Rigorous Software Development
CSCI-GA 3033-009

Instructor: Thomas Wies

Spring 2013

Lecture 4

Today’s Topics

- The Alloy Analyzer (Ch. 5 of Jackson Book)
  - From Alloy models to Analysis Constraints
  - Propositional Logic (Ch. 1 of Huth/Ryan Book)
  - From Analysis Constraints to Propositional Logic
  - Quantifier Elimination
  - Alleviating State Space Explosion

Alloy Analyzer (AA)

- **Small scope hypothesis**: violations of assertions are witnessed by small counterexamples
  - AA exhaustively searches for instances of small scope
- AA can falsify a model but not verify it
  - It can prove that an assertion does not hold for all instances of a model by finding a counterexample.
  - It cannot prove that an assertion holds in all instances of a model,
  - it can only prove that an assertion holds for all instances up to a certain size (bounded verification).

Alloy Analyzer (AA)

- **Small scope hypothesis**: violations of assertions are witnessed by small counterexamples
  - AA exhaustively searches for instances of small scope
- Can we automatically verify Alloy models?
  - The answer is no because the verification problem for Alloy models is undecidable
  - i.e., there is no general algorithm to solve this problem.

From Alloy Models to SAT and Back

- AA is actually a compiler
  - First, the alloy model is translated to a single Alloy constraint, which is called the analysis constraint.
  - Given the scope of the command to execute, the analysis constraint is translated into a propositional constraint.
  - AA then uses an off-the-shelf SAT solver to find a satisfying assignment for the propositional constraint.
  - If a satisfying assignment exists, it is translated back into an instance of the original Alloy model.
- AA reduces the problem of finding instances of Alloy models to a well-understood problem: SAT

Analysis Constraints

- First, the Alloy model is translated into a single Alloy constraint: the analysis constraint.
- The analysis constraint is a conjunction of
  - fact constraints
    - facts that are explicitly declared in the model
    - facts that are implicit in the signature declarations
  - and a predicate constraint:
    - for a **`run`** command: the constraint of the predicate that is run
    - for a **`check`** command: the negation of the assertion that is checked

Analysis Constraints: Example

```
module addressBook

abstract sig Target {}
sig Addr, Name extends Target {}
sig Book {addr: Name->Target}

fact Acyclic {all b: Book | no ^(b.addr) & iden}

pred add [b, b’: Book, n: Name, t: Target] {
  b’.addr = b.addr + n->t
}

run add for 3 but 2 Book
```

Implicit Fact Constraint

The implicit fact constraint is the conjunction of the constraints implicit in the signature declarations:

Example: from the signature declarations

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

AA generates the implicit fact constraint:

```
Name in Target
Addr in Target
no Name & Addr
Target in Name + Addr
no Book & Target
```

Explicit Fact Constraint

The explicit fact constraint is the conjunction of all bodies of the declared facts

Example: the fact

```
fact Acyclic {all b: Book | no ^(b.addr) & iden}
```

generates the explicit fact constraint:

```
all b: Book | no ^(b.addr) & iden
```

Predicate Constraint

The predicate constraint is

- the conjunction of the body of the predicate that is run and the multiplicity and type constraints of its parameters
- or the negation of the body of the assertion that is checked.

Example: running the predicate

```
pred add [b, b’: Book, n: Name, t: Target] {
  b’.addr = b.addr + n->t
}
```

`t in Target and one t`

generates the predicate constraint:

```
b: Book and b’: Book and n: Name and t: Target
b’.addr = b.addr + n->t
```

Analysis Constraint for `addressBook`

```
Name in Target
Addr in Target
no Name & Addr
Target in Name + Addr
no Book & Target
all b: Book | no ^(b.addr) & iden
b: Book and b’: Book
n: Name and t: Target
b’.addr = b.addr + n->t
```

Implicit fact constraint

Explicit fact constraint

Predicate constraint

Satisfying Assignment for Analysis Constraint

Satisfying assignment is mapping from constraint vars to relations of atoms that evaluate the constraint to *true*.

Target = $\{(A_0), (N_0)\}$

Addr = $\{(A_0)\}$

Name = $\{(N_0)\}$

Book = $\{(B_0), (B_1)\}$

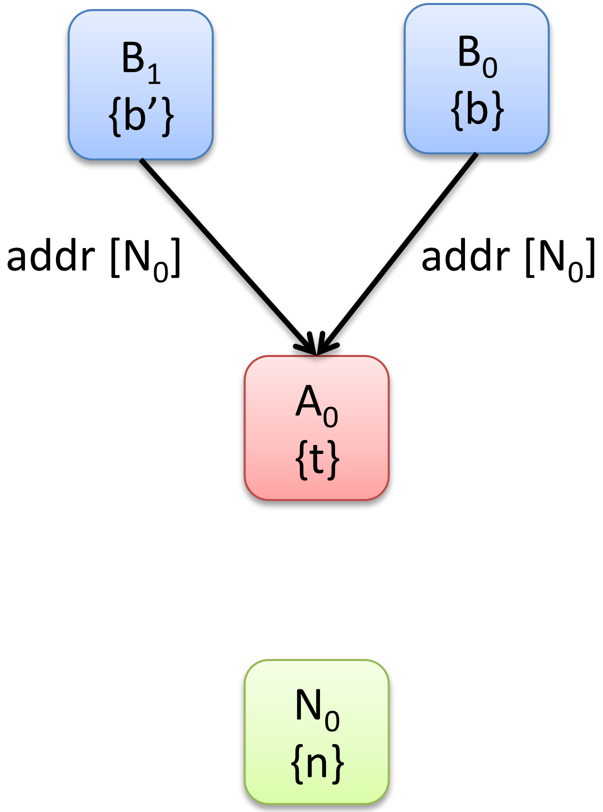
addr = $\{(B_0,N_0,A_0), (B_1,N_0,A_0)\}$

b = $\{(B_0)\}$

b’ = $\{(B_1)\}$

n = $\{(N_0)\}$

t = $\{(A_0)\}$



From Analysis Constraints to Propositional Logic

- Given the scope of the command to execute, the analysis constraint is translated into a constraint in propositional logic.
- The translation guarantees a one-to-one correspondence between satisfying assignments of the propositional constraint and the analysis constraint.
- AA then uses an off-the-shelf SAT solver to find a satisfying assignment for the propositional constraint.
- If a satisfying assignment is found, it is translated back into an assignment of the analysis constraint which in turn represents the instance of the original Alloy model.

What is Logic?

- Like a programming language, a logic is defined by its syntax and semantics.
- Syntax:
  - An alphabet is a set of symbols.
  - A finite sequence of symbols is called an expression.
  - A set of rules defines the well-formed expressions.
- Semantics:
  - Gives meaning to well-formed expressions.
  - Formal notions of induction and recursion can be used to give rigorous semantics.

Syntax of Propositional Logic

- Each expression is made of
  - propositional variables: a, b, . . . , p, q, . . .
  - logical constants: $\top$, $\bot$
  - logical connectives: $\wedge$, $\vee$, $\Rightarrow$, . . .
- Every propositional variable stands for a basic fact
  - Examples:
    I'm hungry, Apples are red, Joe and Jill are married

Syntax of Propositional Logic

- Well-formed expressions are called formulas
- Each propositional variable (a, b, . . ., p, q, . . .) is a formula
- Each logical constant (⊤, ⊥) is a formula
- If ϕ and ψ are formulas, all of the following are also formulas

$$\begin{array}{lll} \neg\phi & \phi \wedge \psi & \phi \Rightarrow \psi \\ (\phi) & \phi \vee \psi & \phi \Leftrightarrow \psi \end{array}$$

- Nothing else is a formula

Semantics of Propositional Logic

- The meaning (value) of ⊤ is always *True*. The meaning of ⊥ is always *False*.
- The meaning of the other formulas depends on the meaning of the propositional variables.
  - Base cases: Truth Tables

P	Q	¬ P	P ∧ Q	P ∨ Q	P ⇒ Q	P ⇔ Q
False	*False*	*True*	*False*	*False*	*True*	*True*
False	*True*	*True*	*False*	*True*	*True*	*False*
True	*False*	*False*	*False*	*True*	*False*	*False*
True	*True*	*False*	*True*	*True*	*True*	*True*

  - Non-base cases: Given by reduction to the base cases
    Example: the meaning of (p ∨ q) ∧ r is the same as the meaning of a ∧ r where a has the same meaning as p ∨ q.

Semantics of Propositional Logic

- An assignment of Boolean values to the propositional variables of a formula is an interpretation of the formula.

P	Q	$P \lor Q$	$(P \lor Q) \land \neg Q$	$(P \lor Q) \land \neg Q \Rightarrow P$
False	*False*	*False*	*False*	*True*
False	*True*	*True*	*False*	*True*
True	*False*	*True*	*True*	*True*
True	*True*	*True*	*False*	*True*

- Interpretations:
  $\{P \mapsto \textit{False}, Q \mapsto \textit{False}\}, \{P \mapsto \textit{False}, Q \mapsto \textit{True}\}, \ldots$
- The semantics of Propositional logic is compositional: the meaning of a formula is defined recursively in terms of the meaning of the formula's components.

Semantics of Propositional Logic

- Typically, the meaning of a formula depends on its interpretation.
  Some formulas always have the same meaning.

P	Q	P ∨ Q	(P ∨ Q) ∧ ¬Q	(P ∨ Q) ∧ ¬Q ⇒ P
False	*False*	*False*	*False*	*True*
False	*True*	*True*	*False*	*True*
True	*False*	*True*	*True*	*True*
True	*True*	*True*	*False*	*True*

- A formula is
  - (un)satisfiable if it is true in some (no) interpretation,
  - valid if it is true in every possible interpretation.
- A formula that is valid or unsatisfiable is called a tautology.

The SAT Problem

- The satisfiability problem for propositional logic (SAT) asks whether a given formula ϕ is satisfiable.
- SAT is decidable.
- Hence, so is validity of propositional formulas.
- However, SAT is NP-complete
- Hence, checking validity is co-NP-complete.

The SAT Problem

- Many problems in formal verification can be reduced to checking the satisfiability of a formula in some logic.
- In practice, NP-completeness means the time needed to solve a SAT problem grows exponentially with the number of propositional variables in the formula.
- Despite NP-completeness, many realistic instances (in the order of 100,000 variables) can be checked very efficiently by state-of-the-art SAT solvers.

Translating the Analysis Constraint

```
Name in Target
Addr in Target
no Name & Addr
Target in Name + Addr
no Book & Target
all b: Book | no ^(b.addr) & iden
b: Book and b’: Book
n: Name and t: Target
b’.addr = b.addr + n->t
```

Characteristic Function of a Relation

Name = $\{(N_0),(N_1),(N_2)\}$

Addr = $\{(A_0),(A_1),(A_2)\}$

address = $\{(N_0,A_0), (N_1,A_1), (N_2,A_1)\}$

Characteristic function of the relation address:

$$\chi_{\text{address}}: \text{Name} \times \text{Addr} \rightarrow \{0,1\}$$

$$\chi_{\text{address}}(N_i,A_j) = 1 \text{ iff } (N_i,A_j) \in \text{address}$$

Characteristic Function of a Relation

Name = $\{(N_0),(N_1),(N_2)\}$

Addr = $\{(A_0),(A_1),(A_2)\}$

address = $\{(N_0,A_0), (N_1,A_1), (N_2,A_1)\}$

Characteristic function of the relation address:

χ_{address}	N_0	N_1	N_2
A_0	1	0	0
A_1	0	1	1
A_2	0	0	0

Propositional Encoding of Relations

χ_{address}	N_0	N_1	N_2
A_0	1	0	0
A_1	0	1	1
A_2	0	0	0

Introduce a propositional variable X_{ij} for every A_i and N_j:

χ_{address}	N_0	N_1	N_2
A_0	X_{00}	X_{01}	X_{02}
A_1	X_{10}	X_{11}	X_{12}
A_2	X_{20}	X_{21}	X_{22}

Propositional Encoding of Relations

χ_{address}	N_0	N_1	N_2
A_2	0	0	0

Each assignment to the propositional variables X_{ij} corresponds to one possible function χ_{address} and thus one possible interpretation of the relation address.

Introduce a propositional variable X_{ij} for every A_i and N_j:

χ_{address}	N_0	N_1	N_2
A_0	X_{00}	X_{01}	X_{02}
A_1	X_{10}	X_{11}	X_{12}
A_2	X_{20}	X_{21}	X_{22}

Translating Relational Operations

- All relational operations in an Alloy constraint are encoded as propositional formulas.
- The propositional variables in the formulas describe the characteristic functions of the relational variables in the Alloy constraint.

χ_{addr}	N_0	N_1	N_2
A_0	X_{00}	X_{01}	X_{02}
A_1	X_{10}	X_{11}	X_{12}
A_2	X_{20}	X_{21}	X_{22}

Propositional Translation: Example

Analysis constraint (scope 3):

`Addr in Target`

Propositional variables for characteristic functions:

`Addr`: A_0, A_1, A_2

`Target`: T_0, T_1, T_2

Propositional encoding of analysis constraint:

$$A_0 \Rightarrow T_0 \wedge A_1 \Rightarrow T_1 \wedge A_2 \Rightarrow T_2$$

Propositional Translation: Example

Analysis constraint (scope 3):

```
address’ = address + n->t
```

Flatten analysis constraint by introducing fresh variables for non-trivial subexpressions.

Flattened analysis constraint:

```
address’ = address + e
e = n->t
```

Propositional Translation: Example

Flattened analysis constraint (scope 3):

```
address’ = address + e
e = n->t
```

Propositional variables for characteristic functions:

address’: $A'_{00}, A'_{01}, A'_{02}, A'_{10}, A'_{11}, A'_{12}, A'_{20}, A'_{21}, A'_{22}$

address: $A_{00}, A_{01}, A_{02}, A_{10}, A_{11}, A_{12}, A_{20}, A_{21}, A_{22}$

e: $E_{00}, E_{01}, E_{02}, E_{10}, E_{11}, E_{12}, E_{20}, E_{21}, E_{22}$

n: N_0, N_1, N_2

t: T_0, T_1, T_2

Propositional Translation: Example

Flattened analysis constraint (scope 3):

```
e = n->t
```

Propositional variables for characteristic functions:

`e`: $E_{00}, E_{01}, E_{02}, E_{10}, E_{11}, E_{12}, E_{20}, E_{21}, E_{22}$

`n`: N_0, N_1, N_2

`t`: T_0, T_1, T_2

Propositional encoding of analysis constraint:

$$\bigwedge_{0 \leq i,j \leq 2} E_{ij} \Leftrightarrow N_i \land T_j$$

Propositional Translation: Example

Flattened analysis constraint (scope 3):

`addr’ = addr + e`

Propositional variables for characteristic functions:

address’: A'_{00}, A'_{01}, A'_{02}, A'_{10}, A'_{11}, A'_{12}, A'_{20}, A'_{21}, A'_{22}

address: A_{00}, A_{01}, A_{02}, A_{10}, A_{11}, A_{12}, A_{20}, A_{21}, A_{22}

e: E_{00}, E_{01}, E_{02}, E_{10}, E_{11}, E_{12}, E_{20}, E_{21}, E_{22}

Propositional encoding of analysis constraint:

$$\bigwedge_{0 \leq i,j \leq 2} A'_{ij} \Leftrightarrow A_{ij} \vee E_{ij}$$

Quantifier Elimination

- Universal and existential quantification over finite sets can be eliminated using finite conjunctions, respectively, disjunctions.

- Example: Replace universal quantifier
  `all x: S | F`
  where `S = {s_0, ..., s_n}` with conjunction
  `F[s_0/x] and ... and F[s_n/x]`

Quantifier Elimination

- Quantifier elimination can be encoded directly in the propositional constraint.
- Example: The universal quantifier

  `all x: Alias | x.addr in Addr`

  can be encoded by the propositional formula

  $$\bigwedge_{0 \leq i,j < n} A_i \wedge R_{ij} \Rightarrow D_j$$

  assuming the scope is n and the propositional variables are A_i for `Alias`, D_i for `Addr`, and R_{ij} for `addr`.

Skolemization

- Existential quantifiers can be treated more effectively using Skolemization
  - Replace top-level existential quantifiers of the form

    ```
    some x: S | F
    ```

    with

    ```
    (xs: S) and F[xs/x]
    ```

    where `xs` is a fresh variable
- Advantage: witness for `x` is made explicit in generated instances

Skolemization

- Skolemization also works for existential quantifiers that appear below universal quantifiers:
  - replace

    `all x: S | some y: T | F`

    with

    `(sy: S->one T) and all x: S | F[x.sy/y]`

    where `sy` is a fresh analysis variable

Symmetries in Satisfying Assignments

Permuting the names of the propositional variables for each characteristic function in a satisfying assignment yields again a satisfying assigment.

Symmetries in Satisfying Assignments

Target = $\{(A_0), (N_0)\}$

Addr = $\{(A_0)\}$

Name = $\{(N_0)\}$

Book = $\{(B_0), (B_1)\}$

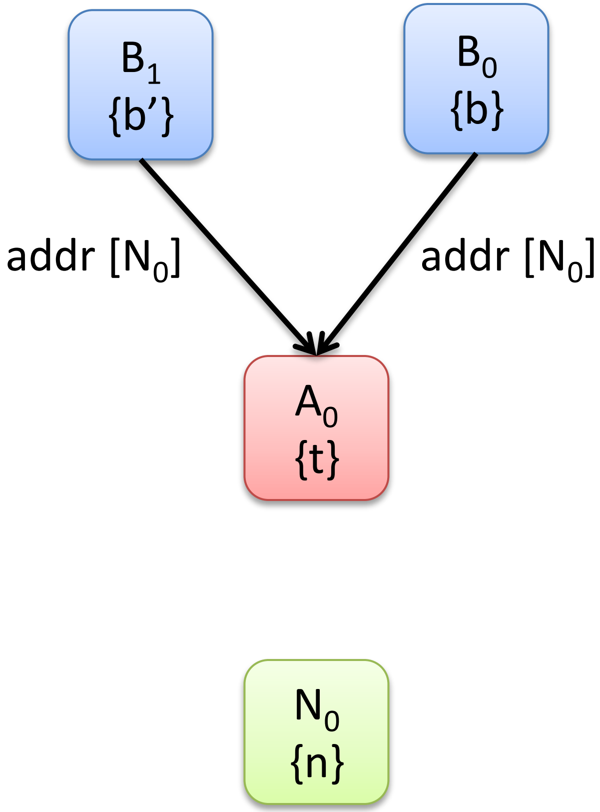
addr = $\{(B_0,N_0,A_0), (B_1,N_0,A_0)\}$

b = $\{(B_0)\}$

b' = $\{(B_1)\}$

n = $\{(N_0)\}$

t = $\{(A_0)\}$



Exchanging the roles of B_0 and B_1 gives a symmetric satisfying assignment.

State Space Explosion Problem

- Symmetries can lead to an exponential blow-up in the number of possible instances.
- This state space explosion problem makes it hard for the SAT solver to solve the propositional constraints.
- Ideally, the SAT solver only has to consider one assignment per equivalence class of symmetric assignments.

Symmetry Reduction

- To reduce the number of symmetries, Alloy adds symmetry breaking constraints to the propositional constraint.
- Example:

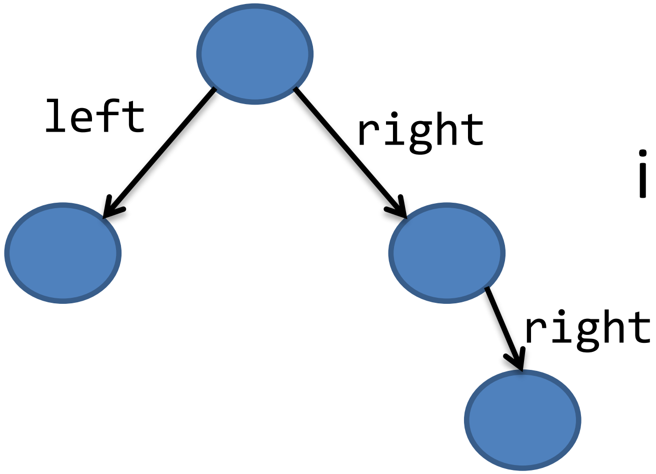
  `util/ordering [Data]`

  - all orderings on `Data` atoms Data0, Data1, Data2, … are symmetric.
  - `util/ordering` enforces one particular ordering on `Data`, namely the lexicographic ordering on atom names:

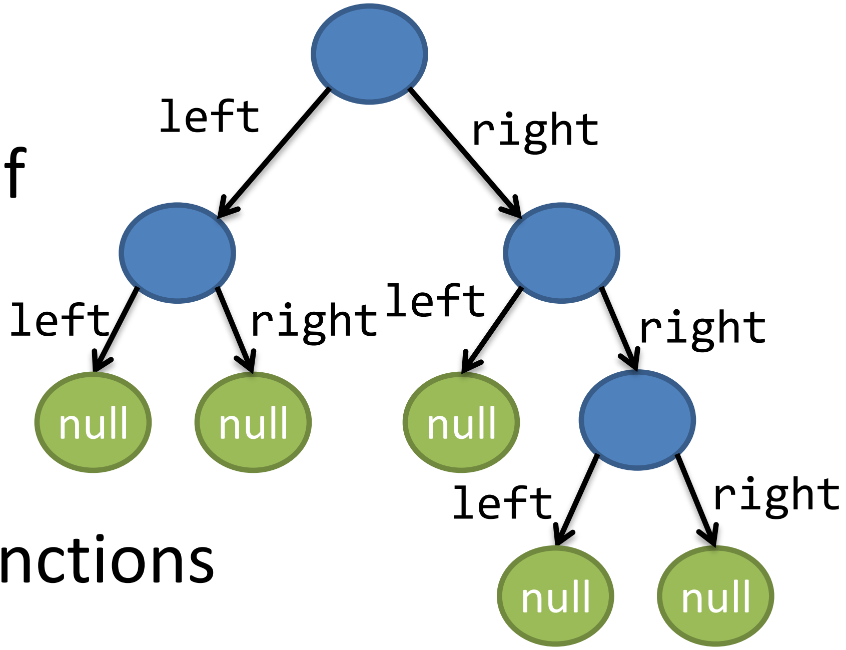
    Data0 < Data1 < Data2 < …

Alleviating State Space Explosion

- Often careful modeling can help to reduce symmetries
- Example: use partial instances when possible



instead of



`left` and `right` are partial functions instead of total functions

Next Week: Design by Contract

- Alloy provides a means for expressing properties of designs
  - Early design refinement saves time
  - Ultimately, we want this effort to impact the quality of implementations
- How can we transition design information to the code?
  - State information (multiplicities, invariants, …)
  - Operations info (pre, post, frame conditions, …)