Example: Dataflow Analysis

- **intent:**
  - stated as a property
  - captured as an event sequence

- **behavior:**
  - model represents some execution characteristics
  - inferred from a model: (e.g., annotated flow graph)
  - inferences based upon:
    - semantics of flow graph
    - semantics captured by annotations

- **comparison:**
  - done by a fsa (e.g., a property automaton)
global dataflow analysis

- classes
  - forward flow problems (e.g., available expressions)
    - what definitions can affect computations at a given point in a program
  - backward flow problems (e.g., live variables)
    - what uses (references) that follow a given point in the program can be affected by computations up to that point

- paths
  - any path
  - all path

Anomalous pairs of ref/defs

d - defined, r - referenced, u - undefined

<table>
<thead>
<tr>
<th></th>
<th>d</th>
<th>u</th>
<th>r</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>dd</td>
<td>bug?</td>
<td>du</td>
<td>bug?</td>
<td></td>
</tr>
<tr>
<td>dr</td>
<td>normal</td>
<td>ud</td>
<td>normal</td>
<td></td>
</tr>
<tr>
<td>uu</td>
<td>harmless?</td>
<td>rr</td>
<td>normal</td>
<td></td>
</tr>
<tr>
<td>rd</td>
<td>normal</td>
<td>ru</td>
<td>normal</td>
<td></td>
</tr>
<tr>
<td>ur</td>
<td>bug</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

P(1,2,3,4; α) = d11r = dr

unreferenced definition

undefined reference
Data Flow Analysis

"property" e.g., unref def

characterize as:
any/all paths forward/backward

use compiler optimizing techniques to propagate to fixed point

Comparison

model
Unreferenced definitions

```
int x, y;
...  
x := 3;
y := x + 2;
if x > 0 then
  x := x + y;
end if;
y := ...
```

Forward flow, all paths problem

General Approach

- Initial values
  - for each node define gen and kill information
- Input Equations
  - for each node we have an equation of the form:
    \[ \text{In}_i := \text{Merge} (\text{Out}_j) \]
  - "Merge" operation over the "predecessors" of \( n_i \)
General Approach

- **Transfer Equations**
  - for each node we have an equation of the form: \( \text{Out}_i := f_i(\text{In}_i) \)
  - Transfer functions usually depend on Gen/Kill information that is computed for each node
  - Usually: \( \text{Out} := (\text{In} - \text{kill}) \cup \text{gen} \)
- We can view the set of variables, transfer functions, and flow graph as a system of equations

Available expressions, forward flow, all paths
- \( \text{In}_i = \cap \text{Out}_j \)
- \( \text{Out}_i := (\text{In} - \text{kill}) \cup \text{gen} \)
- \( \text{Kill} = \text{undefs} \)
- \( \text{Gen} = \text{defs} \)

Live Variables, backward flow, any paths
- \( \text{In}_i = \cup \text{Out}_j \)
- \( \text{Out}_i := (\text{In} - \text{kill}) \cup \text{gen} \)
- \( \text{Kill} = \text{defs} \)
- \( \text{Gen} = \text{refs} \)
- Keep propagating until reach a fixed point solution

In(i) := Merge (Outi)
worklist algorithm

1. Start at initial node (entry for forward; exit for reverse), label \(IN_0\) with pertinent "facts" (initial values)
2. Compute \(OUT_0 = F(IN_0)\) (label \(OUT_0\) with the computed facts)
3. Propagate \(OUT_0\) to \(IN\) (label edge \(N_0 \rightarrow N_i\) with \(OUT_0\)) where \(N_i\) are successor nodes (forward) or predecessor nodes (reverse) of \(N_0\)
4. Compute \(OUT_i = F(IN_i)\), place all \(N_i\) on a "worklist" \(W\), and for all \(N_i\) label \(OUT_i\) with the computed facts.
5. While \(W\) is not empty,
   1. pick \(N_i\) from \(W\) and propagate \(OUT_i\) to \(IN\) (label edges \(N_i \rightarrow N_k\) with \(OUT_i\)) where \(N_k\) are successor nodes (forward) or predecessor nodes (reverse) for \(N_i\); delete \(N_i\) from \(W\)
   2. Compute \(OUT_k = F(IN_k)\) for all \(N_k\) where \(IN_k = \text{MERGE all input edge labels} (\text{MERGE} = \cup \text{ for "some paths" and } \cap \text{ for "all paths"}), label \(OUT_k\) with the computed facts); and if for \(N_k\), \(OUT_k\) changes put \(N_k\) on \(W\)
6. If \(W\) is not empty, then \(W=W'\) and go to 5

Cecil: Olender and Osterweil

- Instead of implicitly defined facts, let the user define application-specific facts
- Represented as a Deterministic Finite State Automaton (DFSA) or as a Quantified Regular Expression (QRE)
- Events
  - Recognizable events
  - Method calls
  - Can reason about sequences of method calls
  - E.g., \(\text{Push}\) must be called before \(\text{Pop}\)
  - Thread interactions
  - \(\text{Join}\) or \(\text{Fork}\)
  - Arbitrary operations
  - \(\ast\)
- Need to be able to treat events as indivisible actions
  - E.g., can treat \(\text{pop}\) and \(\text{push}\) as atomic as long as they do not contain any events of concern
- Propagate the states in the DFSA that can reach each node in the program
Using Quantified Regular Expressions

- Alphabet, quantification, regular expression
- For the events \{open, close, move\} show that for all paths:
  \[((\text{close} \lor \text{move})^*, (\text{open}^* \lor \text{open}^* \lor \text{close})^*)\]

Data Flow Analysis

"property" = Cecil constraint

- Intent: if dfa accepts all traces then the constraint holds for all computations
- Comparison: dfa defined by Cecil constraint
- Trace = computation along path in an annotated dataflow graph

©Rick Adrion 2005 (except where noted)
State Propagation

- States of the property are propagated through the CFG
- The property is proved if only accepting (non-accepting) states are contained in the final node of the CFG
- Cecil DFSA ->
  - lattice \((\hat{\mathcal{F}}(S), \subset, \cup)\)
  - function space \(\delta : \hat{\mathcal{F}}(S) \rightarrow \hat{\mathcal{F}}(S)\)
  - facts at nodes are elements of \(\hat{\mathcal{F}}(S)\)
- Propagate until convergence and check if terminal node in an accepting state of DFSA

Elevator Controller

```c
void main()
{
  1: if (elevatorStopped)
  3:   openDoors();
  ...  
  5: if (elevatorStopped)
  7:   closeDoors();
  9:   moveToNextFloor();
}
```

- States of the property are propagated through the CFG
- For an **all** property: the property is proved if only accepting states are contained in the final node of the CFG
- For a **none** property: the property is proved if only non-accepting states are contained in the final node of the CFG
State propagation

0: if (elevatorStopped) {...
3: openDoors(); <0>
5: if (elevatorStopped) {...
7: closeDoors(); <0,1>
9: moveToNextFloor(); <0,2>

Worklist: 0, 1, 2

Violation

State propagation

0: if (elevatorStopped) {...
3: openDoors(); <0>
5: if (elevatorStopped) {...
7: closeDoors(); <0,1>
9: moveToNextFloor(); <0,2>

Worklist: 0, 1, 2

Violation
**Approaches**

- Static Analysis
  - Inspections
  - Software metrics
  - Symbolic execution
  - Dependence Analysis
  - Data flow analysis
  - Software Verification

- Dynamic Analysis
  - Assertions
  - Error seeding, mutation testing
  - Coverage criteria
  - Fault-based testing
  - Specification-based testing
  - Object-oriented testing
  - Regression testing

**Verification**

- two well-established approaches
- (Automated) mathematical reasoning
  - theorem proving
  - proof checking
- Finite-state verification
  - model checking
    - Logic spec + FSA comp model ⇒ symbolic model checking
    - FSA spec + FSA comp model ⇒ automata-theoretic model checking
  - property checking
**Verification**

- How are they different?
  - (Automated) mathematical reasoning
    - difficult, error prone
  - decidability vs. expressiveness
    - Propositional calculus is decidable
    - Predicate calculus is semi-decidable
- Finite-state verification
  - Reason about a finite model of the system
  - Fast, yields counterexamples, manages partial specifications, applies to concurrency
  - State explosion!

**Proof**

- Intent
- lemmas and theorems in predicate logic
- typically inferred by symbolic execution of the specifications
- model/product
- Behavior
Static Analysis

Floyd Inductive Proof

intent → predicate logic assertions

Floyd Method of Inductive Assertions

- Show that given the input assertions, after executing the program, program satisfies output assertions
- Show that each program fragment behaves as intended
- Use induction to prove that all fragments, including loops, behave as intended
- Show that the program must terminate
- Informal description
  - Place assertions at the start, final, and intermediate points in the code.
  - Any path is composed of sequences of program fragments that start with an assertion, are followed by some assertion free code, and end with an assertion
    - $A_s, C_1, A_2, C_2, A_3, \ldots, A_{n-1}, C_{n-1}, A_f$
  - Show that for every executable path, if $A_s$ is assumed true and the code is executed, then $A_f$ is true
Why does this work?

- Suppose P is an arbitrary path through the program
- Can denote it by
  \[ P = A_0 \ C_1 \ A_1 \ C_2 \ A_2 \ldots \ C_n \ A_n \]
- Where
  - \( A_0 \) - Initial assertion
  - \( A_n \) - Final assertion
  - \( A_i \) - Intermediate assertions
  - \( C_i \) - Loop free, uninterrupted, straight-line code

If it has been shown that

\[ \forall \ i, 1 \leq i < n: A_i C_i \Rightarrow A_{i+1} \]

Then, by transitivity

\[ A_0 \Rightarrow \ldots \Rightarrow A_n \]

Obvious problems

- How do we do this for a path?
- How do we do this for all paths?
  - Infinite number of paths
    - Must find a way to deal with loops
Find loop invariant \((A_i)\)

- subpaths to consider:
  - \(C_1\) Initial assertion \(A_0\) to final assertion \(A_f\)
  - \(C_2\) Initial assertion \(A_0\) to \(A_i\)
  - \(C_3\) \(A_i\) to \(A_i\)
  - \(C_4\) \(A_i\) to final assertion \(A_f\)
- Basically an inductive proof

The “Aha!” moment - finding invariants is hard!

Wensley’s Algorithm

```plaintext
Procedure Wensley (P:input, Q:input, E:input, Y:output);
    Declare P, Q, E, Y, A, B, D real;
    A := 0.0;
    B := Q/2.0;
    D := 1.0;
    Y := 0.0;
    Do_While (D>=E)
        If ~(P - A - B ≥ 0.0) then
            { Y := Y+(D/2.0);
                A := A+B};
        B := B/2.0;
        D := D/2.0;
    End_do;
End Wensley;
```

Input data:
- \(A\): 0.0
- \(B\): \(Q/2\)
- \(D\): 1.0
- \(Y\): 0.0

Conditions:
- \(D\geq E\)
- \(P-A-B < 0.0\)
Floyd Proof: Wensley's Algorithm

- Summary of Five Lemmas Needed
  - $A_0$ to $A_I$
  - $A_I$, true branch, to $A_I$
  - $A_I$, false branch, to $A_I$
  - $A_I$, true branch, to $A_F$
  - $A_I$, false branch, to $A_F$

`code`

$A_I$: 

\[
(A = Q \times Y) \land (B = Q \times (D/2)) \\
\land (k \geq 0, k \text{ integer} \land D = 2^k) \\
\land ((P/Q) - D) \leq Y \leq (P/Q)
\]

\[
D \geq E \quad \text{[constraint]} \\
P - A - B \geq 0 \quad \text{[constraint]} \\
Y = Y + (D/2) \\
A = A + B \\
B = B/2 \\
D = D/2
\]

Lemma III: $A_I$, false branch, to $A_I$

$A'_I$: 

\[
(A' = Q' \times Y') \land (B' = Q' \times (D'/2)) \\
\land (k \geq 0, k \text{ integer} \land D' = 2^k) \\
\land ((P/Q) - D') \leq Y' \leq (P/Q)
\]
**proof of lemma III**

\[ A_1 \Rightarrow A'_1; ((A' = Q*Y') \land (B' = Q*(D'/2)) \land (k \geq 0, k \text{ integer} \land D' = 2^{-k}) \land ((P/Q) - D') < Y' \leq (P/Q)) \]

we have

\[ A' = A + B; \quad B' = B/2.0; \quad D' = D/2.0; \quad Y' = Y + D/2.0; \]

1) \[ A' = A + B = Q*Y + Q*(D/2) = Q*(Y + (D/2)); \quad Y' = Y + (D/2); \quad \therefore A' = Q*Y' \]

2) \[ B' = B/2 = (Q* D/2)/2; \quad D' = D/2 \]

\[ \therefore B' = (Q*2D/2)/2 = Q*D/2 \]

and so on … basically using symbolic evaluation
Hoare axiomatic proof

- assertions are preconditions and post conditions on some statement or sequence of statements
  \[ P(S)Q \]
- if \( P \) is true before \( S \) is executed and \( S \) is executed then \( Q \) is true
- as in Floyd's inductive assertion method, we construct a sequence of assertions, each of which can be inferred from previously proved assertions and the rules and axioms about the statements and operations of the program
- to prove \( P(S)Q \), we need some axioms and rules about the programming language

Hoare axioms and proof rules

- axiom of assignment
  \[ P \{ x:=f \} Q, \]
  where \( Q \) is obtained from \( P \) by substituting \( f \) for all occurrences of \( x \) in \( P \) (symbolic execution)
- rule of composition
  \[ P \{ S1, S2 \} Q \Rightarrow 3 \ P1, P(S1)P1 \land P(S2)Q \]
- rule for the alternative statement
  \[ P\{if B then S1 else S2 \}Q \Rightarrow P\{B \land S1 \}Q \land P\{\neg B \land S2\}Q \]
- rules of consequence
  \[ [P \{ S \} Q \land Q \Rightarrow R] \Rightarrow P \{ S \} Q \]
  \[ [P \{ S \} Q \land R \Rightarrow P] \Rightarrow R \{ S \} Q \]
- rule of iteration
  \[ P\{while B do S \}Q \Rightarrow P\{\neg B \}Q \land 3 I \ P \{ B \land S \} I \land I\{B \land S \} I \land I\{\neg B \}Q \]

loop invariant

backwards substitution
Proof

- Hoare-style and Floyd-style verification are essentially the same
  - one is based on graphical representation and the other on a textual representation.
  - In Floyd-style proof, we visualize the proof goal by annotating a CFG
  - In the other, we define the proof goal as a Hoare triple

- Mechanism for applying proof
  - may work either direction on such a proof, but because it's typically easier to work backwards, often use a technique called backwards substitution
  - we work our way from the post-condition, using the proof rules to "push formulas through" the program
  - at each point where a "pushed-through" predicate "runs into" a supplied predicate, we have a verification condition (VC) that must be proved.
  - After all VCs are proved, we need to be prove termination
    - Without a termination proof, we achieve partial correctness
    - With a termination proof, we achieve total correctness

Approaches

- Static Analysis
  - Inspections
  - Software metrics
  - Symbolic execution
  - Dependence Analysis
  - Data flow analysis
  - Software Verification

- Dynamic Analysis
  - Assertions
  - Error seeding, mutation testing
  - Coverage criteria
  - Fault-based testing
  - Specification-based testing
  - Object-oriented testing
  - Regression testing
**Proof**

predicate logic assertions

Intent

lemmas and theorems in predicate logic

typically inferred by symbolic execution of the specifications

model/product

Behavior

---

**Symbolic Evaluation/Execution**

- Creates a functional representation of a path of an executable component
- \( P \) is composed of partial functions corresponding to the executable paths
  \[ P = \{ P_1, \ldots, P_r \} \]
  \[ P_i : X_i \rightarrow Y \]
- For a path \( P_i \)
  - \( D[P] \) is the domain for path \( P_i \)
  - \( C[P] \) is the computation for path \( P_i \)
### Execution tree (Hantler-King)

```
ABSOLUTE
assume (true)
1 procedure(X);
2 declare X,Y integer
3 if X<0
4 then Y ← -X;
5 else Y ← X;
6 return (Y);
7 end;
```

prove((Y = X') | Y = -X') & Y ≥ 0 & X = X')

PC: true, PV: X: α, Y: -
PC: true
PC: α<0
PC: α≥0

### Loops -- unroll them?

```
input assertion
n do_while predicate1
n+1 if predicate2
n+2 then code;
n+3 else code;
n+4 end;
n+5 output assertion;
```

better: find a loop invariant
Straightforward Observations

- **Problems**
  - formal proofs are long, tedious and are often hard; assertions are hard to get right; invariants are difficult to get right (need to be invariant, but also need to support overall proof strategy)
  - Unsuccessful proof attempt ⇒ ???
    - incorrect software? assertions? placement of assertion? inept prover? although failed proofs often indicate which of the above is likely to be true (especially to an astute prover)
- **Deeper Issues**
  - undecidability of predicate calculus ⇒ no way to be sure when you have a false theorem
  - there is no sure way to know when you should quit trying to prove a theorem (and change something)
  - proofs are generally much longer than the software being verified ⇒ errors in the proof are more likely than errors in the software being verified

Model Checking: Overview

- **properties** usually expressed in
  - in a propositional logic (e.g., temporal logic)
  - as a FSA
- **system** represented as a (possibly "abstracted") reachability graph
- **reasoning engine**
  - logic ⇒ propagates valid sub-formulas through the graph
  - FSA ⇒ compares FSAs via language inclusion; reachability; or bisimulation
Conservative Analysis

- If property is verified, property holds for all possible executions of the system
- If property is not verified:
  - an error found
  - a spurious result
- System model abstracts information to be tractable
  - Conservative abstractions usually over-approximate behavior
  - If inconsistency relies upon over-approximations, then a spurious result
  - e.g. all counter example correspond to infeasible paths

Temporal logic

- augments the standard operators of propositional logic with "tense" operators
- "possible worlds semantics" ⇒ Kripke model
  - relativize the truth of a statement to temporal stages or states
  - a statement is not simply true, but true at a particular state
  - states are temporally ordered, with the type of temporal order determined by the choice of axioms.
- model of time
  - partially ordered time
  - linearly ordered time
    - linear temporal logic is typically extended by two additional operators, "until" and "since"
  - discrete time
  - branching (nondeterministic) time
    - foundation for one of the principal approaches to verifying concurrent systems = Computational Tree Logics.
Computation Tree Logics

- specification language
  - a propositional temporal logic.
- verification procedure
  - exhaustive search of the state space of the concurrent system to determine truth of specification.
- formulas constructed from path quantifiers and temporal operators:
  - path quantifier:
    - A “for every path”
    - E “there exists a path”
  - temporal operator:
    - Xp “p holds next time”
    - Fp “p holds sometime in the future”
    - Gp “p holds globally in the future”
    - pUq “p holds until q holds”

Architecture of FSV Systems

- Property
  - Property Translator
    - Property Representation
      - System Model
        - System Translator
          - Reasoning Engine
            - Counter Examples for Model
              - Property Verified
**mutual exclusion protocol**

- Example: processes can be null, trying to obtain the lock, or in a critical region (n1, t1, c1) or (n2, t2, c2).
  - TURN is a variable that indicates which process can obtain the lock (0,1,2).
  - Need a reachability graph that shows that states (i.e., the values) of the variables.

- Math symbols:
  - `AG(t1 \Rightarrow AF c1)`
  - `t1 \Rightarrow AF c1`

**Example: propagation**

- `A \Rightarrow B` means (B or \neg A)
- `t1 \Rightarrow AF c1` means (AF c1 v \neg t1)

- Reachability graph diagram with states and transitions.
Automata-Theoretic Model Checking

- Properties stated as an FSA
- Intent
- Language containment
- Reachability analysis
- Bisimulation

Example

- Specification:
  - Of the possible observable events (a, b, c), c must happen at least once

Example

- Implementation
- Accepted by?

Example

- (ba)*(ac*+ bbc*)
Some observations

- Model Checking
  - worst case bound linear in size of the model
  - but the model is exponential
  - not clear if model checking or symbolic model checking is superior
    - depends on the problem
  - experimentally often very effective!
    - used selectively to verify hardware designs
    - trying to develop appropriate abstractions to make it applicable to software systems

Verification

- How are they different?
  - (Automated) mathematical reasoning
    - difficult, error prone
    - decidability vs. expressiveness
      - Propositional calculus is decidable
      - Predicate calculus is semi-decidable
  - Finite-state verification
    - Reason about a finite model of the system
    - Fast, yields counterexamples, manages partial specifications, applies to concurrency
    - State explosion!