Static Program Checking
Alloy Engine

Jun.-prof. Mana Taghdiri
Thursday – May 15, 2014
Alloy Analysis

- Terminology notes
  - Alloy solves a model and produces an instance (solution)
    - Alloy model = collection of constraints in Alloy
    - Alloy instance = assignment of symbolic values to Alloy variables
  - In literature, these terms are used differently
    - Problem = collection of constraints
    - Model = assignment of symbolic values to variables used in the constraints
Alloy Analysis

- Alloy problem

\[ \wedge \text{Decls} \wedge \text{Facts} \rightarrow \text{assertion} \]

- Automatic analysis is the biggest advantage of Alloy
  - An instance finder (model finder)
  - As a simulator:
    \[ \text{findInst}(\wedge \text{Decls} \wedge \text{Facts}) \]
  - As a checker (same mechanism, why?)
    \[ \text{findInst}(\neg(\wedge \text{Decls} \wedge \text{Facts} \rightarrow \text{assertion})) \]
  - Constantly used while a model is being developed
Scope-complete analysis

- Alloy logic is undecidable
  - Why?
  - What are some of its decidable subsets?
Scope-complete analysis

- Alloy logic is undecidable
  - Why?
  - What are some of its decidable subsets?
    - Monadic first-order (no relations of arity higher than 1)
    - A particular ordering of quantifiers (prefixes of the form [all]2[some]*)

- Relational calculus (with the join operator) is undecidable

\[(a, b) \in R \cdot S \iff \exists x; (a, x) \in R \land (x, b) \in S\]

\[Exp_1 \subseteq Exp_2 \iff \forall \vec{u}; \vec{v} \in Exp_1 \rightarrow \vec{v} \in Exp_2\]

- Can produce an undecidable combination of quantifiers

- Analysis is performed w.r.t. a scope
  - A multi-dimensional space of test cases
  - Is separate from the model itself

- Small scope hypothesis
  - Most bugs have small counterexamples
Alloy analysis performed via boolean SAT

- Alloy Analyzer is like a compiler
  - Translates F.O. relational formulas to boolean formulas
  - Uses an off-the-shelf SAT solver to solve the boolean formula

- Clarifications
  - Example of an FO relational formula?
  - Example of a boolean formula?
  - What is a SAT problem?
Alloy analysis steps

- Initial conversions
- Translation to a boolean formula
- Conversion to conjunctive normal form
- Solving using an off-the-shelf SAT solver
- Reconstructing an Alloy solution
Alloy analysis steps

- Initial conversions
- Translation to a boolean formula
- Conversion to conjunctive normal form
- Solving using an off-the-shelf SAT solver
- Reconstructing an Alloy solution
Initial conversions

- Negation normal form (NNF)
  - Only elementary formulas are negated
  - Push the negation inwards as much as possible using de Morgan’s law
- Example
  not (all x: X | some y: Y | x.r = y) ?
  some x: X | all y: Y | not x.r = y
Handling quantifiers

- Universal
  - Ground out the quantifier
  - Example:
    
    all x: S | F where S = {S0, S1, S2} ?
Handling quantifiers

- **Universal**
  - Ground out the quantifier
  - Example:
    ```
    all x: S | F where S = \{S0, S1, S2\} ?
    
    F[S0/x] and F[S1/x] and F[S2/x]
    ```

- **Existential**
  - Similar approach
    ```
    some x: S | F where S = \{S0, S1, S2\} ?
    ```
Handling quantifiers

- Universal
  - Ground out the quantifier
  - Example:
    \[
    \text{all } x : S \mid F \quad \text{where } S = \{S_0, S_1, S_2\} \quad ?
    \]
    
    \[F[S_0/x] \text{ and } F[S_1/x] \text{ and } F[S_2/x]\]

- Existential
  - Similar approach
    \[
    \text{some } x : S \mid F \quad \text{where } S = \{S_0, S_1, S_2\} \quad ?
    \]
    
    \[F[S_0/x] \text{ or } F[S_1/x] \text{ or } F[S_2/x]\]

  - Skolemization: Replaces the bound variable by a fresh free variable
    \[(sx : S) \text{ and } F[sx/x]\]
    
    - Why correct?
Skolemization

Why better?
- Better performance
  - This is exactly what the SAT solver is for
- Witness generation
  - A value for the quantified var that makes the body of the formula true
  - In the disjunction case, since x doesn’t appear, it is not clear which disjunct is true when an instance is found
  - Free vars are named as predName_varName in Alloy

What about nested quantifiers?
\[
\text{all } x: S \mid \text{some } y : T \mid F
\]
Skolemization

- Why better?
  - Better performance
    - This is exactly what the SAT solver is for
  - Witness generation
    - A value for the quantified var that makes the body of the formula true
    - In the disjunction case, since x doesn’t appear, it is not clear which disjunct is true when an instance is found
    - Free vars are named as predName_varName in Alloy

- What about nested quantifiers?
  all x: S | some y : T | F

  (sy: S -> one T) and (all x: S | F[x.sy/y])
  - Introduces a fresh function
Alloy analysis steps

- Initial conversions
- **Translation to a boolean formula**
- Conversion to conjunctive normal form
- Solving using an off-the-shelf SAT solver
- Reconstructing an Alloy solution
Translation to boolean

- The proof obligation is reduced to a proposition using the scope information.

- Satisfiability-preserving with respect to the scope
  
  A relational formula $R$ has a solution within a scope of $s$ if and only if the boolean formula $\text{Translate}(R, s)$ has a solution.
Translation to boolean

- Represent relations by bit vectors
  - A unary relation \( r: A \) (signature, scalar, etc.)
    \[ [r_1, r_2, \ldots, r_n] \]
    - \( r_i \) is a boolean variable and \( n = \text{scope}(A) \)
    (a vector)
  - A binary relation \( r: A \rightarrow B \)
    \[ [r_{11} \ r_{12} \ldots r_{1n}, r_{21} \ r_{22} \ldots r_{2n}, \ldots, r_{m1} \ r_{m2} \ldots r_{mn}] \]
    - \( r_{ij} \) is a boolean variable and \( n = \text{scope}(B) \) and \( m = \text{scope}(A) \)
    (an \( m \times n \) matrix)

- All relational operations are performed on these matrices
- Operations are done bottom up on the abstract syntax tree (AST)
- When we get to the root, we are left with a single boolean formula
Translation to boolean

\[ r : A \rightarrow B , \quad s : A \rightarrow B \]

\[ r + s \]
Translation to boolean

\[ r : A \rightarrow B \quad , \quad s : A \rightarrow B \]

- \( r + s \)
  - A matrix of \((rij \text{ or } sij)\)
- \( r \& s \)
Translation to boolean

\[ r : A \to B, \quad s : A \to B \]

- \( r + s \)
  - A matrix of \((rij \ or \ sij)\)
- \( r \ & \ s \)
  - A matrix of \((rij \ and \ sij)\)
- \( r.s \)
Translation to boolean

\[ r : A \rightarrow B , \quad s : A \rightarrow B \]

- \( r + s \)
  - A matrix of \((rij \text{ or } sij)\)
- \( r \& s \)
  - A matrix of \((rij \text{ and } sij)\)
- \( r.s \)
  - Matrix multiplication
- \( r \text{ in } s \)
Translation to boolean

\[ r : A \rightarrow B , \quad s : A \rightarrow B \]

- \( r + s \)
  - A matrix of \((rij \ or \ sij)\)
- \( r \ & \ s \)
  - A matrix of \((rij \ and \ sij)\)
- \( r.s \)
  - Matrix multiplication
- \( r \ in \ s \)
  - A formula of \(\text{and} \ \{ \ (rij \ implies \ sij) \} \)
- \( r = s \)
Translation to boolean

\[ r : A \rightarrow B, \quad s : A \rightarrow B \]

- \( r + s \)
  - A matrix of \((rij \text{ or } sij)\)
- \( r \& s \)
  - A matrix of \((rij \text{ and } sij)\)
- \( r.s \)
  - Matrix multiplication
- \( r \text{ in } s \)
  - A formula of \(\text{and} \{ (rij \text{ implies } sij) \} \)
- \( r = s \)
  - A formula of \(\text{and} \{ (rij \text{ implies } sij) \text{ and } (sij \text{ implies } rij) \} \)
Example

- not (x.r = y)
- x : A and scope(A) = 2
- y : B and scope(B) = 2
- r : A -> B

- How does the AST look like?
- What is the order of translation?
- What is the final boolean formula?
Alloy analysis steps

- Initial conversions
- Translation to a boolean formula
- Conversion to conjunctive normal form
- Solving using an off-the-shelf SAT solver
- Reconstructing an Alloy solution
Conversion to CNF

- Conjunctive normal form (CNF)
  - A conjunction of clauses
  - A clause is a disjunction of literals
  - A literal is a variable or the negation of a variable
  - Example:
    \[(a \textbf{ or } b \textbf{ or } c) \textbf{ and } (!a \textbf{ or } b \textbf{ or } d) \textbf{ and } (!b \textbf{ or } !c \textbf{ or } !d)\]
    
a: true, b: true, c: false

- Standard conversion technique

- CNF is the standard input language of all SAT solvers
  - Enables Alloy to treat SAT solvers as a black box
  - Can always plug in the SAT solver of your own choice
Alloy analysis steps

- Initial conversions
- Translation to a boolean formula
- Conversion to conjunctive normal form
- **Solving using an off-the-shelf SAT solver**
- Reconstructing an Alloy solution
SAT solving

- Complexity?
  - 2-SAT is polynomial
  - 3-SAT is NP-complete

- DPLL SAT solvers

- Recent developments
  - Annual SAT competitions
  - Extra features: unsat core, MaxSAT, SAT Modulo Theories
DPLL algorithm

function DPLL(p: Boolean formula): boolean {
    if p contains an empty clause
        return false;

    if all variables are assigned
        return true;

    for every unit clause c in p
        p = unit-propagate(c, p);

    for every literal l that is pure in p
        p = pure-literal-assign(l, p);

    l = choose-literal(p);
    return DPLL(p and l) or DPLL(p and not l);
}
DPLL algorithm – example

(a !b !c)
(b)
(c !b d)
(a c)
(!c !b !d)
DPLL algorithm – example

\[(a \neg b \neg c) \quad (a \neg c)\]
\[(b)\]
\[(c \neg b \ d) \quad (c \ d)\]
\[(a \ c) \quad (a \ c)\]
\[(\neg c \neg b \ d) \quad (\neg c \ d)\]

Unit clause: \( b = \text{true} \)
DPLL algorithm – example

(a !b !c) (a !c)
(b)
(c !b d) (c d) (c d)
(a c) (a c)
(!c !b !d) (!c !d) (!c !d)

Unit clause: b = true
Pure literal: a = true
DPLL algorithm – example

(a !b !c)  (a !c)
(b)
(c !b d)  (c d)  (c d)
(a c)  (a c)
(!c !b !d)  (!c !d)  (!c !d)  (!d)

Unit clause: b = true
Pure literal: a = true
Choose: c = true
**DPLL algorithm – example**

(a !b !c) (a !c)
(b)
(c !b d) (c d) (c d)
(a c) (a c)
(!c !b !d) (!c !d) (!c !d) (!d)

Unit clause: b = true
Pure literal: a = true
Choose: c = true
Unit clause: d = false
Alloy analysis steps

- Initial conversions
- Translation to a boolean formula
- Conversion to conjunctive normal form
- Solving using an off-the-shelf SAT solver
- Reconstructing an Alloy solution
Backward translation

- If the SAT solver finds no solution,
  - Alloy reports no solutions exist

- If the SAT solver finds a solution
  - Alloy constructs an Alloy solution based on the boolean solution
    - Generates symbolic values for variables
    - If the boolean variable rij = true then the symbolic pair <Ai, Bj> is included in r

- Revisit the example
  - not (x.r = y)
  - What if x and y are singletons?
Alloy Analysis

How big is the search space for a scope of 3?
Alloy Analysis

- How big is the search space for a scope of 3?
  - A binary relation contributes 9 bits to the state
    - This implies 29 states (512)
  - A tiny model with only 4 binary relations:
    - 236 (over a billion) states

- How does Alloy compute transitive closure?
Alloy Analysis

How big is the search space for a scope of 3?
- A binary relation contributes 9 bits to the state
  - This implies 29 states (512)
- A tiny model with only 4 binary relations:
  - 236 (over a billion) states

How does Alloy compute transitive closure?
- Can’t do fixpoint computation statically
- Computes join n-1 times – by powers of two
  \[ \hat{r} = r^+ = r \cup r \circ r \cup \ldots \cup r^{(n-1)} \]
  - Why sufficient?
Symmetry breaking

- Every Alloy model has a natural symmetry
  - Alloy atoms are uninterpreted
  - Alloy doesn’t allow the user to constrain an atom explicitly
  - So, all atoms of a basic type are interchangeable
  - Take an instance and just permute the atoms

- Divide the set of all instances to equivalence classes
  - Two instances are in the same class if they are permutations of each other
  - Each property either holds for all of them or doesn’t for none of them
  - Example?

- Symmetry breaking is to ensure that only “one” solution in each equivalence class is considered
Symmetry breaking

- Helps the performance when equivalence classes are large

- Done by generating more constraints to pass to the SAT solver
  - These are called symmetry-breaking predicates

- Alloy’s symmetry breaking isn’t perfect in theory, but very useful in practice
  - To eliminate all-but-one solution of each class, will need too many constraints that damage the solver’s performance
Example

- A unary relation \( r \) defined over a signature of scope \( k \) has \( k+1 \) equivalence classes
  - Based on the number of elements in \( r \)
- \( A = [a_0, a_1, a_2] \)
  - 000 (good),
  - 001 (good), 010 (bad), 100 (bad),
  - 011 (good), 101 (bad), 110 (bad),
  - 111 (good)
- Good: \([a_0] \leq [a_1] \leq [a_2]\)  (lexicographic order)
- Boolean predicate for \([a_0] \leq [a_1]\) is \( \neg a_0 \lor a_1 \)
Example

- A unary relation $r$ defined over a signature of scope $k$ has $k+1$ equivalence classes.
  - Based on the number of elements in $r$

- $A = [a_0, a_1, a_2]$
  - 000 (good),
  - 001 (good), 010 (bad), 100 (bad),
  - 011 (good), 101 (bad), 110 (bad),
  - 111 (good)

  Good: $[a_0] \leq [a_1] \leq [a_2]$ (lexicographic order)
  - Boolean predicate for $[a_0] \leq [a_1]$ is $(!a_0 \lor a_1)$

- $B = [b_0, b_1, b_2], r: A \rightarrow B = [v_0 \, v_1 \, v_2, \, v_3 \, v_4 \, v_5, \, v_6 \, v_7 \, v_8]$
  - Lexicographic order of $A$: $[v_0v_1v_2] \leq [v_3v_4v_5] \leq [v_6v_7v_8]$
  - Lexicographic order of $B$: $[v_0v_3v_6] \leq [v_1v_4v_7] \leq [v_2v_5v_8]$

  Then convert to predicates
Symmetry breaking predicates

- Preserve the satisfiability of the formula
- Are true of at least one solution in each equivalence class
- Are true of the smallest possible number of solutions in each equivalence class

- Speed up SAT backtracking search by causing a backtrack whenever all extensions of the current partial variable assignment violate the predicate
- Very effective for unsatisfiable formulas
  - They usually take longer because the whole search space must be considered
- Help satisfiable formulas by excluding solutionless regions of search space
  - But makes hitting a solution harder

- Good for enumerating solutions
Sharing detection

- Grounding out quantified formulas is costly
  - Ground form can contain shared formulas
  - Grounding out first, determining identical formulas later is infeasible due to the size of the ground formula

- Sharing detection determines identical expressions and allows them to be shared before grounding
  - Shared through a DAG
  - To avoid multiple boolean formulas for same identical sub-expression

- Example
  - \textbf{all} p:A, q:B | G(p) \textbf{or} H(G(p), q)
    - \((G(A0) \text{ or } H(G(A0), B0)) \text{ and}
    - \((G(A0) \text{ or } H(G(A0), B1)) \text{ and}
    - \((G(A1) \text{ or } H(G(A1), B0)) \text{ and}
    - \((G(A1) \text{ or } H(G(A1), B1)) \text{ and}

  - \(G(A0)\) and \(G(A1)\) are shared four times
Sharing detection – Template mechanism

- Using a template of $G(?)$, sharing can be detected
  - Remember a pointer to the graph node of $G(A0)$ and use it while grounding out the rest of the formula

- General algorithm:
  - Walk the quantified formulas abstract syntax tree (AST) in DFS order
  - For each node, determine the templates matched by its children, then the template matched by the node
  - Either it matches a previously seen template or create a new template
Universal quantifiers over finite signatures

- Problem occurs when a signature is intended to represent all possible values of an entity
  - Contradicts with Alloy semantics
  - Specially when that signature is used with universal quantifier
- Example:
  `sig Set { elements : set Element } sig Element {}`

  `assert closed {` 

  `all s0, s1 : Set | some s2 : Set | s2.elements = s0.elements + s1.elements }`

- Counterexample:
  Set = { (S0), (S1) } Element = { (E0), (E1) } s0 = { (S0) } s1 = { (S1) }

  elements = { (S0, E0), (S1, E1) }

- Analyzer didn’t populate the signature Set with enough values
  - Add a generator axiom

  `fact SetGenerator {`

  `some s : Set | no s.elements all s: Set, e: Element | some s’: Set | s’.elements = s.elements + e }`

- Space explosion problem (for scope(Element) = k, needs 2k Sets)
Generator axioms

- Sometimes the generator axiom requires an infinite number of atoms:

  ```
  abstract sig List {}
  one sig EmptyList extends List {}
  sig NonEmptyList extends List {
    value : Element,
    rest : List }
  ```

- Generator axiom to populate all lists:

  ```
  fact ListGenerator {
    all l: List, e: Element | some l': List | l'.rest = l and l'.value = e }
  ```

  Unless Element is empty, the axiom makes the model inconsistent, all assertions vacuously true

- Not a good idea to declare lists recursively

  - Use “set” if the order doesn’t matter
Why is Alloy useful then?

- Generator axioms are needed for mathematical objects, but not for real problem domains
  - Don’t arise very often in practice
  - Don’t usually say a directory exists for every possible combinations of files!

- No problem with existential quantifier:
  ```plaintext
  assert UnionCommutative {
    all s0, s1, s2 : Set | s0.elements + s1.elements = s2.elements implies
    s1.elements + s0.elements = s2.elements }
  ```
  - Negated fact contains existential quantifier.. No generator axiom needed

- Bottom line:
  - Finite instance finding may produce spurious counterexamples or vacuously-true checks in theory

- Reference:
  - Relational analysis of algebraic datatypes, Viktor Kuncak and Daniel Jackson, 2005