Class 12

- Questions about project
- Assign (see Schedule for links)
  - Project proposal
    - Initial: due by e-mail 9/22/09
    - Final: due (written, 2 pages) 9/29/09

Complex Data Structures

Complex Data Structures

- Arrays, structs, objects
- How to handle them in analysis (pointer, data-flow, …)?

Example:

```
1. struct Bar {
2.     int x;
3.     int y;
4. };
5. main() {
6.     struct Bar b;
7.     b.x = read();
8.     print(b.y);
9. }
```

What is Def(7)?
What is Use(8)?

Complex Data Structures

- Arrays, structs, objects
- How to handle them in analysis (pointer, data-flow, …)?

Example:

```
1. struct Bar {
2.     int x;
3.     int y;
4. };
5. main() {
6.     struct Bar b;
7.     b.x = read();
8.     print(b.y);
9. }
```

What is Def(7)?
What is Use(8)?

Depends on how entities are considered

- As a whole:
  - Def(7) = {b}
  - Use(8) = {b}
  - → spurious du pair
- Distinguishing fields
  - Def(7) = {b.x}
  - Use(8) = {b.y}

Dynamic Analysis

Dynamic Analysis

- Definitions
- Comparison to static analysis
- Need for dynamic analysis
- Problems
- Examples

Static vs. Dynamic Analysis

- Static analyses derives information about a product from an overall model of the product (without execution)
  - Results in definitive information about the product that holds for all inputs
- Dynamic analyses gathers information about the product through instrumentation, actual execution, or simulated execution
  - Results in sampling information about the product that holds for those inputs sampled

Static vs. Dynamic Analysis

- Static analyses
  - Intraprocedural
    - AST, control-flow, control-dependence, data-flow, etc.
  - Complicating factors
    - Interprocedural, recursion, pointers
  - Slicing, demand analysis
  - Applications
- Dynamic analyses
  - Instrumentation, profiling
  - Dynamic versions of control-flow, assertions, etc.
  - Applications such as testing, debugging,
- Combinations of static and dynamic analyses

Even if Proof of Correctness...

- Need to test specifications and assumptions about the environment
- Need to determine performance in practice
- Need to test for qualities such as usability, effectiveness of documentation
- Need to simulate the execution of some systems (but limited)
- Etc.

Thus, dynamic analysis is necessary.

Major Problems

- How do you instrument (insert probes) in an efficient way so as not to incur “too much” overhead?
- How do you make sure that the probes don’t change the behavior of the system?
- How do you select the inputs (test cases) for the analysis?
- How do you know if the test cases are “adequate”?
- How do you compare dynamic methods?
- How do you know when to stop analyzing?

Examples of Dynamic Analysis

- Assertions
- Error seeding
- Coverage criteria
- Fault-based testing
- Specification-based testing
- Object-oriented testing
- Regression testing
- Invariant detection

Testing Basics

What is Testing?

Testing: To execute a program with a sample of the input data

- Dynamic technique: program must be executed
- Optimistic approximation
  - The program under test is exercised with a (very small) subset of all the possible input data
  - We assume (hope) that the behavior with any other input is consistent with the behavior shown for the selected subset of input data
  - The opposite of conservative (pessimistic) analysis

Goals of Testing

- Improve software quality by finding errors

  "A test is successful if the program fails" (Goodeneogh, Gerhart, "Toward a Theory of Test Data Selection", IEEE Transactions on Software Engineering, Jan. 85)

Goals of Testing

- Improve software quality by finding errors

  "A test is successful if the program fails"
  (Goodeneogh, Gerhart, "Toward a Theory of Test Data Selection", IEEE Transactions on Software Engineering, Jan. 85)

- Provide confidence in the dependability of the software product

  (A software product is dependable if it is consistent with its specification.)

Testing Techniques

There exists a number of techniques

- Different processes
- Different artifacts
- Different approaches

There are no perfect techniques

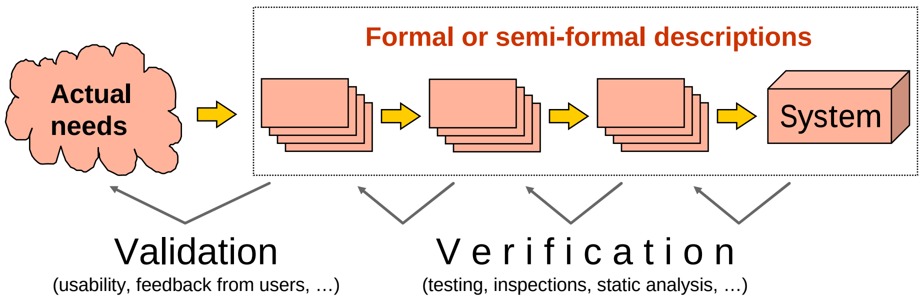
- Testing is a best-effort activity

There is no <u>best</u> technique

- Different contexts
- Complementary strengths and weaknesses
- Trade-offs

Verification and Validation

- **Validation: Are we building the right product?**
  To what degree the software fulfills its (informal) requirements?
- **Verification: Are we building the product right?**
  To what degree the implementation is consistent with its (formal or semi-formal) specification?



Dependability

Correctness

Absolute consistency with a specification

Reliability

Likelihood of correct behavior in expected use

Robustness

Ability of software systems to function even in abnormal conditions

Safety

Ability of the software to avoid dangerous behaviors

The Problem of Verification

The halting problem is not a purely theoretical result

- Most interesting properties of programs' behavior can be reduced to the halting problem
- Verification is almost always an undecidable problem

We must accept inaccuracy!

But, what if we exhaustively test our program?

Exhaustive Testing?

How long would it take (approximately) to test exhaustively the following program?

```
int sum(int a, int b) {return a + b;}
```

Assume int is 32 bits, how many tests would you need?

Exhaustive Testing?

How long would it take (approximately) to test exhaustively the following program?

```
int sum(int a, int b) {return a + b;}
```

Assume int is 32 bits, how many tests would you need?

- $2^{32} \times 2^{32} = 2^{64} =\sim 10^{19}$ tests
- Assume 1 test per nanosecond (10^9 tests/second)
- we get 10^{10} seconds…

Exhaustive Testing?

How long would it take (approximately) to test exhaustively the following program?

```
int sum(int a, int b) {return a + b;}
```

Assume int is 32 bits, how many tests would you need?

- $2^{32} \times 2^{32} = 2^{64} =\sim 10^{19}$ tests
- Assume 1 test per nanosecond (10^9 tests/second)
- we get 10^{10} seconds…

- <u>About 600 years</u>!

Exhaustive Testing is Impossible

"Many new testers believe that

- they can fully test each program, and
- with this complete testing, they can ensure that the program works correctly.

On realizing that they cannot achieve this mission, many testers become demoralized. [...] After learning they can't do the job right, it takes some testers a while to learn how to do the job well."

(C. Kaner, J. Falk, and H. Nguyen,
"Testing Computer Software", 1999)

Failure, Fault, Error

Failure

Observable incorrect behavior of a program. Conceptually related to the behavior of the program, rather than its code.

Fault (bug)

Related to the code. Necessary (not sufficient!) condition for the occurrence of a failure.

Error

Cause of a fault. Usually a human error (conceptual, typo, etc.)

Failure, Fault, Error: Example

```
1.  int double(int param) {
2.      int result;
3.      result = param * param;
4.      return(result);
5.  }
```

- A call to double(3) returns 9
- Result 9 represents a failure
- Such failure is due to the fault at line 3
- The error is a typo (hopefully)

Coincidental Correctness

IMPORTANT: Faults don't imply failure - a program can be coincidentally correct if it executes a fault but does not fail

For example, **double**(2) returns 4

Function **double** is coincidentally correct for input 2

Oracle

An oracle predicts the expected results of a test and is used to assess whether a test is successful or not.

There are different kinds of oracles:

- Human (tedious, error prone)
- Automated (expensive)

Granularity Levels

Unit testing: verification of the single modules

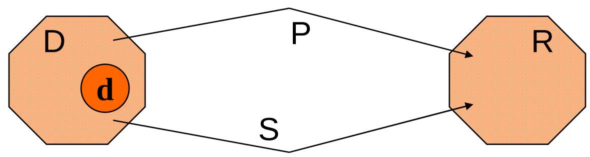
Integration testing: verification of the interactions among the different modules

System testing: testing of the system as a whole

Acceptance testing: validation of the software against the user requirements

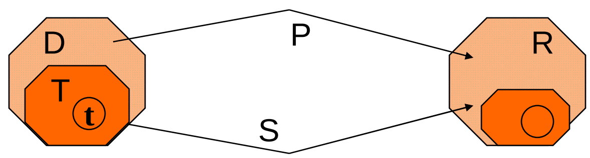
Regression testing: testing of new versions of the software

Correctness



- A program **P** is a function from a set of data **D** (domain) to a set of data **R** (co-domain)
- **P(d)** denotes the execution of **P** with input **d** $\in$ **D**
- **P** is correct iff, $\forall$ **d** $\in$ **D, P(d)** = **S(d)**

Test Suite, Test Set, Test Case



- A test suite or test set **T** for **P** is a subset of **D** x **R**
- An element **t** of **T** is called a test case
- **T** is an ideal test suite for **P** iff the correctness of **P** for all **t** in **T** implies the correctness of **P** for the whole **D**
- In general, it is impossible to define an ideal test suite and we try to approximate it by suitably defining *test selection criteria*

Test Selection Criteria

- Test Selection Criterion C: a rule for selecting the subset of D to place in T
- We want a C that gives test suites that guarantee correctness
- We settle for a C that gives test suites that improve confidence
- Types of criteria:
  - black-box: based on a specification
  - white-box: based on the code

Complementary

Test Adequacy Criteria

Test selection criteria are used to guide the selection of a test suite T: we select test cases that covers some percentage of “coverable” items (as defined by the criteria).

The same criteria can also be used as test adequacy criteria: the adequacy score of T is the percentage of “coverable” items (as defined by the criteria) that are covered by T

Test Requirements, Test Specifications

Test Requirements: those aspects of the program that must be covered according to the considered criterion

Test Specification: constraints on the input that will cause a test requirement to be satisfied

White Box vs. Black Box

Black box

- Is based on a functional specification of the software
- Depends on the specific notation used
- Scales because we can use different techniques at different granularity levels (unit, integration, system)
- Cannot reveal errors depending on the specific coding of a given functionality

White box

- Is based on the code; more precisely on coverage of the control or data flow
- Does not scale (mostly used at the unit or small-subsystem level)
- Cannot reveal errors due to missing paths (i.e., unimplemented parts of the specification)

Selection Criteria: Example

- Specification: function that inputs an integer *param* and returns half the value of *param* if *param* is even, *param* otherwise.
- Implementation

```
int half(int param) {
    int result;
    result=param/2;
    return (result);
}
```

- Function *half* works correctly only for even integers
- The fault may be missed by white-box testing (100% coverage with any value)
- The fault would be easily revealed by black-box testing (typically, we would use at least one odd and one even input)

Selection Criteria: Example

- Specification: function that inputs an integer and prints it
- Implementation:

```
void printit(int param) {
    if(param < 1024) printf("%d", param);
    else printf("%d KB", param/124);
}
```

- Function printit contains a typo
- From the black-box perspective, integers < 1024 and integers > 1024 are equivalent, but they are treated differently in the code
- The fault may be missed by black-box testing
- The fault would be easily revealed by white-box testing (e.g., using the statement coverage criterion)