Static Analysis

Prof. Leon J. Osterweil
CS 520/620
Fall 2012

- Requirements Spec.
  - Characteristics of System to be built must match required characteristics
  - Test Results must match required behavior

- Test Plan
  - Test plan exercises this code

- Design
  - Hi Level consistent views
  - Low level
  - Code must implement design

- Code
DEVELOPMENT PHASES

TEST PLANNING

TESTING PHASES
Summary of Dynamic Testing

- **Strengths:**
  - Microscopic examination of execution details
  - Evaluation in actual runtime environment
  - Oldest approach, most familiar

- **Weaknesses:**
  - Cannot demonstrate absence of faults
  - Hard to generate test data
  - Hard to know when testsets are adequate
  - Testing aids (eg. assertion checkers) alter execution

Some References for Testing

- Ghezzi, Jazayeri, Mandrioli, Fundamentals of Software Engineering, Chapter 6.3 (Testing)
  - Excellent annotated bibliography at the end of Ch. 6
- Many of those references are now rather old
  - Much recent work in this area
- Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)
- Proceedings of International Conference on Software Engineering (ICSE)
Summary of Problems in Doing Testing Effectively

- Hard to cover program execution space effectively
- Hard to select test data effectively
- Hard to tell if test results are right or wrong
  --if program computes complex function(s)
  --if the number of test cases gets large
- Best to detect errors early—before they become faults
- Testing comes at the end of the lifecycle, when time and budget tend to be short

What do you know when testing is done?
What relations have been checked?
  How thoroughly?
  To whose satisfaction?

Relations and Analysis

- A software product consists of
  -- A collection of (types of) artifacts
  -- Related to each other by myriad Relations
- The relations are essentially desiderata
  -- At least initially
- Before the product can be trusted, the relations need to be verified/confirmed
- That is the role of analysis
  -- Does the software do what it is supposed to do?
  -- What are its capabilities and its strengths?
  -- What is the nature of the artifact(s) that have been built?
  -- What can I count on?
  -- What should I worry about?
Static Analysis

- Technique for demonstrating the absence of faults without the need for execution
- Specification of Intent: derived from requirements
- Specification of Behavior: derived from model(s)
- Comparison: Done analytically and mathematically
- Results: Theorems about the program (eg. proofs that certain behaviors are impossible—or mandatory)
Testing and Analysis Framework

- Specification of Intended Behavior
- Development (Synthesis) Process
- Evaluation (Analysis) Process
- Comparison of Behavior to Intent (relation checking)
- Testing/Analysis Results

Dynamic Testing

- Specification of Intended Behavior
- Required Outputs
- Test Execution Results
- Result Comparator (Human or Machine)
- Failure Reporting

Comparison of Behavior to Intent
## Static Analysis

- **Specification of Intended Behavior**
- **(Rqts. (?)) Specification**
- **Derived From Source Text**
- **Mathematical Reasoning**
- **Comparison of Behavior to Intent**
- **Theorems Or Counterexamples**

## Natural Complementarity

- Static Analysis excels at scanning for trouble
- Dynamic testing excels at focusing in sharply
- Both aim to identify violations of well-formedness properties
- Techniques should be mutually supportive, synergistic
- More on this later
Dataflow Analysis

• Specification of Intent: Sequence of events
• Specification of Behavior: Derived from flowgraph model
  – Nodes annotated with events of interest
  – All possible executions modeled as all sequences of
    events along all flowgraph paths
• Comparison: Analytic process
  – Are all possible event sequences the desired one(s)?
• Result: Theorems demonstrating absence of event
  sequence errors
• Examples:
  – No variable referenced before definition
  – No file read before it is opened
  – Elevator doesn’t move until doors are shut
  – Rocket won’t try to fire thrusters after fuel is exhausted

Use of the Flow Graph (ImmFol Relation)

to Determine Actual Behavior

• Determine local information by looking at local behavior
  – Which artifacts are created, used,
• Infer global information
  – By looking at local information
  – And tracing flow graph edges to find relevant
    information at other nodes
Documenting Local Behavior

- First, determine local information that is true at each node in the CFG
  - e.g., What variables are defined
  - What variables are referenced
  - Usually stored in sets
    » e.g., ref(n) is the set of variables referenced at node n
- This same type of information available for pre-code artifacts
  - Interface specifications for modules
  - Input and output specifications of requirements elements

Standard Propagation Rules

- Need to propagate information from IMMFLOL neighbors
  - Backwards or forwards
- Must take branching into account
- Must add in “local” information
  - What is happening at this node
Inferring Global Behavior Example: Reaching Definitions

- Definition reaches a node if there is a def clear path from the definition to that node
- Do this by tracing paths between nodes
- Definition of x at node 1 reaches nodes 2, 3, 4, 5 but not 6

Originally used to determine data dependencies, for debugging, data flow testing, etc.

No need to use this only on models of code. Apply to rqts., design, etc.
Another Example: Live Variables

- a variable, \( x \), is live at node \( p \) if there exists a def-clear path for \( x \) from node \( p \) to a use of \( x \)
- \( x \) is live at 2, 3, 4, but not at node 5

This is best done by tracing backwards.
Different Global Behavior Problems Require Different Propagation Rules

- Some use forward tracing
- Some use backward tracing
- Some require ANDing
- Some require ORing
- Some compute ALL results
- Some compute SOME results
- Some compute NONE results
- And various combinations of the above

“Might” or “Must” reaching behavior

Definitions that **might** reach a node

\[
\text{reaching}\_\text{def} = \{x_i\} \quad \text{reaching}\_\text{def} = \{y_j\} \quad \text{reaching}\_\text{def} = \{x_i, y_j\} \quad \text{reaching}\_\text{def} = \{y_j\}
\]

Definitions that **must** reach a node

\[
\text{reaching}\_\text{def} = \{x_i, y_j\} \quad \text{reaching}\_\text{def} = \{y_j\} \quad \text{reaching}\_\text{def} = \{y_j\} \quad \text{reaching}\_\text{def} = \{y_j\}
\]
“Possible” or “Definite” live variables

Possible live variables

Possible live variables

Definite live variables

Finite State Verification: Using Data Flow Analysis to Verify Properties

- Data flow analysis used to infer behavior
- Requires use of propagation among nodes
  - Multiple propagations may be needed
- Infected behavior can then be compared to desired
  - Often comes from Requirements spec.
    » E.g. robustness specs.
- Works on any graph—not just graphs derived from code
  - E.g. DFGs in requirements or architectures
Examples of Properties

- Language specific
  - No undefined references
  - Can’t write to a file until it is opened
- Specific to a system
  - Can’t get money until pwd is OK
  - Can’t move elevator with doors open
  - Etc.

Some properties are much more complicated.
Longer sequences of events
**Approach**

- Define a desired (or undesired) property as a FSA
  - Based on identification of an alphabet of significant events
- Trace all paths through a graph that models the software
  - Where some nodes are annotated as the sites of events of interest
- Use the FSA as an acceptor
  - All paths from start to end through the model should reach an accepting state

**Using FSAs to Define: The “Safe Elevator” Property**

A Property, \( P \): The elevator does not move while its doors are open.

Events: \{open, close, move\}

\( L(P) \) is the set of all strings accepted by property \( P \)

All traces through the system should satisfy this property (drive FSA to accepting state)

Error: “trap” state
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{close})^*)^*, \text{open}^*)

Event Stream Comes from
Traces through an annotated CFG

- CFG \(G\) is \((N, n_{\text{initial}}, n_{\text{final}}, E)\)
- Use annotation relation to associate events with nodes
  - \(\Sigma_G\) is the alphabet of \(G\)
  - Events must be indivisible wrt other events in the property
- \(L(G)\) is the language of \(G\)
  - The set of all strings in \((\Sigma_G)^*\) that occur on paths from the initial node to the final node
Static Dataflow Analysis

Specification of Intended Behavior

Event Sequences

Possible Execution Sequences

Dataflow Propagation Algorithms

Comparison of Behavior to Intent

Proofs of the Presence or Absence of Faults

Architecture of a Verifier

Property

Event alphabet

Property Translator

System Translator

System

Annotated CFG

Reasoning Engine

Consistent

Inconsistent + counter example
Reasoning Engine: State Propagation

- States of the property are propagated through the CFG
  - For each node of the CFG, indicate the states of the property that the program could be in at that point in the program
  - Need to show $L(G) \subseteq L(P)$
- The property is proved if only accepting (non-accepting) states are contained in the final node of the CFG

Simple Example

call_for_elevator
1: if (stopped at call_floor) then
2: open;
end if;
...
3: if (stopped at non_call_floor) then
4: close;
end if;
5: move;
...
State Propagation

1: if

2: open

3: if

4: close

5: move

1: if

2: open

3: if

4: close

5: move

1: if (stopped) then
2: open;
end if;

3: if (stopped) then
4: close;
end if;

5: move;
**Fixed Example**

call_for_elevator
1:  if (stopped at call_floor) then
2:    open;
else
4:    close;
    move_to_call_floor
6:    open;
end if;
7:    close;
5:    move;
...

**State Propagation**

1:  if
   
2:  open
   
4:  close

6:  open

7:  close

5:  move

1:  close

2:  open

4:  close

6:  open

7:  close

5:  move
State Propagation: Strengths

- Relatively efficient approach for proving properties of a system
- Requires limited manual intervention
  - Techniques being developed to automatically determine constraints or system model
- Actually proves that the property is true for all possible executions
  or
  provides a counter example
State Propagation: Weaknesses

- Difficult to state the properties precisely
  - Program, property, or both could be wrong
- Must look at each counter example to determine if it corresponds to an executable path
- Can only reason about static models of the system
- Still a sampling technique
  - Must decide what properties to verify

Concurrent and Distributed Systems

- Extremely difficult to develop and test such systems
- Non-determinism means that
  - the same inputs might produce different outputs on different executions
  - When reasoning about a system there are numerous different alternatives to consider
    - Usually more than a programmer can reasonably consider
- In addition to the problems that can arise with sequential programs, have problems that are unique to concurrent systems
  - Data access problems
  - Synchronization problems
Conclusions

- Finite state verification approaches are improving
- Being used in industry for hardware systems
- With the increasing interest in software security and quality, becoming more widely used for software systems
- FLAVERS provides a demonstration of its potential effectiveness

Evaluation of Static Analysis

- Strengths:
  - Can demonstrate the absence of faults
  - Proofs can be automatically generated and proven
  - Algorithms are fast (low-order polynomial)
  - No need to generate test data
  - You know when you are done

- Weaknesses
  - Behavior specification is a model with inaccuracies
    » Not all paths are executable
  - Only certain classes of faults analyzable
    » Mostly sequence specific
    » Weak on functionality
Symbolic Execution

- Specification of Intent: Formulae, functions
- Specification of Behavior: Functions derived from annotated flowgraph, symbol table
  - Annotate nodes with function(s) computed there
  - Specify path to be studied
  - Compute function(s) computed as composition(s) of functions at path nodes, constraints of path edges
  - Comparison: Solving simultaneous constraints; symbolic algebra
- Results: Demonstrations that given paths computed the right function(s)
Example Symbolic Representation

1: input A,B

2: A>0?

3: C :=0

4: C := A*B

5: B>0?

6: X := C*(A+2*A)

7: X := A+B

8: output X

• P1: (1,2,4,5,6,8)
  • n: path values
  • condition
    1: A=a, B=b true
    2: a≤0
    4: C=a*b
    5: a≤0 ∧ b>0
    6: X = a*b*(a+2*a)
    8: out 3*a**2*b
• Path condition: a≤0 ∧ b>0
• Variable values
  A: a
  B: b
  C: a*b
  X: 3*a**2*b

Example Symbolic Representation

1: input A,B

2: A>0?

3: C :=0

4: C := A*B

5: B>0?

6: X := C*(A+2*A)

7: X := A+B

8: output X

• P2: (1,2,4,5,7,8)
  • n: path values
  • condition
    1: A=a, B=b true
    2: a≤0
    4: C=a*b
    5: a≤0 ∧ b≤0
    7: X = a+b
    8: out a+b
• Path condition: a≤0 ∧ b≤0
• Variable values
  A: a
  B: b
  C: a*b
  X: a+b
Example Symbolic Representation

1: input A, B
2: A>0?
3: C := 0
4: C := A*B
5: B>0?
6: X := C*(A+2*A)
7: X := A+B
8: output X

- P1: (1,2,3,5,7,8)
  - n: path values
  - path condition: 1: A=a, B=b
    - true
    - 2: a>0
    - 3: C= 0
    - 5: a>0 ∧ b≤0
    - 7: X = a+b
    - 8: out a+b
  - Path condition: a>0 ∧ b≤0
  - Variable values:
    - A: a
    - B: b
    - C: 0
    - X: a+b

Applications of Symbolic Evaluation

- Symbolic Testing
  - examination of path domain and computation for detecting failures
  - especially useful for scientific applications
- Path and Test Data Selection
  - select paths to cover structure and determine feasibility of condition
  - select data to satisfy path condition or "revealing" condition
- Debugging
  - examine symbolic representation for faulty data manipulation
- Verification
  - prove consistency of specification assertions
  - inductive assertion method for proving correctness ...
  - \( \{I\} S \{Q\} \) ...