

## Analysis of Software Architecture

### Lecture 12 CS294-1

### Background

---

- Specification languages have a long history
  - Esp. in software engineering community
  - Long recognized importance of separate specs
- But how do we know a spec makes sense?
  - Before we code
  - Experience shows most specs are buggy
    - Often buggier than the software
    - Why? Not usually testable

### Background (Cont.)

---

- Idea: Check specs automatically
  - For self-consistency
  - For certain properties
- Boils down to automatic analysis of simple, declarative languages
  - The specification languages
  - So-called "software architecture"

### Jackson's Work

---

- A series of papers on analyzing specifications
  - Nitpick
    - An early version of the idea, embodied in a tool
  - Alloy I
    - Relation specifications w/o quantification
  - Alloy II
    - Relational specifications w/quantification
  - Finding bugs in code
    - Applying ideas to programs, not just specifications
- We read only the last two

### An Example

---

- Consider specifying an employee database
- Use *relations* on sets
  - More general than functions
  - Still (somewhat) analyzable
  - Example sets and relations:

worksFor: Employee  $\leftrightarrow$  Company  
hasBoss: Employee  $\leftrightarrow$  Employee

### Example (Cont.)

---

- Everyone is employed by a company
  - Introduce Un, the universal set or relation
  - Which relation/set is meant can be inferred
    - Via type inference

dom worksFor = Un



### Example (Cont.)

- Every employee has at most one boss
  - (postfix) relational transpose  $\sim$
  - relational composition  $\circ$
  - The identity relation  $\text{Id}$

$$\text{hasBoss} \circ \text{hasBoss} \sim \subseteq \text{Id}$$

*Consider the employees of boss X; the set of bosses of those employees is just {X}*

### Example (Cont.)

- The employees of a boss work for the same company as the boss

$$\text{worksFor} \circ \text{hasBoss} \subseteq \text{worksFor}$$

*Any employee's boss' company is the same as the employee's company*

### Relational Expressions

Expr ::=	Id	
	Un	
	0	
	Expr $\cup$ Expr	
	Expr $\cap$ Expr	
	Expr $\setminus$ Expr	relational difference
	Expr ; Expr	
	Expr $\sim$	
	Expr $^+$	transitive closure
	dom Expr	
	Expr $\leftarrow$ Expr	domain restriction
	Var	

### Formulas

Formula ::=	Expr = Expr
	Expr $\subseteq$ Expr
	$\neg$ Formula

### Models

- A *model* of a formula is a collection of sets/relations that satisfy the formula

- Example

$$\begin{aligned} \text{Employee} &= \{e_1, e_2, e_3\} \\ \text{Company} &= \{c_1, c_2, c_3\} \\ \text{worksFor} &= \{e_1 \rightarrow c_1, e_2 \rightarrow c_1, e_3 \rightarrow c_2\} \\ \text{hasBoss} &= \{e_1 \rightarrow e_2\} \end{aligned}$$

### Goal

- In applications, we want to answer one of two questions:

- Is a formula *consistent*?
  - Does it have a model?
- Is a formula *valid*?
  - Is every possible assignment a model?



## Consistency Checking

---

- Note  
 $F$  is valid  $\Leftrightarrow \neg F$  is inconsistent
- So either is sufficient

## A Problem

---

- Consistency is undecidable for this language
- But any reasonable logic for specification is undecidable, too
- What to do?

## The Leap

---

- If a formula is invalid, there must be a counterexample
- Observation: Often there is a small counterexample
- The Leap: Maybe there is nearly always a small counterexample

## The Idea

---

- Fix sets at some small size
  - Say, 2,3, 4, or 5
  - The *scope*
- Check for consistency within scope  $k$ 
  - All sets of size  $k$
  - Clearly decidable
    - Only finitely many possibilities

## Algorithms

---

- While decidable, still very expensive
  - More than doubly exponential in the scope
- We need a way to search the space of possible counterexamples efficiently

## The Idea

---

- Represent formula as a boolean formula
  - A big boolean formula
- Use standard boolean formula satisfiability packages to find satisfying assignment
- Translate satisfying assignment back to model of the formula



## Relations as Tables

- Consider a relation  $T \leftrightarrow S$
- Represent  $T \leftrightarrow S$  as a table  $M$  of booleans
  - $M_{ij}$  is 1 if the  $i$ th element of  $T$  is related to the  $j$ th element of  $S$
  - $M_{ij}$  is 0 otherwise
- If the scope is finite,  $M$  is finite

## An Example w/Scope 3

Employee =  $\{e_1, e_2, e_3\}$

Company =  $\{c_1, c_2, c_3\}$

worksFor =  
 $\{e_1 \rightarrow c_1, e_2 \rightarrow c_1, e_3 \rightarrow c_2\}$

hasBoss =  $\{e_1 \rightarrow e_2\}$

worksFor

1	0	0
1	0	0
0	1	0

hasBoss

0	1	0
0	0	0
0	0	0

## Constraints

- The previous example is a particular model
- The problem is to compute such a model
  - Need to capture all possibilities
  - And narrow them down to the actual models
- Move from booleans to constraints on boolean variables

## Translation for Scope 3

For relation  $M$ , fresh boolean variables  $M_{ij}$

$M(1,1)$	$M(1,2)$	$M(1,3)$
$M(2,1)$	$M(2,2)$	$M(2,3)$
$M(3,1)$	$M(3,2)$	$M(3,3)$

## Translation for Scope 3 (Cont.)

Expr ::= Id	
Un	Id, Un, 0 constant matrices
0	
Expr <sub>1</sub> ∪ Expr <sub>2</sub>	Expr(i,j) = Expr <sub>1</sub> (i,j) ∨ Expr <sub>2</sub> (i,j)
Expr <sub>1</sub> ∩ Expr <sub>2</sub>	Expr(i,j) = Expr <sub>1</sub> (i,j) ∧ Expr <sub>2</sub> (i,j)
Expr <sub>1</sub> \ Expr <sub>2</sub>	Expr(i,j) = Expr <sub>1</sub> (i,j) ∧ ¬Expr <sub>2</sub> (i,j)
Expr <sub>1</sub> ; Expr <sub>2</sub>	Expr(i,j) = (Expr <sub>1</sub> (i,1) ∧ Expr <sub>2</sub> (1,j)) ∨ (Expr <sub>1</sub> (i,2) ∧ Expr <sub>2</sub> (2,j)) ∨ (Expr <sub>1</sub> (i,3) ∧ Expr <sub>2</sub> (3,j))
Expr <sup>+</sup>	Expr; Expr; Expr
dom Expr	Expr(i) = Expr(i,1) ∨ Expr(i,2) ∨ Expr(i,3)
Expr <sub>1</sub> < Expr <sub>2</sub>	Expr(i,j) = Expr <sub>2</sub> (i,j) ∧ dom Expr <sub>1</sub>
Var	

## Translation of Formulas

Formula ::= Expr <sub>1</sub> = Expr <sub>2</sub>	∧ <sub>i,j</sub> Expr <sub>1</sub> (i,j) ⇔ Expr <sub>2</sub> (i,j)
Expr <sub>1</sub> ⊆ Expr <sub>2</sub>	∧ <sub>i,j</sub> Expr <sub>1</sub> (i,j) ⇒ Expr <sub>2</sub> (i,j)
¬Formula	¬Formula



## Satisfiability

- Now we have a boolean formula
- Submit to a satisfiability checker
  - First convert to CNF
  - May find an assignment
  - May report there is no assignment
    - Or give up
- An assignment can be converted to a model of the original formula

## Rationale

- Reduction to boolean satisfiability not obviously a good idea
  - Huge blowup in the formulas
  - Direct solver for relational formulas might be better
- But, SAT solvers are highly refined
  - And available
  - And getting better all the time

## Quantification

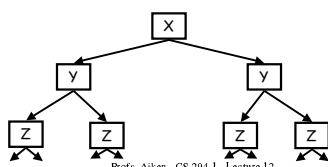
- Add quantifiers to expressions:
    - $\forall x. \text{Expr}$
    - $\exists y. \text{Expr}$
  - More expressive than relations
    - And arguably more natural, too
- $$\forall x, y. (\text{hasBoss}(x,y) \Rightarrow (\exists c. \text{worksFor}(x,c) \wedge \text{worksFor}(y,c)))$$

## Implementing Quantification

- Implementation is standard
  - For people steeped in programming languages
- Structurally defined, as before
  - Keep track of an *environment*
    - The set of free variables in the sub-expression

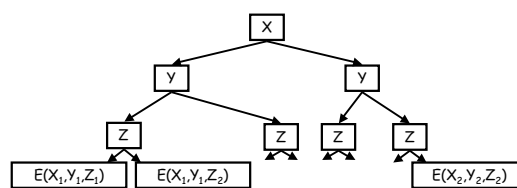
## Visually

- Each free variable may take on  $k$  values
  - $k$  = scope
  - For  $n$  free variables, represent the  $k^n$  possibilities as a full  $k$ -ary tree
  - Example: 3 variables, scope 2



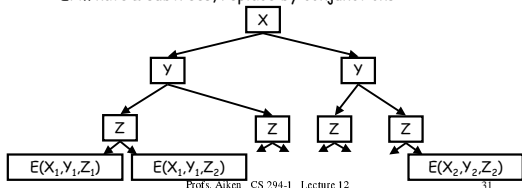
## Visually (Cont.)

- Each leaf has the same expression
  - With different boolean variables plugged in
  - Expressions are computed as before



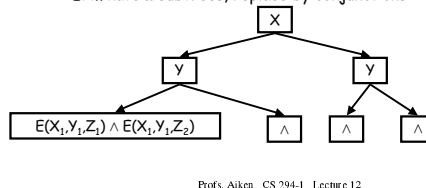
### Closing $\forall$

- To process  $\forall z.E$ 
  - Maintain variables in nesting order
    - So  $z$  is at the fringe of the tree for  $E$
  - Build expressions for  $E$
  - Eliminate  $z$  subtrees, replace by conjunctions



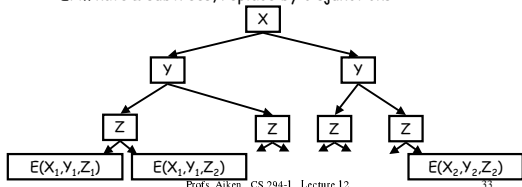
### Closing $\forall$

- To process  $\forall z.E$ 
  - Maintain variables in nesting order
    - So  $z$  is at the fringe of the tree for  $E$
  - Build expressions for  $E$
  - Eliminate  $z$  subtrees, replace by conjunctions



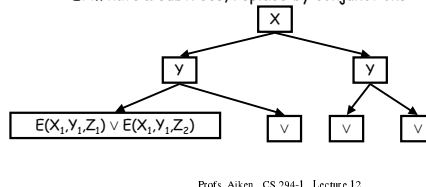
### Closing $\exists$

- To process  $\exists z.E$ 
  - Maintain variables in nesting order
    - So  $z$  is at the fringe of the tree for  $E$
  - Build expressions for  $E$
  - Eliminate  $z$  subtrees, replace by disjunctions



### Closing $\exists$

- To process  $\exists z.E$ 
  - Maintain variables in nesting order
    - So  $z$  is at the fringe of the tree for  $E$
  - Build expressions for  $E$
  - Eliminate  $z$  subtrees, replace by disjunctions



### Details

- Subformulas may have different #s of vars
  - Need operation to add missing vars to each
  - Easy: add missing levels in the tree
- Top-level formula is closed
  - No free variables
  - So final result is just a boolean formula

### Scalability

- SAT is already NP-hard
- But model-finding for scope  $k$  is worse
  - Much worse
- Implies producing the formula is expensive
  - Need optimizations to reduce formula size
  - Many; discuss three



## Symmetry Breaking

---

- Observation
  - Elements of relations are uninterpreted
  - So their order doesn't matter
  - Idea: fix an order to reduce formula size
- Example:
  - $P \neq Q$  translates to  $\forall_{ij} P(i,j) \neq Q(i,j)$
  - Size is  $k^2$  for scope  $k$
  - Instead, just pick the 0,0 element:  $P(0,0) \neq Q(0,0)$
  - Can break symmetry only once per type

## Negation Caching

---

- Consider relational composition  $E = P; R$
- The formula is  $E(i,j) = \forall_k P(i,k) \wedge R(k,j)$ 
  - In CNF, this has  $2^k$  clauses of  $k$  literals
- Consider the negation  $\neg E(i,j) = \bigwedge_k \neg P(i,k) \vee \neg R(k,j)$ 
  - In CNF already, with  $k$  clauses of 2 literals
- Idea: Cache the negation of compositions

## Temporary Variable Introduction

---

- Negations can lead to huge blowup
  - Especially for complex expressions
- Introduce a fresh variable for every subexpression
  - $P; Q; R \Rightarrow X = P; Q \quad Y = X; R$
- Benefits
  - Negation does less damage to smaller expressions
  - Common subexpression elimination

## Checking Programs

---

- We can also apply this to checking code
  - Not just checking specifications
- Need to extract a formula from a program
  - Must model control and dataflow

## Starting Point

---

- We will do this only for finite programs
  - No loops or recursion
- How?
  - Unroll each loop  $N$  times
- Weak justification
  - We are only working with bounded scope anyway, so we should be able to bound executions as well

## Modeling Control Flow

---

- Number all program points
- For a transition from point  $i$  to point  $j$ 
  - Introduce boolean variable  $E_{ij}$ 
    - True iff edge is taken in execution
  - Add constraints saying that for each node executed, exactly one of the successor nodes is executed



## Modeling Dataflow

---

- Need to fix the scope
  - Limits the number of elements of each data type in the program state
- Introduce variable for each state element
  - At every program point
  - Fully flow-sensitive
- Constraints relate how state changes
  - Update elements change according semantics
  - Unchanged elements have = constraints between one control point and the next

## The Formula

---

- Form the formula

$$\text{Pre} \wedge \text{Program-Model} \wedge \neg \text{Post}$$

- Models of this formula
  - Start in a valid state (the precondition)
  - Execute respecting the control- and data-flow of the program
  - End in a state violating the postcondition

## Opinions

---

- This approach is hopelessly non-scalable
  - The formulas grow out of control very quickly
- But, in some sense it isn't supposed to scale
  - Only work with "small" worlds
  - Look for small counterexamples to beliefs about the system
- And, there is only one approximation
  - Scope
  - Everything else is faithfully modeled
- Summary: This is interesting . . .

