Testing

Prof. Leon Osterweil
CS 520/620
Spring 2014

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)

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?

Some Examples of “Relations”

- Executing this code must meet this requirement
- This code must conform to that design element
- This compiled code came from this compiler
- This design element addresses those requirements
- These lower level requirements are elaborations of these higher level requirements
- This is the date by which that test must be passed
- Component invocations conform to component abstract interface specifications
- Documentation describes the actual system
- ETC.....

We now examine the ways in which consistency is defined and determined through the use of relation specifications
Basic Notions and Definitions

- Consistency determination is fundamental
  - Specific relations start out as statements of intent
  - Product "has" these qualities if its behavior is consistent with (satisfies) statements of intent
- Basic Definitions:
  - failure: inconsistency between actual behavior of software and specification of intent
  - fault: software flaw whose execution caused the failure
  - error: human action that results in software containing a fault

More Definitions

- Testing: The systematic (?) search of a program’s execution space for the occurrence of a failure

More Definitions

- Testing: The systematic (?) search of a program’s execution space for the occurrence of a failure
- Debugging: Searching for the fault that caused an observed failure
- Analysis: The static examination of a program’s textual representation for the purpose of inferring characteristics

What are V and V?

Informal Requirements

Formal Specifications

Software Implementation

Validation

Verification

Verification: Is this software right?
Validation: Is this the right software?
More Definitions

- Testing: The systematic (?) search of a program’s execution space for the occurrence of a failure
- Debugging: Searching for the fault that caused an observed failure
- Analysis: The static examination of a program’s textual representation for the purpose of inferring characteristics
- Verification: Using analytic inferences to formally prove all executions of a program are consistent with specifications

The Essence of Analysis

<table>
<thead>
<tr>
<th>INTENT</th>
<th>COMPARISON of BEHAVIOR to INTENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Originates with requirements</td>
<td></td>
</tr>
<tr>
<td>- Different types of intent (requirements)</td>
<td></td>
</tr>
</tbody>
</table>

| BEHAVIOR |
| - Can be observed as software executes |
| - Can be inferred from execution model |
| - Different models support different inferences |

| COMPARISON |
| - Can be informal—done by human eyeballs |
| - Can be done by computers (comparing text strings) |
| - Can be done by formal machines (eg. FSM's) |
| - Can be done by rigorous mathematical reasoning |

*Results obtained will vary as a function of the above*

(Dynamic) Testing

- Behavior determined by examining test execution results
- Intent derived (somehow) from (various) specifications
- Comparison typically has been done by text examination
  - Although much more "automatic" testing done now
- Testing is aimed at discovering the presence of faults
  - By discovering failures
  - And using debugging to trace them to faults
- Testing should select test cases likely to reveal failures

The Framework

- Specification of Intended Behavior
- 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
Informal Dynamic Testing

Black Box vs. White Box Testing

Structural (White Box) Testing

Input Space Partitioning

Rigorously defined Flowgraph helps

<table>
<thead>
<tr>
<th>Specification of Intended Behavior</th>
<th>Test Cases</th>
<th>Execution Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Evaluation Process</td>
<td>Failures Observed</td>
<td>Comparison of Behavior to Intent</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input Space</th>
<th>Program</th>
<th>Output Space</th>
</tr>
</thead>
</table>

Black Box vs. White Box Testing

SELECTED INPUTS
RESULTANT OUTPUTS
DESIRED OUTPUT
SOFTWARE DESIGN

“BLACK BOX” TESTING

“WHITE BOX” TESTING
(CLEAR BOX TESTING)

Structural (White Box) Testing

• Testcase choices driven by program structure
• Flowgraph is most commonly used structure:
  – Represent statements by nodes
  – If a statement can execute immediately after another, connect the nodes representing them by an edge
  – Every program execution sequence is a path
• Criteria based on "coverage" of program constructs
  – All statements (node coverage)
  – All control transitions (edge coverage)
  – All possible paths, loop iterations (path, loop coverage)
• How to generate input data to do this?
  – What exact data sets are used to force these coverages?
  – It matters

Input Space Partitioning

Input Space Divided into Domains

Different paths partition the execution space

Rigorously defined Flowgraph helps

```python
totalpay := 0.0;
for i = 1 to last_employee
    if salary[i] < 50000.
        salary[i] := salary[i] * 1.05;
    else
        salary[i] := salary[i] * 1.10;
    end loop;
end loop;
print totalpay;
```
Using Flowgraphs to Partition Input Space

- Requires use of static analysis (described soon)
- Use program branching conditions to create sets of constraints
- Solving constraints for each path yields the input domain that forces execution down that path
- Called Symbolic Execution
- More on this soon

How to assure testing is thorough?

- Assure every node is covered by at least one test case (node coverage/statement coverage)
- Assure every edge is covered by at least one test case (edge coverage/branch coverage)
- Assure every loop is exercised
  - Both by iteration and fall-through
- Assure all execution paths are covered
  - Practical impossibility

Best to remember that:

Testing is buying knowledge

Automated Dynamic Testing

Specification of Intended Behavior

Rigorous Specs. (Asserts)

Evaluation Process

Execution Results

Automated Comparator

Comparison of Behavior to Intent

How to assure testing is thorough?

- Specification of Intent:
  - Expressed explicitly
  - Increasingly completely
    - Functionality, timing, accuracy, robustness, ...
  - Increasingly rigorously
    - Mathematics, FSA’s
    - Ideally arise directly from requirements and design specifications
- Specification of Behavior
  - Tools to capture test outputs (inputs too)
- Comparison
  - With automatic comparators

Functional (Black Box) Testing

And there are many other criteria
Examples of Black Box Testing Goals

• Does the software do what it is supposed to do?
• When might it fail?
• How fast does it run?
• How accurate are the results?
• What are its failure modes and characteristics?
• What can I count on?
• What should I worry about?
• What are its strengths and weaknesses?

Assertion-Based Testing

• Imbed assertions with program executable code
• Assertions function as runtime checks
• Supports zooming in on internal workings of the program
• Examine behaviors at internal program locations while the program is executing
  – Augments examining only final outputs
• Comparison: Runtime evaluation of assertions
  – Facilities for programming reactions to violations
• Also useful as a debugging aid

```
X := Y;
Time := Y * 2.0 * T;
ASSERT Time > 0.0;
```

Inserted by Tester

Automatically processed into

```
if ~(Time > 0.0) Then
  Assertion_violation_handler;
```

Functional vs. Structural Testing

• Cannot determine if software does what it is supposed to do without considering the intent
  – a special case not handled by the implementation will not be tested unless the specification/requirements are considered
• Cannot ignore what is actually done by the software
  – program may treat one element in domain differently than stated in the specification
  – implementation often employs use an algorithmic technique with special characteristics that are not highlighted in the specification
• Both functional and structural testing must be done

Should use all available information in testing

How much testing is enough?

• Test programs by running data through them
  – Does the program select “correct answers”?
  – Reject incorrect ones
How much testing is enough?

- Test programs by running data through them
  - Does the program select "correct answers"?
  - Reject incorrect ones
- Test test datasets by running programs through them
  - Can the testsets separate the correct programs from the MUTANTS?

Mutation Testing

- Determines the adequacy of sets of testcases
- Theory
  - Testsets should be sufficient to pick up all faults
  - In order for a testset to be adequate one of its testcases must differentiate between correct code and code with a fault

Mutation Testing Approach

- Produce a family of "mutants" of the original program
- Use it to test the adequacy of the program’s testcase set:
  - Run mutants and original program over the set
  - Make sure some testcase produces different results
  - If not, make sure mutants didn’t really change the program
  - If it did, then add a new testcase that forces different results

How to build mutants

- The "competent programmer" assumption
- Mutants are simple coding errors
  - Change each integer to another
  - Change each loop iteration count (by 1)
  - Misspell each variable name
  - Change each arithmetic operator
  - Etc.
- More complicated mutants seem unnecessary
- Multiple mutants seem unnecessary
Create a mutant

Example:
Change
x := a + b
To
x := a - b

Continue to create more mutants

So, how does this work?

Start with all mutants

Consider a set of testcases
Run each testcase through all mutants

Compare mutant output to original output

If results differ then "kill" the mutant

Continue for all testcases

A good set of testcases kills all mutants

Note, however, that, for example: a test set where \( b = 0 \) will not kill the mutant created by replacing
\[
\begin{align*}
x &:= a + b \\
\text{with} \\
x &:= a - b
\end{align*}
\]
Mutation testing

- A fun idea
- Lots of interest in this over the decades
- Still not a particularly practical idea
- So what do we do instead?

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?

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 test sets are adequate
  - Testing aids (e.g., assertion checkers) alter execution

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 (e.g., proofs that certain behaviors are impossible—or mandatory)