Static Analysis

Prof. Leon J. Osterweil
CS 520/620
Spring 2014

End of the Semester Schedule

• April 22: Assignment #5 Distributed
• April 24: 620 Term Paper abstracts due
• April 29: Take-Home Final Exam Distributed
• May 2: Assignment #5 is due
• May 8: Take-Home Final is due
• May 8: 620 Term Papers due
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 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
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)

ANALYSIS AND TESTING PHASES
"Test First" development is really more like "analyze first"
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 → Theorems Or Counterexamples

Comparison of Behavior to Intent
Move from Dynamic Analysis to Static Analysis

- Dynamic analysis approaches are based on sampling the input space
  - Infer behavior or properties of a system from executing a sample of test cases
  - Sharp, narrow focus on one execution
  - Runs in real execution environment
  - Black Box versus White Box approaches

- Static analysis approaches tend to be based on a “global” assessment of the behavior
  - Based on an understanding of the program (artifact)
  - Broad focus on all paths
  - Can prove the absence of faults
  - Based on model of execution environment

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

Important use of Graphs we have seen

- Control Flow graph supports specification of actual behavior
- FSA supports specification of intent
- (Automated) mathematical reasoning supports comparison
Flow Graph (ImmFol Relation) to Determine Actual Behavior

- Determine local information by looking at local behavior
  - Which artifacts are created, used,
  - These node annotations are relations like those we have seen before
- Infer global information
  - By looking at local information
  - And tracing flow graph edges to find relevant information at other nodes
- Uses the ImmFol relation
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 IMMFL 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, 5 but not at node 6

Applicable to models of pre-code artifacts too
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

Definitions that must reach a node
"Possible" or "Definite" 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
- Inferred 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:

Open file, then
Write to the file, then
(read, write)*, then
Close file
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:
  {Open file, then
   Write to the file, then
   (read, write)*, then
   Close file}*

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\}

\( \mathcal{L}(P) \) 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

Simple Example: Actual Behavior

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;
...

Intended Behavior

Determining consistency between functionality and robustness requirement

Worklist: 2, 3, 4, 5

1: if
   {1}
   {1}

2: open
   {1}
   {2}

3: if
   {1,2}
   {1,2}

4: close
   {1,2}
   {1}

5: move
   {1,2}
   {1,3}
Determining consistency between functionality and robustness requirement

Worklist: 2, 3, 4, 5

Violation Path

1: if
2: open
3: if
4: close
5: move

1: if
2: open
3: if
4: close
5: move
Appears to be Unexecutable: A “False Positive”

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;
...

One Way to Fix the Example

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

1: if
2: open
4: close
6: open
7: close
5: move
Now Property Holds

1: if
   \{1\}
   \{1\}

2: open
   \{1\}
   \{2\}

4: close
   \{1\}
   \{1\}

6: open
   \{1\}
   \{2\}

7: close
   \{2\}
   \{1\}

5: move
   \{1\}
   \{1\}

1: if
   \{1\}
   \{1\}

2: open
   \{1\}
   \{2\}

4: close
   \{1\}
   \{1\}

6: open
   \{1\}
   \{2\}

7: close
   \{2\}
   \{1\}

5: move
   \{1\}
   \{1\}
But is it really necessary to modify the program?

When maybe the problem is with the precision of the analysis

The Previous Example

Worklist: 1, 2, 3, 4

1: if
   {1}

2: open
   {1}
   {2}

3: if
   {1,2}
   {1,2}

4: close
   {1,2}
   {1}

5: move
   {1,2}
   {1,3}
Produced this Result

With this Spurious Report of a Fault

Spurious because the fault is on a path that is not executable

...
How to address these kinds of problems attributable to lack of precision?

Use more Graphs to Capture Constraints

• Constraints are represented as FSAs
• Describe conditions necessary for feasible execution
• Have a special *violation state* that is entered when an infeasible path is detected
  – Violation is a trap state; once it is entered, never leave that state
Boolean Variable Constraint

== is a predicate
= is assignment

S==t
S=t
S==f
S=f

S==t
S=t
S==f
S=f

== is a predicate
= is assignment
How do constraints affect the data flow equations

- IN and OUT are now sets of tuples of FSA states
- Merge is still union
- Transfer function now has to look at each FSA state in the in-tuple when computing the out-tuple
  - Property states not propagated when any constraint automaton is in the violation state
- Result looks at paths that are feasible with respect to the constraints
  - The property state is the same as before
  - Every constraint must be in an accepting state

Elevator Revisited

```
1: if
2: S==t
4: S==f
3: open;
end if;
5: if
6: S==t
8: S==f
7: close;
end if;
9: move;
...```

1,2,4: if (stopped) then
3: open;
end if;
...
5,6,8: if (stopped) then
7: close;
end if;
...
9: move;
...
State Propagation

Worklist: [2, 4, 6, 8]

1: if <1, u> 
<1, u> 
<1, t> 
<1, t> 
<2, t> 
<2, t>, <1, f> 
<2, t>, <1, v> 
<2, t>, <1, f> 
<2, t>, <1, v> 
<2, t>, <1, f> 
<2, t>, <1, v> 

2: S==t 
4: S==f 
<1, u> 
<1, f> 

3: open 

5: if <2, t>, <1, f> 
<2, t>, <1, f> 

6: S==t 
8: S==f 

7: close 

9: move
State Propagation

Worklist: 1, 2, 3, 4, 5, 6, 7, 8, 9

1: if <1,u> <1,u>

2: S==t <1,u>
   <1,t> <2,t>
   3: open

4: S==f <1,f>
   <1,t> <2,t>
   5: if <2,t>,<1,f>
       <2,t>,<1,t>
       6: S==t <2,t>,<1,f>
           <2,t>,<1,t>
           7: close <2,t>,<1,f>
               <2,t>,<1,t>
               8: S==f <1,t>,<1,f>
                   <1,t>,<1,f>
                   9: move <1,t>,<1,f>

Worklist: 1, 2, 3, 4, 5, 6, 7, 8, 9

1: if <1,u> <1,u>

2: S==t <1,u>
   <1,t> <2,t>
   3: open

4: S==f <1,f>
   <1,t> <2,t>
   5: if <2,t>,<1,f>
       <2,t>,<1,t>
       6: S==t <2,t>,<1,f>
           <2,t>,<1,t>
           7: close <2,t>,<1,f>
               <2,t>,<1,t>
               8: S==f <1,t>,<1,f>
                   <1,t>,<1,f>
                   9: move <1,t>,<1,f>
State Propagation

Worklist: 2, 3, 4, 5, 6, 7, 8

1: if
   <1,u>
   <1,t>

2: S==t
   <1,u>
   <1,t>

3: open
   <1,t>
   <2,t>

4: S==f
   <1,u>
   <1,f>

5: if
   <2,t>, <1,f>
   <2,t>, <1,f>

6: S==t
   <2,t>, <1,t>
   <2,t>, <1,f>

7: close
   <2,t>, <1,t>
   <1,t>

8: S==f
   <2,t>, <1,t>
   <2,t>, <1,f>

9: move
   <1,t>, <1,f>
   <1,t>, <1,f>

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
Power and Value of Static Analysis is Best Seen When Applied to Concurrent Systems

- Concurrent software is much harder, more complicated
- Requires very powerful analysis
- Finite state verification shines in this domain

Data Access Anomalies

- Typically you want mutual exclusion
  - shared resource (e.g., data) that should only have a single access at a time
  - e.g., don’t want two travel agents assigning the last seat on a plane
- Typically you don’t want race conditions
  - order of execution affects results
  - undesirable nondeterminism

```plaintext
if initially x = 5, then could output (6, 5), (4, 5), or (5, 5)
```
Interleaved Model of Execution; Examples

```
<table>
<thead>
<tr>
<th>task A</th>
<th>task B</th>
</tr>
</thead>
<tbody>
<tr>
<td>x := x + 1; write x;</td>
<td>x := x - 1; write x;</td>
</tr>
<tr>
<td>task B</td>
<td>task A</td>
</tr>
<tr>
<td>x := x - 1; write x;</td>
<td>x := x + 1; write x;</td>
</tr>
</tbody>
</table>
```

Infinite wait anomalies (continued)

- **Deadlock**
  - set of tasks mutually waiting on each other; none can advance
  - Often tasks hold resources that other tasks need
  - Also called, deadly embrace

- **Livelock**
  - execution does not come to a standstill but none of the tasks can advance
Reachability-based Model Checking: some history

- Originally proposed for hardware
- Early 80’s: E. Clarke and Emerson; Quielle and Sifakis
- Late 80’s: Improved algorithms and property notations (E. Clarke, Emerson, Sistla)
- 90’s: Symbolic Model Checking (SMV) and other optimizations (Burch, E. Clarke, Dill, Long, and McMillan)
- Current: Hybrid approaches that combine model checking with
  - Theorem proving techniques
  - Symbolic execution
  - Optimization techniques (e.g., points to analysis)

Reachability Graph

- Models state space
  - Each node represents a possible state in a distributed system
    - States represent the value of all the variables
    - For concurrent systems, this includes the program counter for each task
  - Considering only the value of the program counter for each task then each state is a vector where the ith element is the current program counter of the ith task
    - \(<pc_{1,i}; pc_{2,i}; ...pc_{r,i}>\)
    - Coarse-grained representation of a RG; doesn’t t consider values of variables
  - More powerful analysis requires more complete and detailed state vector
Reachability Graph

- Typically, each edge represents progress in a single task
  - Multiple concurrent events may be possible, but allowing only single events captures all states and simplifies the graph structure (interleaved execution model)
- Only have multiple tasks progress when required by the semantics of the programming construct
  - E.g., rendezvous
- Only contains states that are potentially reachable from the start state

Reachability Graph Example

Task control flow graphs

T1
- begin
- T2.Q
- end

T2
- begin
- Accept Q
- end

Reachability graph

- b1, b2
- q, b2
- b1, q'
- q, q'
- q, e2
- e1, q'
- e1, q
Reachability Graph Example (clarified)

task control flow graphs

T1

begin
T2.Q
end

T2

begin
Accept Q
end

Reachability graph

\[ \text{b}_q \text{ means that a task is blocked at } q \]

\[ r(q, q') \text{ means that the rendezvous between } q \text{ and } q' \text{ occurs} \]

Use a worklist to build the reachability graph

Worklist: \( <b_1,b_2><b_q,b_2><b_1,b_q><r(q,q')><e_1,q'><q,e_2><e_1,e_2> \)
Useful in determining possible/impossible synchronizations

Two possible synchronizations

But this is "precise up to symbolic execution"
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

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)