272: Software Engineering

Instructor: Tevfik Bultan

Lectures 2 and 3: Alloy and Alloy Analyzer
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

• 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
  – My slides are based on an old Alloy tutorial, documentation about the current version of Alloy is available here: https://alloytools.org/
A Book Store Data Model in UML

- Book Category
  - 0..*
  - 0..*
  - 1

- Book
  - 1
  - 1..*
  - 0..*

- Book Edition
  - 1

- User
  - 1
  - 0..1

- Shopping Cart
  - 1

- Order Line
  - 1
  - 0..*
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
}
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
Textual Representation

• Alloy is a textual language
  – There used to be a graphical representation to support it initially

• The textual representation represents the Alloy model completely
  – i.e., a graphical representation is not needed
Basics of Alloy Semantics

• Each sig 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
Signatures

- In Alloy sets of atoms such as FSObject, File, Dir, FileSystem 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
  - FileSystem and FSObject are top-level signatures
    - They represent disjoint sets of objects

- Extensions of a signature are also disjoint
  - File and Dir are disjoint sets

- An abstract signature has no elements except those belonging to its extensions
  - There is no FSObject that is not a File or a Dir
Subclasses as subsets

• The keyword `extends` indicates disjoint subsets
  – This is the default, if a subset is not labeled, then it is assumed to extend
  – `File` and `Dir` are disjoint sets (their intersection is empty)
    • There is no `FSObject` that is both a `File` and a `Dir`

• The keyword `in` indicates subsets, not necessarily disjoint from each other (or other subsets that extend)
Class associations are relations

- For example, *live is a relation between FileSystem to FSObject*

- Relations are expressed as fields of signatures
  - These correspond to associations in UML-OCL
  - They express relations between object classes
Signatures

• Textual representation starts with \texttt{sig} declarations defining the signatures (sets of atoms)
  – You can think of signatures as object classes, each signature represents a set of objects

• Multiplicity:
  – \texttt{set} zero or more
  – \texttt{one} exactly one
  – \texttt{lone} zero or one
  – \texttt{some} one or more

• \texttt{extends} and \texttt{in} are used to denote which signature is subset of which other signature
  – \texttt{extends} denotes disjoint subsets
Signatures

sig A {}

set of atoms A

sig A {}
sig B {}

disjoint sets A and B. As an Alloy expression we can write: \( \text{no } A \land B \) (Alloy expressions are discussed in later slides)

sig A, B {}
same as above

sig B extends A {}

set B is a subset of A. As an Alloy expression: \( B \text{ in } A \)

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

B and C are disjoint subsets of A: \( B \text{ in } A \land C \text{ in } A \land \text{no } B \land C \)

sig B, C extends A {}
same as above
Signatures

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: 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
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
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)
• 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

```plaintext
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:

```plaintext
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

\[ \textbf{fun} \ f[\texttt{\texttt{x1}: e1, \ldots, xn: en}] \ : \ e \ \{ \ E \} \]

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

\[
\text{pred } p[x_1: e_1, \ldots, x_n: e_n] \{ F \}
\]

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

```plaintext
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 $M \land \neg F$ where $M$ is the formula representing the model

```plaintext
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
```
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 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

```plaintext
sig Name { } 
abstract sig Person {
  name: one Name,
}

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**

\[ x: \text{m e} \quad \text{x is a subset of e and its cardinality (size) is restricted to be m} \]

\( \text{m can be:} \)

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

\[ x: \text{e} \quad \text{is equivalent to} \quad x: \text{one e} \]

\( \text{SomePeople: set Person} \)

SomePeople is a subset of the set Person
Set Operators

+ union

\& intersection

- difference

in subset

= equality
Product Operator

\[\rightarrow \quad \text{cross product}\]

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

Person \rightarrow Name =
\{(P0, N0), (P0, N1), (P1, N0), (P1, N1)\}

Person \rightarrow Name \rightarrow Address =
\{(P0, N0, A0), (P0, N1, A0), (P1, N0, A0), (P1, N1, A0)\}
Relation Declarations with Multiplicity

\( r: A \ m \rightarrow n \ B \)  \quad \text{cross product with multiplicity constraints}
\( m \text{ and } n \text{ can be one, lone, some, set} \)

\( r: A \rightarrow B \)  \text{ is equivalent to (default multiplicity is set)}
\( r: A \text{ set} \rightarrow \text{ set B} \)

\( r: A \ m \rightarrow n \ B \)  \text{ is equivalent to:}
\( r: A \rightarrow B \)
\( \text{all } a: A \mid n \ a.r \)
\( \text{all } b: B \mid m \ r.b \)
Relation Declarations with Multiplicity

\[ r: A \rightarrow \text{one } B \]
- \( r \) is a function with domain \( A \)

\[ r: A \text{ one } \rightarrow B \]
- \( r \) is an injective relation with range \( B \)

\[ r: A \rightarrow \text{lone } B \]
- \( r \) is a function that is partial over the domain \( A \)

\[ r: A \text{ one } \rightarrow \text{one } B \]
- \( r \) is an injective function with domain \( A \) and range \( B \) (a bijection from \( A \) to \( B \))

\[ r: A \text{ some } \rightarrow \text{some } B \]
- \( r \) is a relation with domain \( A \) and range \( B \)
Relational Join (aka navigation)

\[ p \cdot q \]

dot is the relational join operator

Given two tuples \((p_1, \ldots, p_n)\) in \(p\) and \((q_1, \ldots, q_m)\) in \(q\) where \(p_n = q_1\)
\(p.q\) contains the tuple \((p_1, \ldots, p_{n-1}, q_2, \ldots, q_m)\)

\[
\{(N0)\} \cdot \{(N0, D0)\} = \{(D0)\}
\]
\[
\{(N0)\} \cdot \{(N1, D0)\} = \{}
\]
\[
\{(N0)\} \cdot \{(N0, D0), (N0, D1)\} = \{(D0), (D1)\}
\]
\[
\{(N0), (N1)\} \cdot \{(N0, D0), (N1, D1), (N2, D3)\} = \{(D0), (D1)\}
\]
\[
\{(N0, A0)\} \cdot \{(A0, D0)\} = \{(N0, D0)\}
\]
box join, box join can be defined using dot join

\[
e_1[e_2] = e_2.e_1
\]

\[
a.b.c[d] = d.(a.b.c)
\]
Unary operations on relations

\[ \sim \quad \text{transpose} \]

\[ ^\land \quad \text{transitive closure} \]

\[ ^\ast \quad \text{reflexive transitive closure} \]

these apply only to binary relations

\[ ^\land r = r + r \cdot r + r \cdot r \cdot r + \ldots \]

\[ ^\ast r = \text{idem} + ^\land r \]

parent = \{(N1,N3), (N2, N3)\}

\[ \sim \text{parent} = \text{child} = \{(N3,N1), (N3, N2)\} \]
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)}
rangle(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

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

\[ m' = m ++ (k > v) \]

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)}
// 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
// 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
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
}
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 {
}

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 ::=  
  elemFormula  
  | compFormula 
  | quantFormula 

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

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

unop ::=  
  ~  
  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 $\rightarrow$ Boolean
  – $E$: which interprets an expression as a relation value
    • $E$: Expression, Instance $\rightarrow$ 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 \{x_1, x_2, \ldots\} is represented as a relation \{(\text{unit},x_1), (\text{unit},x_2), \ldots\}\.
- Scalar types are singleton sets, i.e., a scalar \(x_1\) is represented as \{x\} which is actually represented as the relation \{(\text{unit},x_1)\}. 
Alloy Kernel Semantics

M: Formula, Instance → Boolean

Formula Semantics:

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

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

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

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

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

\(i \oplus x \rightarrow v\) 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[none]i = ∅

E[p+q]i = E[p]i ∪ E[q]i

E[p&q]i = E[p]i ∩ E[q]i

E[p–q]i = E[p]i \ E[q]i

E[p.q]i = {(p_1, ..., p_{n-1}, q_2, ..., q_m) | (p_1, ..., p_n) ∈ E[p]i ∧ (q_1, ..., q_m) ∈ E[q]i ∧ p_n = q_1}

E[~p]i = {(y,x) | (x,y) ∈ E[p]i}

E[^p]i = {(x,y) | ∃p_1, ..., ∃p_n, n≥0 | (x,p_1), (p_1,p_2), ..., (p_n,y) ∈ 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
  \[ \text{all } x:X \mid \text{some } y:Y \mid x.r = y \]

• To check this formula we formulate its negation as a problem
  \[ r: X \rightarrow Y \]
  \[ \neg \text{all } x:X \mid \text{some } y:Y \mid x.r = y \]

• 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 validity 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

\[ \forall x : X \mid \exists y : Y \mid x.r = y \]

is converted to

\[ \exists x : X \mid \forall y : Y \mid \neg x.r = y \]

which is converted to the problem

\[ r : X \rightarrow Y \\
x : X \\
\forall y : Y \mid \neg x.r = y \\
\exists z : X \mid z = x \]
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
  
  \[ \text{all } y: Y \mid !x.r=y \]

  using a scope of 2 would be translated as follows

- First let’s look at the negation of the formula
  
  \[ \text{some } y: Y \mid x.r=y \]

- Generate a vector \([x_0 \ x_1]\) for \(x\) and a matrix \([r_{00} \ r_{01}, \ r_{10} \ r_{11}]\) for \(r\)

- The expression \(x.r\) corresponds to the vector
  
  \[ [x_0 \land r_{00} \lor x_1 \land r_{10} \ \ x_0 \land r_{01} \lor x_1 \land r_{11}] \]
Translation Overview

• Given,
  \[ x.r \equiv [x_0 \land r_{00} \lor x_1 \land r_{10} \land x_0 \land r_{01} \lor x_1 \land r_{11}] \]
  and \( y \equiv [y_0 \ y_1] \), we get
  \[ x.r = y \equiv \]
  \[ (y_0 \leftrightarrow (x_0 \land r_{00} \lor x_1 \land r_{10})) \land (y_1 \leftrightarrow (x_0 \land r_{01} \lor x_1 \land r_{11})) \land (y_0 \land \neg y_1 \lor \neg y_0 \land y_1) \]

• Then the boolean logic translation for some \( y \): \( Y \mid x.r=y \) is
  \[ true \leftrightarrow (x_0 \land r_{00} \lor x_1 \land r_{10}) \land false \leftrightarrow (x_0 \land r_{01} \lor x_1 \land r_{11}) \land false \leftrightarrow (x_0 \land r_{00} \lor x_1 \land r_{10}) \land true \leftrightarrow (x_0 \land r_{01} \lor x_1 \land r_{11}) \equiv (x_0 \land r_{00} \lor x_1 \land r_{10}) \land \neg (x_0 \land r_{01} \lor x_1 \land r_{11}) \land \neg (x_0 \land r_{00} \lor x_1 \land r_{10}) \land (x_0 \land r_{01} \lor x_1 \land r_{11}) \]
Translation Overview

- Hence, the formula \( \text{some } y: Y | x.r=y \) is satisfiable within a scope of 2 if and only if the following boolean logic formula is satisfiable:
  \[
  (x_0 \land r_{00} \lor x_1 \land r_{10}) \land \neg (x_0 \land r_{01} \lor x_1 \land r_{11}) \\
  \lor \neg (x_0 \land r_{00} \lor x_1 \land r_{10}) \land (x_0 \land r_{01} \lor x_1 \land r_{11})
  \]

This is equivalent to checking validity of the formula:
\[
\text{all } y: Y | \neg x.r=y
\]
equivalently we can write
\[
\equiv \neg (\text{some } y: Y | x.r=y)
\]
and then check satisfiability of its negation:
\[
(\text{some } y: Y | x.r=y)
\]
within the scope of 2 that is equivalent to the boolean logic formula above:
\[
\neg((x_0 \land r_{00} \lor x_1 \land r_{10}) \land \neg (x_0 \land r_{01} \lor x_1 \land r_{11}) \\
\lor \neg (x_0 \land r_{00} \lor x_1 \land r_{10}) \land (x_0 \land r_{01} \lor x_1 \land r_{11}))
\]
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