CSC 408F/CSC2105F Lecture Notes

These lecture notes are provided for the personal use of students taking CSC 408H/CSC 2105H in the Fall term 2004/2005 at the University of Toronto.

Reading Assignment

van Vliet	Chapter	13	(this lecture)
van Vliet	Chapter	7	(next lecture)

Debugging vs. Testing

- *Debugging* is the process of finding errors in a program under development that is not thought to be correct [fix errors]
- *Testing* is the process of attempting to find errors in a program that is thought to be correct. Testing attempts to establish that a program satisfies its Specification [prevent errors]
- **Testing can establish the presence of errors but cannot guarantee their absence** (E.W. Dijkstra)
- Exhaustive testing is not possible for real programs due to combinatorial explosion of possible test cases
- Amount of testing performed must be balanced against the cost of undiscovered errors
- *Regression Testing* is used to compare a modified version of a program against a previous version

Toy Testing vs. Professional Testing

- Toy Testing
  - Small quantity of test data
  - Test data selected haphazardly
  - No systematic test coverage
  - Weak analysis of test output
- Professional Testing
  - Large quantity of test data
  - Systematic selection of test cases
  - Test coverage evaluated and improved.
  - Careful evaluation of test output
  - Heavily automated using scripts and testing tools.

Testing & Bad Attitude

- The software testers job is to find as many errors as possible in the program under test with the least effort expended
- Productivity measure is errors discovered per hour of testing
- Testers attitude should be

  What is the absolutely worst thing I can do to the program?

  and not

  What can I do to make this program look good?

  Don’t let developer’s *ego* stand in the way of vigorous testing.
- Test case selection is one key factor in successful testing
- Insight and imagination are essential in the design of good test cases
- Testers should be independent from implementors to eliminate oversights due to propagated misconceptions [verification and validation ...]

Levels of Program Correctness

1. No syntax errors [compilers and strong-typed programming languages]
2. No semantic errors
3. There exists some test data for which program gives a correct answer
4. Program gives correct answer for reasonable or random test data
5. Program gives correct answer for difficult test data
6. For all legal input data the program gives the correct answer
7. For all legal input and all likely erroneous input the program gives a correct or reasonable answer
8. For all input the program gives a correct or reasonable answer

Faults and Failures

- A *fault* is an error in a software system

  A *failure* is an *event* during which a software system failed to perform according to its requirements specification.

- Failures are caused by (one or more) faults.
- Many possible reasons for failure:
  - Incorrect or missing requirement
  - Incorrect or unimplementable specification
  - System design may contain a fault.
  - Program design may contain a fault.
  - Program implementation may contain a fault.
- The purpose of testing is to discover faults and prevent failures.

Causes of Faults During Development

Requirements	Incorrect, missing or unclear requirements.
Specification	Incorrect translation to design
Design	Incorrect or unclear specification.
Implementation	Misinterpretation of design
	Incorrect documentation
	Misinterpretation of programming language semantics. [a small quiz]
Testing	Incomplete test procedures
	New faults introduced by fault repair.
Maintenance	Incorrect documentation.
	New faults introduced by fault repair.
	Changes in requirements.

A small quiz on Java language

```
int a[2] = {4, 4};
int b;
b = 1;
a[b] = b = 0;
--------------
while (1) {
}
--------------
1 << 32;
--------------
String a="12", b="12";
a == b;
```

Types of Faults

- Algorithmic faults - incorrect algorithm
- Language usage faults - misunderstand/misuse language.
- Computation and precision faults.
- Documentation faults.
- Stress or overload faults.
- Capacity or boundary faults.
- Throughput or performance faults.
- Recovery faults.
- Hardware and system software faults.

Testing should be a Planned Activity

- Testing should be planned for during Requirements Definition and Specification. Allocate human and computer resources for testing
- May need to design hooks for testing into the software
- May need to develop testing tools, test drivers, databases of test data, etc.
- Testability should be one of the requirements of every software system
- A **Test Plan** is usually developed to describe the testing process in detail
- Testing must be planned for in the overall project schedule
  Allow time for fixing problems discovered during testing
- Testing activity should be traceable back to requirements and specification.

Testing Documentation

- Testing requirements and *Test Specification* are developed during the requirements and specification phases of the system design
- *Test Plan* describes the sequence of testing activities and describes the testing software that will be constructed
- *Test Procedure* specifies the order of system integration and modules to be tested. It also describes unit tests and test data
- Logging of tests conducted and archiving of test results is often an important part of the Test Procedure

Test Plan Components

- Establish test objectives
- Designing test cases
- Writing test cases
- Testing test cases
- Executing tests
- Evaluating test results

Types of Testing

- *Unit testing* - test individual module or small set of modules
- *Integration testing* - Test entire system for correct operation
- *Performance testing* - Test that system satisfies performance and capacity requirements.
- *Software Quality Assurance* - Independent testing to assess quality of the software
- *Acceptance testing* - Test system to see if it satisfies specifications for the purpose of project completion
- *Installation testing* - Test system installation procedures and correct operation in target environment.
- *User Testing* - inflict poorly tested software on the end users and let them find the bugs
- *Regression testing* - Test system after a modification, compare output for standard test cases on old and new systems

Unit Testing

- Testing at the module level
- Test information flows across module interface
- Test module's handling of its local data structure
- Test boundary conditions for input
- Test all control flow paths in the module
- Execute all statements at least once
- Test error-handling and response to bad input
- Develop test cases for all of the above

  Unit testing is made easier by high cohesion
- Write driver program to read test cases and call module

  Write stubs to simulate modules called by module under test

Integration Testing

- Incremental vs. non-incremental system integration

  Incremental is almost always easier, more effective and less chaotic

- Approaches to Incremental Integration

  - *Top Down* - Start with main program module, gradually add subordinate modules.

    Could use breadth-first or depth-first strategy.

    Test system as each new modules is added.

    Regression test previous models as required

  - *Bottom Up* - Start with atomic (leaf) modules.

    Use drivers to tests clusters of modules.

    Gradually add calling modules and move testing upward

Top-Down Integration

- Advantages
  - Verifies major control and decision functions early in testing

    This is a big win if there are major control problems.
  - Depth-first strategy makes parts of system available early for demo.
- Disadvantages
  - Building stubs adequate to test upper levels may require a lot of effort

    May not be able to do complete top-down integration
  - A lot of effort (later discarded) is put into designing and implementing drivers and stubs

Bottom-Up Integration

- Advantages
  - Stubs aren’t required
- Disadvantages
  - Entire system isn’t available until end of integration
  - Won’t detect major control/data/interface problems until relatively late in the testing process

Practical Integration Testing

- Identify *critical* modules and test them early

  Critical modules:

  - Address multiple software requirements
  - Have a high level of control
  - Is particularly complex or error-prone
  - Have definite performance requirements
  - Have a high coupling with other modules
  - Appear to be potential performance bottle-necks
  - **Are on the critical path of the project**

- Use *sandwich testing* as a compromise between top-down and bottom-up
  - Use top-down strategy for upper levels of the program
  - Use bottom-up strategy of lower levels of the program
  - *Carefully* meet in the middle

Validation Testing

- Post integration test that entire system satisfies its Software Requirements Specification
- Black box tests to demonstrate conformity with requirements
- Should test *all* functional and performance requirements
- All deviations from Requirements (i.e. failed tests) are documented and a plan is devised (negotiated) for correcting the deficiencies
- *Configuration Review* verifies that all software elements have been properly developed and documented and are maintainable
- Software developed for a single customer is subjected to an *Acceptance Test*
  Multi-customer software is subjected to alpha and beta testing
- alpha testing - system is tested by real (friendly, interested) users with close observation by developer
- beta testing - system is tests by real (friendly, interested) users at their own sites with less direct observation by the developer

System Testing

- Higher level tests to determine suitability and performance of entire *system* (i.e. hardware, software and procedures)
- *Recovery Testing* - Test that system can recover from faults (e.g. trashed data base) and continue processing
- *Security Testing* - Test that system provides adequate protection for access and modification of data. Often performed by specialist *system crackers*
- *Stress Testing* - Test performance of system at high levels of load. Very large inputs or very high levels of concurrent usage
- *Performance Testing* - Test run-time performance and response of complete system

Types of Testing

- White Box Testing
  - Uses source program as source of test information
  - Can design tests to exercise all paths through program, all conditionals true and false branches
  - Execute all loops at boundaries
  - Test operation at/near implementation limits
  - Identify possible performance bottlenecks
- Black Box Testing
  - Test without knowledge of source program
  - Use requirements, specifications and documentation to derive test cases
  - Test for missing or incorrect functionality
  - Test for interface errors. Test system performance
  - Test for errors in external data structures or database access
  - Test for initialization and termination errors

Testing Styles

- Top Down Testing
  - Start at subsystem level using module stubs
  - Test at module level using procedure stubs
  - Test complete system
  - Advantages: detect major design errors early, may allow early use of system
  - Disadvantages: Requires many test stubs, hard to do and expensive
- Bottom Up Testing
  - Start testing at procedure level using driver code
  - Test at module level using module drivers
  - Test at subsystem level
  - Advantages: easier to create test cases and drivers
  - Disadvantages: find high level design errors late, no working system until late

Test Adequacy Criteria

- The *test adequacy criteria* is a non-functional requirement for a software system that describes how thoroughly the system should be tested.
- The test adequacy criteria can be used
  - to decide when to stop testing
  - as a way to generate tests
  - as a standard to measure testing progress.
  - as a guide for adding tests to a test suite.
- Possible testing adequacy criteria include

  Test coverage criteria What fraction of the program has been tested by a test suite

  Fault Seeding Criteria What fraction of some set of faults deliberately added (seeded) to a program have been discovered by testing?

  Graph based criteria What fraction of the programs control flow graph has been covered by a suite of tests.

Testing Coverage

- One way to assess the thoroughness of testing is to look at the *testing coverage*.
- Testing coverage measures what fraction of the program has actually been exercised by a suite of tests.
- The ideal situation is for 100% of the program to be covered by a test suite. In practice 100% coverage is very hard to achieve.
- In analyzing test coverage we think in terms of the programs control flow graph which consists of a large number of nodes (basic blocks, serial computation) connected by edges that represent the branching and function call structure of the program.
  Coverage can be analyzed in terms of program control flow or in terms of data flow.
- Automated tools can be used to estimate the coverage produced by a test suite.

Control Flow based Coverage

Statement coverage - every statement (i.e. all nodes in the programs control flow graph) is executed at least once.

All-Paths coverage - every possible control flow path through the program is traversed at least once. Equivalent to an exhaustive test of the program.

Branch coverage - for all decision points (e.g. if and switch) every possible branch is taken at least once.

Multiple-predicate coverage Boolean expressions may contain embedded branching (e.g. `A && ( B || C )`). Multiple-predicate coverage requires testing each Boolean expression for all possible combinations of the elementary predicates.

Cyclomatic number coverage All linearly independent paths through the control flow graph are tested.

Data Flow Definitions

Definition A variable is *defined* in a statement if it is assigned a (new) value in the statement.

Definition-clear A path through the program is definition clear for some variable if it does *not* contain a definition (i.e. assignment) for the variable.

Liveness A definition from statement X is *alive* in some other statement Y statement if there is a definition-clear path from X to Y.

*This means that value assigned in statement X **could** be the value used in statement Y*.

P-use (predicate-use) The use of a variable in a predicate is called a P-use.

P-uses could affect the flow of control through the program

C-use (computational use) All uses of a variable that are not P-use are C-use.

C-uses could affect the values computed by the program.

Data Flow Based Coverage

All-uses coverage Traverse at least once every definition-free path from every definition to all P-use or C-use of that definition.

All-DU-Path coverage All-uses coverage plus the constraint that every definition clear path is either cycle-free or a simple cycle.

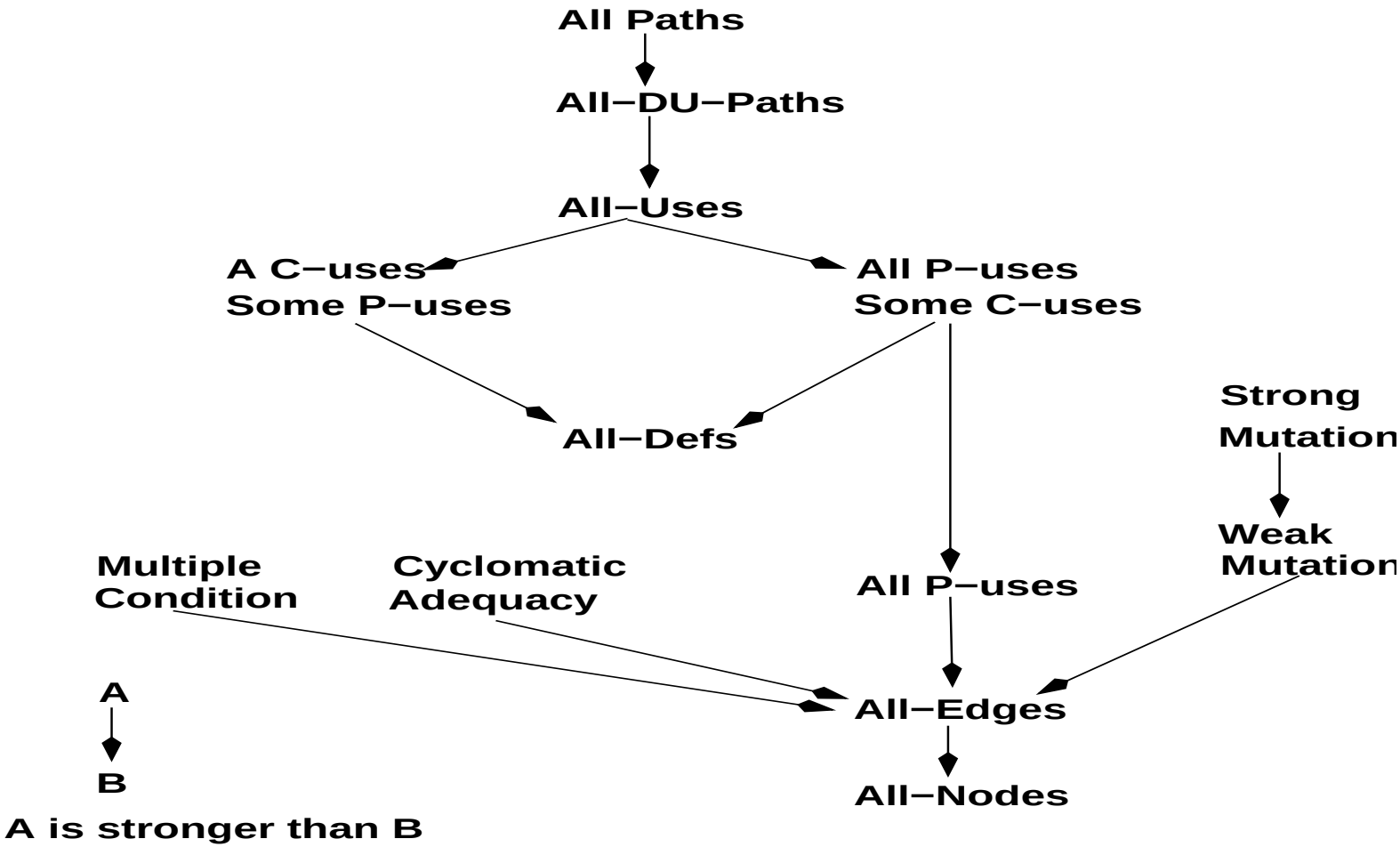
All-defs coverage Test so that each definition be used at least once.

All-C-uses/Some P-uses coverage Test so that all definition free paths from each definition to all C-uses are tested. If a definition is used only in predicates, test at least one P-use.

All-P-uses/Some C-uses coverage Test so that all definition free paths from each definition to all P-uses are tested. If a definition is used only in computations, test at least one C-use.

All-P-uses coverage Test so that all definition free paths from every definition to every P-use is tested.

Subsume Hierarchy for Program Based Testing[a]



[a]van Vliet Figure 13.17

Fault Seeding Techniques

- Fault seeding is used to estimate the number of faults in a program and to estimate the thoroughness of a test suite.
- The fault seeding technique
  - Some number of faults NS are added to a program.
    Need to take care that the seeded faults are similar to the real faults.
  - The program is tested and some number of faults NF are found.
    Let NSF be the number of *seeded faults* that were found.
  - The percentage of seeded faults that were found is used to estimate the total number of faults in the program using the formula

$$Total\_faults = \frac{(NF - NSF) \cdot NS}{NSF}$$

- If most of the faults found are the seeded faults, the results can probably be trusted. Otherwise the software probably needs a lot more testing.

Error Based Testing

- This testing technique is based on assumptions about probable types of errors in program.
- Historical experience tells us that many program errors occur at boundary points between different domains
  - input domains
  - internal domains (processing cases)
  - Output domains
- Error based testing tries to identify such domains and generate tests on both sides of every boundary.

Sources for Test Cases

- Requirements and Specification Documents
- General knowledge of the application area
- Program design and user documentation
- Specific knowledge of the program source code
- Specific knowledge of the programming language and implementation techniques
- Test at and near (inside and outside) the boundaries of the programs applicability
- Test with *random* data
- Test for response to probable errors (e.g. invalid input data)
- **Think nasty when designing test cases.**
  Try to destroy the program with your test data

More on White Box Testing

- *Basic Path Testing* - design test cases to guarantee that every path in the programs control flow graph is executed at least once

  Note that Boolean expressions may imply embedded control flow

  Derive test cases from examination of program flow graph

- *Condition Testing* - design test cases that all possible outcomes for each condition (Boolean expression) in the program

- *Branch testing* - design test cases to cause each condition to evaluate to true and false

- *Domain testing* - more rigorous testing of conditions to find relational operator errors

More on White Box Testing

- *Data Flow Testing* - design tests to link definition (i.e. value assignment) and use of variables in the program

  Try to execute every definition-use chain at least once

- *Simple Loop Testing* - design test cases to exercise every loop in the program
  - Loop not executed at all - tests code after loop for correct handling of null case
  - Loop executed once - tests initialization and exit condition
  - Loop executed twice - tests passing of values between iterations
  - Loop executed random legal number of times
  - Loop executed one less than allowable maximum
  - Loop executed exactly allowable maximum
  - Loop executed one more than allowable maximum

More on Black Box Testing

- *Equivalence Partitioning* - divide input domain(s) into classes such that testing with a member of the class is equivalent to testing with all members of the class.

  Define equivalence classes for valid and invalid data items

- *Boundary Value Analysis* - Observed that many errors occur at or near boundaries of the input domain. So test heavily in this area.

  Examples: null input, maximum size input

- *Comparison Testing* - Compare outputs from multiple independently implemented versions of the program. Used primarily for reliability critical systems

Testing - Example

Program	Search an array for a given value
	function Search(*Ar* : **array** * .. * **of int** , *val* : **int**) : **int**
Specification	if *val* is an element of the array *Ar* then Search returns its index in Ar. Otherwise Search returns -1
Aside	What's wrong with this Specification?

Test Data for Search

Array with zero elements

Array with one element

- val in, not in below, not in above

Array with random even size

- val not in, in random, in first, in last, in middle +- 1

Array with random odd size

- val not in, in random, in first, in last, in middle, in middle +- 1

Array with two elements

- val not in below, not in above, in first, in last

Arrays containing ordered, reverse ordered and unordered data

Array random size containing all one value

Array of maximum allowable size

Array with upper bound of largest allowable integer

Array with lower bound of smallest allowable integer

Array with negative upper and lower bounds

Array containing largest and smallest integers

Code Inspections to Reduce Errors[a]

- A *Code Inspection* is a formal, systematic examination of the source code for a program by peer developers
- The primary goal of Code Inspection is the *detection of errors*
- Code Inspection Team
  - Moderator: run inspection meeting, logs errors
    Ensures inspection process is followed
  - Inspectors: Knowledgeable peer designers
  - Author: present as a *silent* observer
    Assists only when explicitly requested

[a] G. Russell, Inspection in Ultralarge-Scale Development, IEEE Software, January 1991

Code Inspection Process

- Start with debugged, compilable, untested code
- Inspectors read/study source code before inspection session
- Inspectors *paraphrase* (describe verbally) the source code a few lines (≤ 3) at a time
- Proceed slowly intentionally

  ≤ 150 source lines/hour, ≤ 2 hours/session
- Detected errors are summarized & logged

  No attempt is made to correct errors on-the-fly
- Slow pace is intended to produce intense scrutiny of the source code
- **Inspections usually find the cause of an error.**

  Testing usually finds the symptoms of the error

BNR Code Inspection Example

- BNR produced approximately 20,000,000 lines/code in 10 years
  DMS digital switch software is about 10,000,000 lines
- Inspected 2,500,000 lines, 8 releases, over 2 years
- Found 0.8 . . . 1 errors per person hour inspecting
  Inspection is 2 . . . 4 times *more effective* than testing
- Found about 37 errors per 1000 lines of code
  (Other studies found 50 .. 95 errors per 1000 lines)
  Type of error: 50.6% incorrect statement, 30.3% missing statement
  19.1% extra statement
- An error discovered in *released* software takes about 33 person hours to diagnose and repair
- For their large, real time systems, a designer produces 3,000 .. 5,000 lines of finished, documented code *per person year*

Software Testing Strategies

- Test from module level outward
- Use different kinds of testing for different levels of the system
- Testing by an independent group increases confidence in the testing result
  - Developers make poor testers. Unwilling to make own program look bad
  - Developers will have strongly held views about how program works. These view may blind them to errors and omissions
  - Developers will often test at the module level, independent testers test entire system
- Debugging should occur before testing.
  It's easier to test a (mostly) correct program
- Testing activity *must* be planned for during system design
  Preparation for testing (e.g. tool & test case building) should start well before testing is scheduled to begin

Independent Test Groups

- Separate software group with a mandate for testing

  Often organizationally separate from development team

- Works with developers during system specification and design

  Design in hooks for testing

  Make sure specifications are testable

- Developers work with testers during testing to repair errors

How Much Testing is Enough?

- *Theory:* Never finished testing
- *Practice:* Must stop sometime. Economic issue.

  Users continue testing for lifetime of system
- Try to achieve balance between the benefits of finding errors before the user and the (people and hardware) costs of testing
- Use statistical models to estimate remaining errors in system
- Use test coverage to measure thoroughness of testing.
- Use results of testing to monitor rate of discovery of new errors in the software.

  Typically find many errors at the start of testing, error discovery rate tapers off once all the ”easy” errors have been found

Why Testing Can Fail

- Assume general control flow model for program

  Computations (actions) and conditional branching

- Missing Control Flow Paths

Fault	failure to test a condition
Causes	(non) execution of (in)appropriate action
Arises	failure to see that some (combination of condition(s) requires a unique action
Implies	Execution of all control flow paths won't detect the error

- Inappropriate Path Selection

Fault	A predicate is expressed incorrectly
Causes	Action is performed or omitted under inappropriate conditions
Arises	Coding error or misunderstanding
Implies	Exercising all statements and all path predicates won't detect the error

- Inappropriate or Missing Action

Fault	An action is incorrect or missing
	e.g. faulty expression, variable not assigned a value
	calling procedure with incorrect arguments
Causes	Incorrect output or crash
Arises	Coding error or misunderstanding
Implies	Executing incorrect action may or may not find the error
	Forcing all statements to be executed helps
	Use profile tool to confirm statement execution

Testing Automation

- Desirable: automate the the testing process to reduce testing costs
- Automation techniques
  - Test case generators, process Requirements or Specification documents
  - Test case data bases, record test cases and expected results
  - Test drivers, run all test cases and record results
  - Test result comparators, for regression testing
  - Automatic logging of test runs for legal requirements

Program Analysis Tools

- Static (source code) analysis
  - Code analyzers, e.g. lclint
  - Structure checker
  - Data analyzer - checks linkage, proper usage
  - Dependency analyzer, e.g. makedepend
- Dynamic analysis
  - Execution flow summarizers
  - Test coverage tools
  - Debugging memory allocators

Execution Flow Summarizers

- Software tool to show where time is spent during program execution
- Flow summary process
  - Instrument program (if required)
  - Execute program with representative test data

    Profile tool records execution control flow data
  - Post process execution control flow data to generate coverage reports
- Used to locate *Hot Spots* for program optimization

  90-10 Rule:

  10% of the program is responsible for 90% of its execution time
- Used to confirm test case coverage

  Find parts of program that are not being exercised by a given set of test cases

Execution Flow Summarizers

- Exact solution
  - Compiler or preprocessor inserts counters in program to tabulate control flow
  - Run program, dump counters to file at end of execution
  - Minimum counters, one per edge in control flow graph
  - *Advantages:* exact solution, links to source code
  - *Disadvantages:* requires source code, counters may distort execution characteristics, preprocessor/compiler modifications, only works for one language
- Approximate solution
  - Asynchronously sample program counter of executing program
  - Generate histogram of execution time vs. relative position in program
  - *Advantages:* works on binary programs, multilingual, easy to build, finds problems in run-time packages
  - *Disadvantages:* inexact solution, harder to relate to source code, timer interrupts may distort execution profile, time interrupts may be too coarse on fast machines

Other Testing Tools

- Differential File Comparators
  - Tools to extract the *differences* between two or more files (.e.g diff, rcsdiff)
  - Use to analyze output from testing, to compare outputs from different test runs
  - May need *smart* diff if some test outputs can legitimately be different (e.g. dates & times)
  - Differential file compare heavily used in regression testing
- Database packages
  - Used to manage test data for large systems
  - Used to manage test output for large systems
- File processing utilities
  - Programs like fgrep, awk, vi, sed
  - Used to extract interesting results testing output
  - Used to postprocess test output to produce summaries

- Simulators and Test Drivers
  - Simulator imitates some part of the programs intended environment
    Used to test input/output, transaction handling
  - Allows exact replication of input data, control of input timing
  - Allows testing of system with many concurrent inputs, stress testing
  - Many simulators are script or table driven
  - Simulate unavailable hardware devices, replace expensive hardware for some testing
  - *Disadvantage:* if simulator doesn't provide *exact* simulation then testing results can be misleading

  [JUnit, CppUnit, see Tutorial 4]