SOFTWARE TESTING

Goal of testing is to show that the software meets its specifications.

Generally two approaches:

A priori - Show that that software meets its goals before producing the source code - generally verification (i.e., formal proof) methods

A posteriori - Show that the software meets its goals after producing the source code - generally testing

Some attributes are more amenable for verification and some are more amenable for validation.

Software Testing - II

Alternative characterization:

Validation - Show that a software system meets actual requirements (e.g.,

usability testing, performance analysis)

Verification - Show that a software system meets its formal description (e.g., functional correctness, security properties)

Problem with testing - Can only "show the presence of bugs, but never their absence" - Dijkstra (1972)

Testing attributes

Correctness: Show consistency between two Formalisms (e.g., specifications design, design

to source code, requirements to specifications, the "what" to the "how").

Reliability: A quantifiable relationship between failures and system behavior. Defined from perspective of user profiles.

Robustness: Provides acceptable performance in the face of correctness or reliability failures (e.g., graceful degradation). Concerned with behavior in face of unusual conditions.

Safety: Prevention of hazardous conditions from arising. Related to both reliability and robustness.

Properties related to correctness issues (e.g., prove that a system cannot get into a certain state) Security attribute similar to safety issues.

TESTING (Verification and validation)

It is a process that extends throughout the development life cycle.

- Need to plan for testing early in life cycle
- Need to perform testing throughout life cycle
- Need to feedback results of testing throughout life cycle

Some attributes (e.g., reliability) only available late in life cycle, so need to substitute other related attributes (e.g., fault density) earlier in life cycle.

Needs (reliability, correctness, robustness) differ depending upon application domain and sometimes these attributes are in conflict or conflict with other attributes (e.g., time to market, cost)

Testing Process

(Based on M. Young and M. Pezze model)
Requirements:
- Identify attributes of interest
- Acceptance test planning
- Elicit requirements from users
- Identify risks

Specifications:
- Validate specifications
- System test planning
- Create functional tests

Architectural design:
- Architectural design analysis
- Design inspection
- Integration and unit test planning
- Develop maintenance plan
- `Develop operational profile`

Testing Process - II

Detailed design:
- Design inspections
- Verify designs
- Generate oracles
- Generate black-box test cases
- Unit test planning

Coding:
- Code inspections
- Create scaffolding
- Unit test execution
- (Alternative: cleanroom process, walkthroughs [not as effective])
- Automated code analysis
- Coverage analysis

Testing Process - III

Integration:
- Integration test
- System test execution
- Acceptance test execution
- Performance analysis
- Deliver regression test suite

Maintenance:
- Implement problem report process
- Regression test execution
- Revise regression test suite
- Develop aging model - when to fix versus throw away

What is needed in all of above:
- Visibility of processes
- Feedback of results to affect future behavior

Module testing framework

Use of scaffolding- test drivers and stubs:Also called test harness:
- Driver (initialize data variables)
- Call UUT (Unit under test)
- Stub -- module called by UUT

Specification-based testing: "Black box" testing - the structure of the program is not known to the tester

Examples: Algebraic data types

POP(PUSH(S,I))=S leads to a test case. Finite state diagrams. Grammars

Structural testing: "White box" testing - the structure of the program is known to the tester

Tests are determined by the structure of the program. Develop processes for determining whether all functionality of a module has been tested

Mutation-based testing: For each expression, derive alternative expressions. Develop tests to see if data can distinguish between original program and a mutation

State table solution to above requirements: When to switch RTs.

And—Or table. Read conditions down ("and" them). Each column defines a separate ("or) sequence of actions. (● – don't cares)

C&C MDM acting as bus controller	T	T	T	T
Detection of transaction errors in 2 consecutive processing frames T	T	T	T	
Errors are on selected messages	T	T	T	T
The RTs 1553 FDIR is not inhibited	T	T	T	T
A backup BC is available	T	T	T	T
The BC has been switched in the last 20 sec	T	T	T	T
SPD card reset capability is inhibited	T	T	●	●
SPD card has been reset in the last 10 major (10 sec) frames	●	●	T	T
Transaction errors are from multiple RTs	T	T	T	T
The current channel has been reset within the last major frame	T	F	T	F
Bus channel's reset capability is inhibited	●	T	●	T

Specification-based testing

- Use each column to define a set of constraints on the program in order to develop a test set.
- But need to limit the number of test cases.
- Above table can generate 4+2+4+2 = 12 test cases.

Structural-based testing

Remember definitions from earlier in semester:

- Error – a concept that is incorrect.
- Fault – the place that contains the error.
- Failure – the place in the artifact where the error becomes known.

Goal of testing:

- Determine whether a relatively small part of a program apparently behaves. Two reasons:
- Find faults (debugging)
- Provide confidence that there are no errors (acceptance testing)

Testing criteria

Goal of testing research, since mid-1972, is to define a test set and to define test set adequacy so that each test case from the test set represents one test from a class of possible tests.

Testing is then:

- Run one test case from each test set class
- If there is an error in the program for that test set class, then the test will exhibit that error

If you can define all possible classes of tests, you can show that the program must be correct.

- However, we run into undecidable issues in trying to define test set adequacy criteria.

Testing criteria

Statement coverage - Every statement in the program is executed. A measure of test set adequacy could be the percentage of statements executed by the test set.

Branch coverage - Every branch in the program is executed. A measure of test set adequacy is the percentage of control transfers executed.

Condition coverage - Every predicate must be both true and false for the test set.

Path coverage - A path is a sequence of statements from the entry point to the exit point of the program. Path coverage requires that all possible paths (with repetitions) are executed. The percentage of possible paths is the adequacy measure. (How to handle DO loops?: 0, 1, multiple passes through loop)

Mutation adequacy - Seed a program with faults with simple changes. Mutation adequacy is the percentage of mutated faults discovered by the test set versus the number of seeded faults.

Def-Use coverage

DEF(X) - Variable X is assigned a value (X := expr)

USE(X) - Variable X is used (Y := …+X+…)

- Dataflow coverage - Every path from DEF(X) to USE(X) for all variables is executed.

Test adequacy coverage

Adequacy requirements:

- Reliability- A program testing successfully on one test case will test successfully on all test cases of a test criterion.

- Validity - Every error in a program is found by at least one test criterion.

But both conditions together are undecidable.

Test adequacy - formally

Test set adequacy stopping criterion C:

- **p = program under test**
- **s = specifications for program**
- **t = set of tests ($t \subset 2^D$ where D is set of inputs)**
- **C(p,s,t) = {true, false} depending upon whether t is an adequate test set under criterion C.**

Test set measurement criteria:

- **C(p,s,t) = r, $r \in [0,1]$**
- **Larger r is more adequate criterion**

- **Partition testing one method to try and achieve this.**

Partition testing

Basic idea for a test set:

- **Divide program input data space into equivalence classes**
- **Choose data from each equivalence class**

Category partition method:

- **For each function identify parameters that can be tested**
- **Partition each parameter into choices**
- **Determine any constraints on these choices**
- **Write tests**

Examples of partition categories

Boundary testing:

- →0, negative, positive
- →Arraysize-1, Arraysize, Arraysize+1

Erroneous conditions:

- →Look at predicates in program as indicators of partitions
- →Choose data that represent alternative values of predicate

Problem: How to know what input data gives these internal values.

Problems in structural testing

When is testing completed?

- When money is used up?
- When project is due?
- Can we define rational testing criteria?

Feasibility?

- Can all necessary paths be generated by tests?
- Adequacy scores based on coverage
- NASA/SEL - FORTRAN coverage -- good code with scores of 70%

Problems with other testing methods

Interprocedural call and dataflow testing:

How to manage complexity?

How to access internal structure?

Regression testing:

run a component with all previous test sets

A heuristic approach to confidence

How large a test set to use?

Late and dynamic bindings:

Data items and structure determined at run-time

Similar to interprocedural, only bigger problem

Test oracles

An oracle:

- An inspector of executions
- A correct copy of program - Feed in data and get correct answer out
- Do not usually have this, so do a good approximation

M out of N programming:

- Develop multiple copies of program
- Correct answer is majority vote
- Problems:
  - →Similar errors, so use different teams
  - →Expensive

Oracles from specifications

Part of requirements development.

Develop testing results at time of system design

Make specifications testable:

→State diagrams allow for easily testing specifications

→Bad: Response time is acceptable

→Good: Response time is 10 seconds 90% of time

- Use of assertions:
  - Assert(InputRecordNumber<FileSize);

Remember PBR testing model

INTEGRATION AND SYSTEM TESTING

Integration testing strategies:

- Big bang -- Put everything together at once and see what happens
- Top down -- use stubs for uncompleted modules
- Bottom up -- use drivers to call modules
- Threads -- integrate complete paths from top to bottom
- Critical Modules -- Do most important components first

System testing

- Performed after components are integrated
- Software is compared with its specifications
- Usually performed by the developer
- Sometime IV&V used on critical systems

Independent validation and verification (IV&V):

- System testing performed by developer group independent of development team
- Used often on critical systems
- Expensive since testers do not know system
- But because of that, independent team does not have the mindset of the developer group and less apt to make similar logic mistakes
- Some studies show IV&V costs 40% more
- But, can you afford the failure?

Experience of NASA/SEL on IV&V

- Code NOT more reliable
- Cost +30%
- Use depends upon application domain

Acceptance testing:

- Software is compared with end-user requirements
- Usually performed by customer

Comparison of testing phases

Concept	Unit test	Integration test	System test
What defines test cases	**Module specs**	**Interface specs**	**Requirements**
Visibility	**Code**	**Integration structure**	**No visibility**
Scaffolding	**Complex**	**Some**	**None**
Behavior	**Single module**	**Interaction among modules**	**Sys functionality**

Cleanroom development

- Developed by Harlan Mills when at IBM FSD
- Merge verification with testing strategies
- All code verified before being compiled (e.g., remove computer from the programmer)
  - Forces programmer to be precise
- But, goal is to verify design, not necessarily code

Results:

- It works.
- Verified in many experiments (Some at NASA/SEL)
- But still resisted by industry

Testing Summary

Testing is an activity that extends throughout lifecycle

Planning is essential - Test plan developed during early specifications stage

Need complementary techniques - No silver bullet of a technique that will solve testing problem

Can use tool support - often indispensable