Testing

Prof. Leon Osterweil
CS 520/620
Spring 2013

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.
Relation verification should take place continuously through development

- Analysis should occur continuously from the start:
  - requirements validated against user needs
  - requirements shown internally consistent
  - validate current artifacts against user needs
  - use information from previous analyses to verify current artifacts

- Test plans should begin with requirements and be reviewed and refined with each phase
  - test plans should be executed as early as possible

More details coming up soon
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

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
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
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
- **Validation**: Using any means possible to confirm that the software is solving the stakeholders’ problem(s)

The Essence of Analysis

**COMPARISON of BEHAVIOR to INTENT**

**INTENT**
- Originates with requirements
- Different types of intent (requirements)

**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
Dynamic Testing

Specification of Intended Behavior

Required Outputs

Test Execution Results

Result Comparator (Human or Machine)

Comparison of Behavior to Intent

Failure Reporting

Examples of Goals for Testing

- 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?

Should be driven by requirements and Software product well-formedness specifications
Testing is too long and hard to do all at once at the end of development

• Divide the job into subtasks
• Do some activities during development
  – Can do test planning during development
  – And *should* do so
  – May reduce or eliminate the need for some post-coding testing
• Phase testing at the end
  – Using test plans previously developed

Testing Phases

• Unit/Module
  – Comparing a code unit or module with design specifications.
  – planned during coding: done after coding
• Integration
  – Systematic combination of software components and modules
  – planned during design: done after unit/module V&V
• Software System
  – Comparing entire software system with requirements
  – planned during requirements: done after integration
• System
  – Comparing integrated hardware/software system to requirements
  – planned during informal requirements: after SW System
DEVELOPMENT PHASES

- Requirements Specification
- Architecting
- Implementation Designing
- Coding

TEST PLANNING

- System Test Plan
- Software Sys. Test Plan
- Integration Test Plan
- Unit Test Plan

TESTING PHASES

- System Testing
- Software Sys Testing
- Integration Testing
- Unit Testing

Flow of control edge (the ImmFol relation)

Data Flow edge (artifacts flow from tail to head)

A better representation
A better representation

“Testing” focuses on how to do these

Where Requirements Looks Like This
Where Requirements Looks Like This

And “testing” is Done here

Analogous View of Design

1 Declare HLDesign Element
2 +
3 Define HLDesign Elements
4 Declare HLDesign Elements
5 
6 Declare and Define HLDesign Elements
7 ~ HLD OK
8 Requirements
9 ~ Rqmts OK

High-Level Design

Declare HLDesign Elements

Develop Rqmt Element

A Rqmt OK

HLDesign OK

design riqms OK

Add new requirements element

design riqms conformance check

Designer does as many as he/she thinks are needed

Missing riqms

requirements inconsistent

Declar HLDesign Elements

Declare and Define HLDesign Elements

Declare Design Elements

Develop Design Elements

High-Level Design
Black Box vs. White Box Testing

**“BLACK BOX” TESTING**

SELECTED INPUTS → RESULTANT OUTPUTS → DESIRED OUTPUT

**“WHITE BOX” TESTING**

(CLEAR BOX TESTING)

SELECTED INPUTS → RESULTANT OUTPUTS → DESIRED OUTPUT

INTERNAL BEHAVIOR → SOFTWARE DESIGN
Testing is Sampling the Input Space

- Key problem: What is the input space?
  - What is the software intended to do?
- Subproblem: The input space is large
  - One dimension for each program input
  - Each dimension can have as many elements as there are legal inputs (e.g., 2**32 different integers)
  - Each input really is different
  - How different? Which differences matter?
- Key Problem: How to sample from this input space?
What is the input space?

**Specification**

sum_of_roots takes an arbitrarily long sequence of real numbers, and computes the sum of their square roots. The real number sequence must be ended with the number 9999.99

**Implementation**

Program sum_of_roots;
Real sum, x, r;
sum := 0;
Do forever
input x;
if x = 9999.99 then exit
else
r := sqrt(x);
sum := sum + r;
end do;
print sum;
end;

---

Computing the Input Space

- There are ~2**32 possible different values for each input
- If n values are read in, then there are (2**32)**n different points in the input space
- The number of different input values read in is unlimited
- There is no limit (theoretically) to the size of the input space
Some observations about the example program input space

- There is no real need to test every possible combination of input values
  - Most executions behave “the same”
- But some input combinations are “different”
  - Negative values will produce a failure
- There is a virtually limitless number of inputs that don’t cause the negative square root failure
- A sufficiently large sequence of input values will cause an overflow failure

Effective selection of test cases requires thought and care

The Testcase Selection Problem

- Testing lets you put your program under a microscope
  - Can examine minutiae
  - But only for current execution
- To find faults you need to select test data to cause failures
- Testing can demonstrate the presence of faults (when suitable test cases are selected)
- But demonstrating the absence of faults requires knowing the behaviors of all executions
  - But there are (virtually) infinitely many possible executions
  - So how to sample the inputs representatively?
Partitioning the Input Space

- **Rationale:** All points in the same subdomain are processed “equivalently” by the program
- **But:**
  - How to determine the partition?
  - How to know how far the equivalence holds?
  - How to select the point(s) within each domain to use as the actual input(s)?
- The manual and/or requirements specification can help

Equivalence Partitioning

- **The typical approach**
- A test of any value in a given class is equivalent to a test of any other value in that class
- If a test case in a class reveals a failure, then any other test case in that class should reveal the failure
- Some approaches limit conclusions to some chosen class of faults and/or failures
Testing

Input Space

Program

Output Space

Input Space Partitioning

Input Space Divided into Domains

Program
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

Rigorously defined Flowgraph helps

totalpay := 0.0;
for i = 1 to last_employee
  if salary[i] < 50000.
    then salary[i] := salary[i] * 1.05;
    else salary[i] := salary[i] * 1.10;
  totalpay := totalpay + salary[i];
end loop;
print totalpay;

Different paths partition the execution space
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

And there are many other criteria
Functional (Black Box) Testing

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

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?
Functional (Black Box) Testing Guidelines

- Result in many test cases
- Some test cases may satisfy many heuristics
- Keep track of the goal of each test case
- Changes in specification will cause changes in functional test cases
- Need a way to organize, use, reuse, and monitor functional testing
- NOTE: many of the functional testing guidelines can also be applied in structural testing (at a more detailed and formal level)

Assertion-Based Testing

- Zoom in on internal workings of the program
- Examine behaviors at internal program locations while the program is executing
  - Augments examining only final outputs
- Assertions: Specifications of intended properties of execution state (e.g. relations among the values of program variables)
  - Development of increasingly elaborate assertion languages
  - Often: Checking relations between code and design
- Comparison: Runtime evaluation of assertions
  - Facilities for programming reactions to violations
- Also useful as a debugging aid
<code sequence>
X := Y;
Time := Y * 2.0 * T;
**ASSERT Time > 0.0:**
<rest of code>
</code>

if ~(Time > 0.0) Then
  Assertion_violation_handler;

---

**Assertion-Based Dynamic Testing**

- Specification of Intended Behavior
- **Functional Behavior Assertions**
- **Intermediate Execution Results**
- **Runtime Assertion Checking**
- Comparison of Behavior to Intent
- Reports on Internal Failures

- Specification of Actual Behavior
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

How much testing is enough?

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

• 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
Start with program to be tested

Create mutants
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

Example:

Change

\[ x := a + b \]

To

\[ x := a - b \]

Create a mutant
Create a mutant

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
If results differ then “kill” the mutant

Compare mutant output to original output

If results differ then “kill” the mutant
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 $x := a + b$ with $x := a - b$.

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 testsets are adequate
  – Testing aids (eg. 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 (eg. proofs that certain behaviors are impossible—or mandatory)