ~address.address - **iden**

Atoms and Relations

- Atoms are Alloy's primitive entities.
  - They are indivisible, immutable, and uninterpreted.
- A relation is a structure that relates atoms.
  - It consists of a set of tuples, each tuple being a sequence of one or more atoms.
  - All relations are first-order, i.e., relations cannot contain relations.
- Every value in the Alloy logic is a relation.
  - Relations, sets, and scalars all are the same thing.
  - A scalar is represented by a singleton set.

Everything Is a Relation

- *Sets* are unary relations

$$\begin{array}{lcl} \text{Name} & = & \{(\text{N0}), (\text{N1}), (\text{N2})\} \\ \text{Addr} & = & \{(\text{A0}), (\text{A1}), (\text{A2})\} \\ \text{Book} & = & \{(\text{B0}), (\text{B1})\} \end{array}$$

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

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

- Binary relation

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

- Ternary relation

$$\begin{array}{lcl} \text{addrs} & = & \{(\text{B0}, \text{N0}, \text{A0}), (\text{B0}, \text{N1}, \text{A1}), \\ & & (\text{B1}, \text{N1}, \text{A2}), (\text{B1}, \text{N2}, \text{A2})\} \end{array}$$

Constants

none	empty set
univ	universal set
iden	identity

Example

Name = {(N0), (N1), (N2)}
Addr = {(A0), (A1)}

none = {}
univ = {(N0), (N1), (N2), (A0), (A1)}
iden = {(N0, N0), (N1, N1), (N2, N2), (A0, A0), (A1, A1)}

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

-> arrow (product)

Example

Name = {(N0), (N1)}
Addr = {(A0), (A1)}
Book = {(B0)}

Name->Addr = {(N0, A0), (N0, A1), (N1, A0), (N1, A1)}
Book->Name->Addr =
{(B0, N0, A0), (B0, N0, A1), (B0, N1, A0), (B0, N1, A1)}

Relational Join

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

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

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)}
Addr = {(A0), (A1), (A2)}
Host = {(H0), (H1)}
myName = {(N1)}
myAddr = {(A0)}
address = {(B0, N0, A0), (B0, N1, A0), (B0, N2, A2)}
host = {(A0, H0), (A1, H1), (A2, H1)}

Book.address = {(N0, A0), (N1, A0), (N2, A2)}
Book.address[myName] = {(A0)}
Book.address.myName = {}
host[myAddr] = {(H0)}
address.host = {(B0, N0, H0), (B0, N1, H0), (B0, N2, H1)}

Unary Operators

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

$$\hat{}r = r + r.r + r.r.r + \ldots$$
$$*r = \textbf{iden} + \hat{}r$$

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 ++ q =
p − (domain[q] <: p) + q

Example

Name = {(N0), (N1), (N2)}
Alias = {(N0), (N1)}   Addr = {(A0)}
address = {(N0, N1), (N1, N2), (N2, A0)}

address :> Addr = {(N2, A0)}
Alias <: address = {(N0, N1), (N1, N2)}
address :> Name = {(N0, N1), (N1, N2)}
address :> Alias = {(N0, N1)}

workAddress = {(N0, N1), (N1, A0)}
address ++ workAddress = {(N0, N1), (N1, A0), (N2, A0)}

m' = m ++ (k->v)   *update map m with key-value pair (k, v)*

Boolean Operators

IM NTU

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)

Quantification

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

Example

some n: Name, a: Address | a **in** n.address
some name maps to some address - address book not empty

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

all n: Name | **lone** a: Address | a **in** n.address
every name maps to at most one address - address book is functional

all n: Name | **no disj** a, a': Address | (a + a') **in** n.address
no name maps to two or more distinct addresses - 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 address
address book is not empty - it has a tuple

no (address.Addr − Name)
nothing is mapped to addresses except names

all n: Name | **lone** n.address
every name maps to at most one address

Let Expressions and Constraints

IM NTU

let x = e | A — A can be a constraint or an expression.

f **implies** e1 **else** e2 — **if** f **then** e1 **else** e2

Example

Four equivalent constraints:

all n: Name | (**some** n.workAddress
implies n.address = n.workAddress **else** n.address = n.homeAddress)

all n: Name | **let** w = n.workAddress, a = n.address |
(**some** w **implies** a = w **else** a = n.homeAddress)

all n: Name | **let** w = n.workAddress |
n.address = (**some** w **implies** w **else** n.homeAddress)

all n: Name | n.address =
(**let** w = n.workAddress | (**some** w **implies** w **else** n.homeAddress))

Comprehensions

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

Example

{n: Name | **no** n.^address & Addr}
set of names that don't resolve to any actual addresses

{n: Name, a: Address | n -> a **in** ^address}
binary relation mapping names to reachable addresses

Declarations

IM NTU

relation-name : expression

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

Example

address: Name−>Addr

a single address book mapping names to addresses

addr: Book−>Name−>Addr

a collection of address books mapping books to names to addresses

address: Name−>(Name + Addr)

a multilevel address book mapping names to names and addresses

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

senderAddress: Addr
senderAddress is a singleton subset of Addr

senderName: **lone** Name
senderName is either empty or a singleton subset of Name

receiverAddresses: **some** Addr
receiverAddresses is a nonempty subset of Addr

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

workAddress: Name -> **lone** Addr
each name refers to at most one work address

members: Group **lone** -> **some** Addr
address belongs to at most one group name and group contains at least one address

Cardinality Constraints

`#r`	number of tuples in r
`0,1,...`	integer literal
`+`	plus
`-`	minus

=	equals
<	less than
>	greater than
=<	less than or equal to
>=	greater than or equal to

sum x: e | ie

sum of integer expression *ie* for all singletons *x* drawn from *e*

Example

all b: Bag | #b.marbles =< 3

all bags have 3 or less marbles

#Marble = **sum** b: Bag | #b.marbles

the sum of the marbles across all bags equals the total number of marbles

Outline

"I'm My Own Grandpa" in Alloy

```
module language/grandpa1 /* 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 { /* constraint 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 { /* constraint paragraphs */
   p.(mother + father).father
}
pred ownGrandpa[p: Person] { /* constraint paragraphs */
   p in grandpas[p]
}
run ownGrandpa for 4 Person /* commands */
```

“I’m My Own Grandpa” in Alloy (Cont’d)

```
module language/grandpa3
...
fact {
  no p: Person | p in p.^(mother + father) /* biology */
  wife = ~husband /* terminology */
  no (wife + husband) & ^(mother + father) /* social convention */
}
...
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
```

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

Signatures (Cont'd)

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

abstract sig A {}
sig B **extends** A {}
sig C **extends** A {}
A is partitioned by disjoint subsets B and C (no B & C && A = (B + C))

Signatures (Cont'd)

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: **one** 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,
   addrs: names -> Addr
}
```
dependent fields (**all** b: Book | b.addrs: b.names -> Addr)

Fields in the "Self-Grandpas" Example


```
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

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.'

Facts in "Self-Grandpas"

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

- No person is his or her own ancestor.
- A man's wife has that man as a husband.
- A woman's husband has that woman as a wife.

Functions

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

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

Example

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

fun lookup[b: Book, n: Name] : set Addr {
  b.addr[n]
}

fact everyNameMapped {
  all b: Book, n: Name | some lookup[b, n]
}
```

Predicates

IM NTU

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

- Predicates are named formulae with declaration parameters.

Example

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

pred contains[b: Book, n: Name, d: Addr] {
  n->d in b.addr
}

fact everyNameMapped {
  all b: Book, n: Name |
    some d: Addr | contains[b, n, a]
}
```

Functions and Predicates in "Self-Grandpas"

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

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

- A person's grandpas are the fathers of one's own mother and father.

"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

assert a { F }

- An assertion is a constraint intended to follow from facts of the model.

Example

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

fact { Node in Root.*children }

assert someParent { // invalid assertion
  all n: Node | some children.n
}

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

Check Commands

assert a { F }

check a *scope*

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

Example

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, invalid if any unbound

Check Commands (Cont'd)

Example

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 Man, 5 Woman
check a **for** 4 Person
check a **for** 3 Man, 4 Woman
check a **for** 3 Man, 4 Woman, 2 Grandpa

// invalid, because top-level bounds unclear
check a **for** 3 Man
check a **for** 5 Woman, 2 Grandpa

Assertion Checks in "Self-Grandpas"

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

assert noSelfFather {
    no m: Man | m = m.father
}

check noSelfFather
```

- The check command instructs the analyzer to search for a counterexample to noSelfFather within a scope of at most 3 Persons.

Run Commands

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

- instructs the analyzer to search for an instance of the predicate within scope
- if the model has facts M, finds a solution to
  M && $(some\ x : X,\ y : Y,\ ...\ |\ F)$

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

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

Predicate Simulation in "Self-Grandpas"

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

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

run ownGrandpa for 4 Person
```

- The run command instructs the analyzer to search for a 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 types 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 values 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 $s.r$ where s is a set and r 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.
- When a field name refers to possibly 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* is trivially resolved.

all o: Object | **some** o.contents
// The occurrence of *contents* here is not resolved, and the constraint is rejected.

Outline

About Alloy

Logic

Language

Analysis

The Alloy Analyzer

- The Alloy Analyzer is a 'model finder'.
- Given a logical formula, 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 is a state invariant INV, models of INV will be states that satisfy the invariant.
  - If it is 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.
- The Alloy Analyzer restricts the simulation and checking operations to a finite scope.
- The validity and consistency problems within a finite scope are decidable problems.
- *Most bugs have a 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 p **in** q 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.

Differences 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 states.
- 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.