CMPE 187 - Software Quality Engineering

Chapter 1 – Basic Concepts and Preliminaries

The quality revolution

Statistical quality control (SQC) is a discipline based on measurement and statistics

- SQC methods use seven basic quality management tools.
    - Pareto analysis, Trend Chart, Flow chart, Histogram, Scatter diagram, Control chart, Cause and effect diagram

"Lean principle" was developed by Taiichi Ohno of Toyota

- "A systematic approach to identifying and eliminating waste through continuous improvement, flowing the product at the pull of the customer in pursuit of perfection."

Plan – Establish the objective and process to deliver the results.
Do – Implement the plan and measure its performance.
Check – Assess the measurements and report the results to decision makers.
Act – Decide on changes needed to improve the process.

Key elements of TQC (Total Quality Control):

- Quality comes first, not short-term profits
- The customer comes first, not the producer
- Decisions are based on facts and data
- Management is participatory and respectfully of all employees.
-

Five Views of Software Quality:

- **Transcendental view**: What is quality, you know it when you see it.
- **User's view**: prestige, appearance, covers your needs, trust
- **Manufacturing view**: High quality manufacturing gives high quality products / manufacturing process is good = product is good.
- **Product view**:
- **Value-based view**: value added feeling

Software Quality in terms of quality factors and criteria

- A quality factor represents behavioral characteristic of a system
- A quality criterion is an attribute of a quality factor that is related to software development

Quality Models, examples: ISO 9126, CMM, TPI and TMM

Software quality assessment divide into two categories:

- Static analysis
    - It examines the code and reasons over all behaviors that might arise during rum time. (Ex. Code review, inspection and algorithm analysis)
- Dynamic analysis

- Actual program execution to expose possible program failure
- One observes some representative program behavior, and reach conclusion about the quality of the system

Static and Dynamic Analysis are complementary in nature
Focus is to combines the strengths of both approaches.

Verification

- Evaluation of software system that help in determining whether the product of a given development phase satisfy the requirements established before the start of that phase (Building the product correctly).

Fan in, fan out: refers to dependency. *Fan-in* is the number of inputs a gate can handle. *fan-out* of a logic gate output is the number of gate inputs it can feed or connect to

Validation

- Evaluation of software system that help in determining whether the product meets its intended use (Building the correct product).

Failure: A *failure* is said to occur whenever the external behavior of a system does not conform to that prescribed in the system specification. Something that deviates from the requirements
Error: An *error* is a state of the system. An *error* state could lead to *failure* in the absence of any corrective action by the system.
Fault: A *fault* is the adjudged cause of an *error*
Defect: It is synonymous of *fault*, aka *bug*

Semantic: meaning to the whole domain

The Notion of Software Reliability

- It is defined as the probability of failure-free operation of a software system for a specified time in a specified environment
- It can be estimated via *random testing*
- Test data must be drawn from the input distribution to closely resemble the future usage of the system.
- Future usage pattern of a system is described in a form called ***operational profile***.

Test case is a simple pair of <input, expected outcome>
State-less systems: A compiler is a stateless system. Outcome depends solely on the current input. Value of all the variables of the system, current value of the variables.
State-oriented: ATM is a state oriented system.

- Test cases are not that simple, a test case may consist of a sequences of <input, expected outcome>.
- The outcome depends both on the current state of the system and the current input.

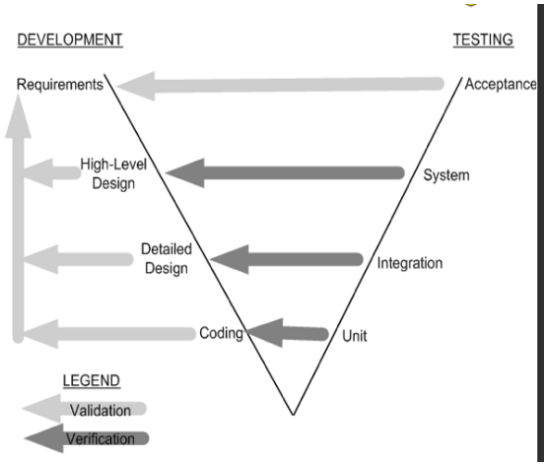
Expected outcome
An outcome of program execution may include

- Value produced by the program
- State Change
- A sequence of values which must be interpreted together for the outcome to be valid.

A *test oracle* is a mechanism that verifies the correctness of program outputs
- Generate expected results for the test inputs.
- Compare the expected results with the actual results of execution of the IUT.

The concept of Complete Testing
- Complete or exhaustive testing means: "There are no undisclosed faults at the end of test phase"
- Complete testing is near impossible for most of the system
    - The domain of possible inputs of a program is to large, Valid inputs and Invalid input.
    - The design issues may be too complex to completely test
    - It may not be possible to create all possible execution environments of the system.

Testing Level
- Unit testing:
    - Individual program units, such as procedure, methods in isolation
- Integration testing:
    - Modules are assembled to construct larger subsystem and tested
- System testing:
    - Includes wide spectrum of testing such as functionality, and load
- Acceptance testing:
    - Customer's expectations from the system
    - Two types of acceptance testing:
        - UAT: System satisfies the contractual acceptance criteria
        - BAT: System will eventually pass the user acceptance test



White-box and Black-box Testing
- White-box testing a.k.a. structural testing
- Examines source code with focus on:
    - Control flow
    - Data flow
- Control flow refers to flow of control from one instruction to another
- Data flow refers to propagation of values from one variable or constant to another variable
- It is applied to individual units of a program
- Software developers perform structural testing on the individual program units they write

- Black-box testing a.k.a. functional testing
- Examines the program that is accessible from outside
- Applies the input to a program and observe the externally visible outcome
- It is applied to both an entire program as well as to individual program units
- It is performed at the external interface level of a system
- It is conducted by a separate software quality assurance group

Test Planning and Design

- The purpose is to get ready and organized for test execution
- Test plan provides a:
    - **Framework**: A set of ideas, facts or circumstances within the tests will be conducted
    - **Scope**: The domain or extent of the test activities
    - **Details of resource needed**
    - **Effort required**
    - **Schedule of activities**
    - **Budget**
- Test objectives are identified from different sources
- Each test case is designed as a combination of modular test components called test steps
- Test steps are combined together to create more complex tests

Test **case effectiveness** metrics

- Measure the "defect revealing ability" of the test suite
- Use the metric to improve the test design process

Test-**effort effectiveness** metrics

- Number of defects found by the customers that were not found by the test engineers

Test Tools and Automation

- Increased productivity of the testers
- Better coverage of regression testing
- Reduced durations of the testing phases
- Increased effectiveness of test cases
- The test cases to be automated are well defined
- Test tools and an infrastructure are in place
- The test automation professionals have prior successful experience in automation
- Adequate budget has been allocation for the procurement of software tools

Chapter 2 - Theory of Program Testing

Theory of Goodenough and Gerhart

Program faults occur due to our:

- inadequate understanding of all condition that a program must deal with.
- failure to realize that certain combinations of conditions require special care.

Kind of program faults:

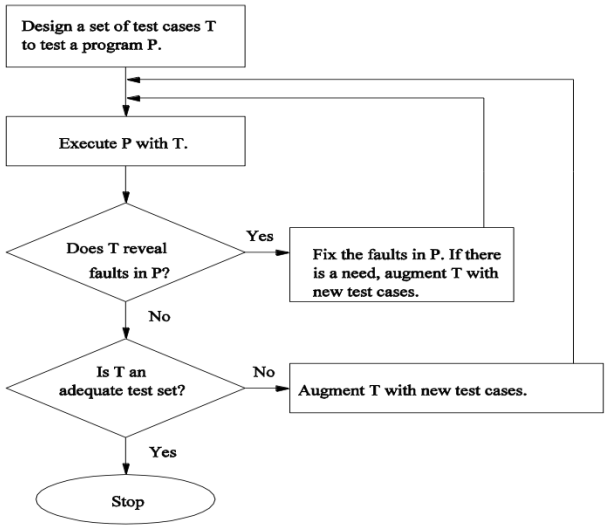
- Logic fault
    - Requirement fault
    - Design fault
    - Construction fault
- Performance fault
- Missing control-flow paths

- Inappropriate path selection
- Inappropriate or missing action

Test predicate: It is a description of conditions and combinations of conditions relevant to correct operation of the program.

Adequacy of Testing

Reality: New test cases, in addition to the planned test cases, are designed while performing testing. Let the test set be T.



If a test set T does not reveal any more faults, we face a dilemma:

- P is fault-free. OR
- T is not good enough to reveal (more) faults.
  - ➔ Need for evaluating the adequacy (i.e. goodness) of T.

Some ad hoc stopping criteria

- Allocated time for testing is over.
- It is time to release the product.
- Test cases no more reveal faults.

Two practical methods for evaluating test adequacy

- **Fault seeding**
  - Implant a certain number (say, X) of known faults in P, and test P with T.
  - If k% of the X faults are revealed, T has k% of the unknown faults.
- **Program mutation**
  - A mutation of P is obtained by making a small change to P.
  - Some mutations are faulty, whereas the others are equivalent to P.
  - T is said to be adequate if it causes every faulty mutation to produce unexpected results.

Limitations of Testing

Dijkstra's famous observation

- Testing can reveal the presence of faults, but not their absence.

The result of each test must be verified with a **test oracle**.

- Verifying a program output is not a trivial task.
- There are **non-testable** programs. A program is non-testable if
  - There is no test oracle for the program.
  - It is too difficult to determine the correct output.

Summary

Theory of Goodenough and Gerhart
- Ideal test, Test selection criteria, Program faults, Test predicates

Theory of Weyuker and Ostrand
- Uniformly ideal test selection
- Revealing subdomain

Theory of Gourlay
- Testing system
- Power of test methods ("at least as good as" relation)

Adequacy of Testing
- Need for evaluating adequacy
- Methods for evaluating adequacy: fault seeding and program mutation

Limitations of Testing
- Testing is performed with a test set T, s.t. |T| << |D|.
- Dijkstra's observation
- Test oracle problem

Chapter 3 – Unit Testing

Concept of Unit Testing

Static Unit Testing

- Code is examined over all possible behaviors that might arise during run time
- Code of each unit is validated against requirements of the unit by reviewing the code

Dynamic Unit Testing

- A program unit is actually executed and its outcomes are observed
- One observes some representative program behavior, and reach conclusion about the quality of the system

Static unit Testing is not an alternative to dynamic unit testing, both are complementary in nature. In practice, partial dynamic unit testing is performed concurrently with static unit testing. Recommended to do static unit testing before dynamic unit testing.

Static Unit Testing

In static unit testing, code is reviewed by applying following techniques:

- **Inspection:** It is a step by step peer group review of a work product, with each step checked against pre-determined criteria
- **Walkthrough:** It is review where author leads the team through a manual or simulated execution of the product using pre-defined scenarios

The idea here is to examine the source code in detail, in a systematic manner and to *review* the code, and *not* the author of the code.

The key to the success of code is to divide and conquer

- An examiner inspects small parts of the unit in isolation
    - Nothing is overlooked
    - The correctness of all examined parts of the module implies the correctness of the whole module

The following metrics can be collected from a code review:

- The number of lines of code (LOC) revied per hour
- The number of CRs generated per thousand lines of code (KLOC)
- The number of CRs generated per hour
- The total number of hours spend on code review process

Five different types of system documents are generated by engineering department:

- Requirement
- Functional Specification
- High-level Design
- Low-level Design
- Code

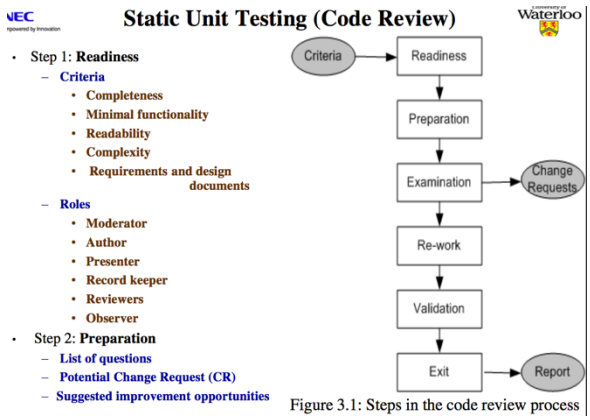


Figure 3.1: Steps in the code review process

A **Change Request (CR)** includes the following details:

- Give a brief description of the issue
- Assign a priority level (major or minor) to a **CR**
- Assign a person to follow it up
- Set a deadline for addressing a **CR**

Defect Prevention

- Build instrumentation code into the code
- Use standard control to detect possible occurrences of error conditions
- Ensure that code exists for all return values
- Ensure that counter data fields and buffer over/underflow are appropriately handled
- Provide error messages and help texts from a common source
- Validate input data
- Use assertions to detect impossible conditions
- Leave assertions in the code
- Fully document the assertions that appears to be unclear
- After every major computation reverse-compute the input(s) from the results in the code itself
- Include a loop counter within each loop.

Dynamic Unit Testing

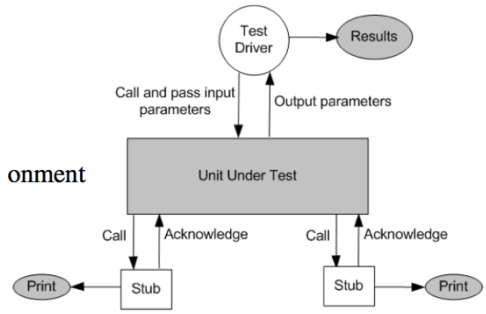
The environment of a unit is emulated and tested in isolation
The caller unit is known as *test driver*

- A *test driver* is a program that invokes the unit under test (UUT)
- It provides input data to unit under test and report the test result

The emulation of the units called by the UUT are called *stubs*

- It is a dummy program

The *test driver* and the *stubs* are together called *scaffolding*
The low-level design document provides guidance for selection of input test data



Selection of test data is broadly based on the following techniques:

- Control flow testing
    - Draw a control flow graph (CFG) from a program unit
    - Select a few control flow testing criteria
    - Identify a path in the CFG to satisfy the selection criteria
    - Derive the path predicate expression from the selection paths
    - By solving the path predicate expression for a path, one can generate the data
- Data flow testing
    - Draw a data flow graph (DFG) from a program unit and then follow the procedure described in control flow testing.
- Domain testing
    - Domain error are defined and then test data are selected to catch those faults
- Functional program testing
    - Input/output domains are defined to compute the input values that will cause the unit to produce expected output values.

Mutation Testing - is used to design new software tests and evaluate the quality of existing software tests. Mutation testing involves modifying a program in small ways.
Mutant: Program with a fault introduced into it
Mutant Killed: If test fails with mutant, in other words test discovers the fault

Equivalent

- Mutant passes test suite
- Mutant is non-killable (need new test cases to discover faults)
- In other words, mutation has no effect

Adequacy of testing

- If a test suite detects all induced faults
- i.e., all mutants killed by a test suite
- Such a test suite is termed mutation – adequate (Mutation score = 1 or 100%)
- Can be used as a test stopping criterion
- Remember we are talking about **Unit Testing**
- Can we apply this to Integration or System Testing, why/why not?

Mutation Score = *# of Mutations Discovered / Total # of Mutations*

Types of Mutations

- Constant replacement
- Scalar variable replacement
- Scalar variable for constant replacement
- Constant for scalar variable replacement
- Array reference for constant replacement
- Array reference for scalar variable replacement
- Constant for array reference replacement
- Scalar variable for array reference replacement
- Array reference for array reference replacement
- Source constant replacement
- Data statement alteration
- Comparable array name replacement
- Arithmetic operator replacement
- Relational operator replacement
- Logical connector replacement
- Absolute value insertion
- Unary operator insertion
- Statement deletion
- Return statement replacement
- Replacing a type with a compatible subtype (inheritance)
- Changing the access modifier of an attribute, a method
- Changing the instance creation expression (inheritance)
- Changing the order of parameters in the definition of a method
- Changing the order of parameters in a call
- Removing an overloading method
- Reducing the number of parameters
- Removing an overriding method
- Removing a hiding Field
- Adding a hiding field

Mutation testing makes two major assumptions:

- Competent Programmer hypothesis
  - Programmers are generally competent and they do not create *random* programs
- Coupling effects
  - Complex faults are coupled to simple faults in such a way that a test suite detecting simple faults in a program will detect most of the complex faults

Debugging

The process of determining the cause of a failure is known as *debugging*. It is a time consuming and error-prone process. It involves a combination of systematic evaluation, intuition and a little bit of luck.

The purpose is to isolate and determine its specific cause, given a symptom of a problem.

There are three approaches to *debugging*

- Brute force
- Cause elimination (Induction / Deduction)
- Backtracking

Tools for Unit Testing

Code auditor

- This tool is used to check the quality of the software to ensure that it meets some minimum coding standard

Bound checker

- This tool can check for accidental writes into the instruction areas of memory, or to other memory location outside the data storage area of the application

Documenters

- These tools read the source code and automatically generate descriptions and caller/callee tree diagram or data model from the source code

Interactive debuggers

- These tools assist software developers in implementing different debugging techniques
    - Examples: Breakpoint and Omniscient debuggers

In-circuit emulators

- It provides a high-speed Ethernet connection between a host debugger and a target microprocessor, enabling developers to perform source-level debugging

Memory leak detectors

- These tools test the allocation of memory to an application which request for memory and fail to de-allocate memory

Static code (path) analyzer

- These tool identify paths to test based on the structure of code such as McCabe's cyclomatic complexity measure

Software inspection support

- Tools can help schedule group inspection

Test coverage analyzer

- These tools measure internal test coverage, often expressed in terms of control structure of the test object, and report the coverage metric

Test data generator

- These tools assist programmers in selecting test data that cause program to behave in a desired manner

Test harness

- This class of tools support the execution of dynamic unit tests

Performance monitors

- The timing characteristics of the software components be monitored and evaluate by these tools

Network analyzers

- These tools have the ability to analyze the traffic and identify problem areas

Simulators and emulators

- These tools are used to replace the real software and hardware that are not currently available. Both the kinds of tools are used for training, safety, and economy purpose

Traffic generators

- These produces streams of transactions or data packets.

Version control

- A version control system provides functionalities to store a sequence of revisions of the software and associated information files under development

Chapter 4 – Control Flow Testing

There are two kinds of basic program statements:

- Assignment statements (Ex. x = 2*y)
- Conditional statements (Ex. if(), for(), while(), …)

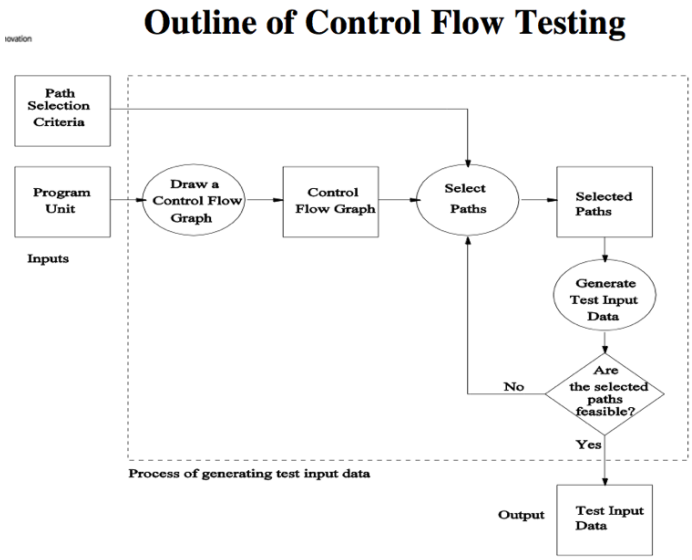
Control flow

- Successive execution of program statements is viewed as flow of control.
- Conditional statements alter the default flow

Program path

- A program path is a sequence of statement from entry to exit.
- There can be a large number of paths in a program.
- There is an (input, expected output) pair for each path.
- Executing a path requires invoking the program unit with the right test input.
- Paths are chosen by using the concepts of path selection criteria.

Tools: Automatically generate test inputs from program paths.



Outline of Control Flow Testing

Control Flow Assumptions

- We have correct specifications
- Definitions of data are correct
- Data can be accessed correctly
- All known bugs have been resolved (Except for the one that affect control flow)

Application: new software, Unit testing

Control Flow Graph: Graphical representation of a program's control structure.

Complete Path

- Path the begins at the entry to a routine and ends at its exit
- Also called Entry – Exit Path

Complete Paths are useful

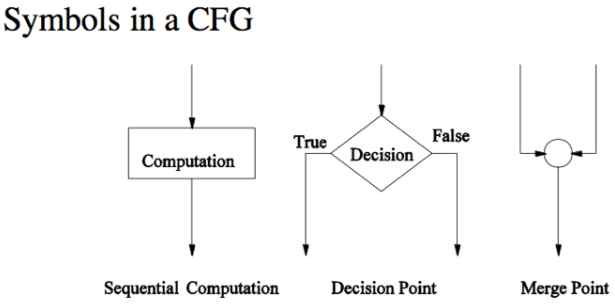
- It is difficult to start execution at an arbitrary statement
- It is difficult to start execution at an arbitrary statement
    - Without modifying code

There can be many paths between entry and exit

- Even small/short routines can have a large number of paths

Control Flow Graph

- Decision point – where control can diverge
- Process block – statements uninterrupted by decisions or junctions
- Junction point – where control flow merges



Outline of Control Flow Testing

- Complete Testing
    - Execute every path from entry to exit
        - Path Coverage
        - Also known as Path Testing
    - Execute every statement from entry to exit
        - Statement Coverage, also known as Statement Testing

    - Execute every condition from entry to exit
        - Branch Coverage, also known as Branch Testing
- 100% Path coverage may be impractical
    - Even small routines may have a large number of paths
- Statement and Branch Coverage are more commonly used
    - Insistence based on common sense
- Input to the test generation process
    - Source code
    - Path selection criteria: statement, branch, …
- Generation of control flow graph (CFG)
    - A CFG is a graphical representation of a program unit.
    - Compilers are modified to produce CFGs. (You can draw one by hand.)
- Selection of paths
    - Enough entry/exit paths are selected to satisfy path selection criteria.
- Generation of test input data
    - Two kinds of paths
        - Executable path: There exists input so that the path is executed.
        - Infeasible path: There is no input to execute the path.
    - Solve the path conditions to produce test input for each path.
- Every decision has a True and a False in its column
    - This implies branch coverage
    - Every decision is tested for T and F
- Is every edge executed at least once
    - Edges representing statement (blocks/segments)
    - Implies statement coverage
- Select enough number of paths
    - To get branch coverage
    - To get statement coverage

Path Selection Criteria

Program paths are selectively executed.

Question: What paths do I select for testing?

The concept of path *selection criteria* is used to answer the question.

Advantages of selecting paths based on defined criteria:

- Ensure that all program constructs are executed at least once.
- Repeated selection of the same path is avoided.
- One can easily identify what features have been tested and what not.

Path selection criteria

- Select all paths.
- Select paths to achieve complete statement coverage.
- Select paths to achieve complete branch coverage.
- Select paths to achieve predicate coverage.

All-path coverage criterion: Select all the paths in the program unit under consideration.

Statement coverage criterion

- Statement coverage criterion
    - Statement coverage means executing individual program statements and observing the output.

- 100% statement coverage means all the statements have been executed at least once
    - Cover all assignment statements.
    - Cover all conditional statements.
- Less than 100% statement coverage is unacceptable.

Branch coverage criterion

- A branch is an outgoing branch (edge) from a node in a CFG.
    - A condition node has two outgoing branches – corresponding to the True and False value of the condition.
- Covering a branch means executing a path that contains the branch.
- 100% branch coverage means selecting a set of paths such that each branch is included on some path.

Predicate coverage criterion

- If all possible combinations of truth values of the conditions affecting a path have been explored under some tests, then we say that predicate coverage has been achieved.

Generating Test Input

Having identified a path, a key question is how to make the path execute, if possible.

- Generate input data that satisfy all the conditions on the path.

Key concepts in generating test input:

- **Input vector**
    - An input vector is a collection of all data entities read by the routine whose values must be fixed prior to entering the routine.
    - Members of an input vector can be as follows.
        - Input arguments to the routine
        - Global variables and constants
        - Files
        - Contents of registers (Assembly)
        - Network connections
        - Timers
- **Predicate**
    - A predicate is a logical function evaluated at a decision point.
- **Path predicate**
    - A path predicate is the set of predicates associated with a path.
- **Predicate interpretation**
    - A path predicate may contain local variables.
    - Local variables play no role in selecting inputs that force a path to execute
    - Local variables can be eliminated by a process called **Symbolic execution.**
    - Predicate interpretation is defined as the process of
        - Symbolically substituting operations along a path in order to express the predicate solely in terms of the input vector and a constant vector.
    - A predicate may have different interpretations depending on how control reaches the predicate.
- **Path predicate expression**
    - An interpreted path predicate is called a path predicate expression.
    - A path predicate expression has the following attributes.

        - It is void of local variables.
        - It is a set of constraints in terms of the input vector, and, maybe, constants.
        - Path forcing inputs can be generated by solving the constraints.
        - If a path predicate expression has no solution, the path is infeasible.
    - An example of infeasible path
- **Generating input data from a path predicate expression**
    - Consider the path predicate expression of Figure 4.13 (reproduced below.)

```
0 < AS            ≡ True    ..... (1)
value[0] != -999  ≡ True    ..... (2)
value[0] >= MIN   ≡ True    ..... (3)
value[0] <= MAX   ≡ True    ..... (4)
1 < AS            ≡ False   ..... (5)
1 > 0             ≡ True    ..... (6)
```

    - One can solve the above equations to obtain following test input data

AS	= 1
MIN	= 25
MAX	= 35
Value[0]	= 30

    - Note: the set is not unique →
- A program unit may contain a large number of paths.
    - Path selection becomes a problem. Some selected paths may be infeasible.
    - Apply a path selection strategy:
        - Select as many short paths as possible.
        - Choose longer paths.
    - The are efforts to write code with fewer/no infeasible paths.

Outline of Control Flow Testing

Effectiveness

- Control flow testing is effective in unstructured programs
- Unit testing is dominated by control flow testing
- Evidence shows
    - Control flow testing catches 50% of all bugs caught in unit testing
    - That is about 33% of all bugs

Control Flow Testing is dominated by

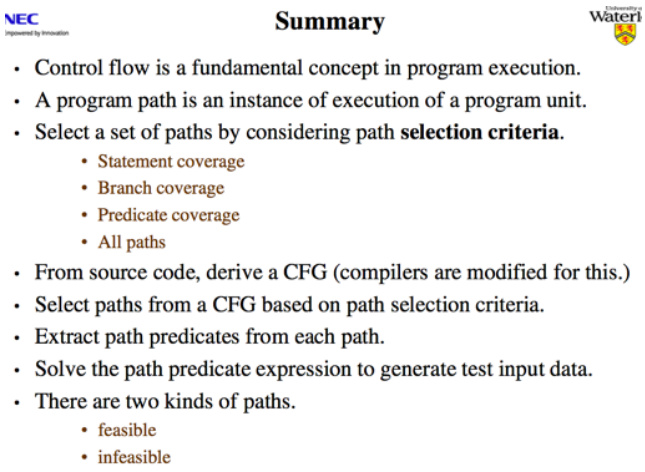
- Statement Testing/Coverage
- Branch Testing/Coverage

Limitations

- Not all interface errors are caught
- Not all initialization mistakes are caught
- Not all specification errors are caught

That is because Control Flow Testing assumptions

- We have correct specifications
- Definitions of data are correct
- Data can be accessed correctly
- All known bugs have been resolved
    - Except for the one that affect control flow



Summary

- Control flow is a fundamental concept in program execution.
- A program path is an instance of execution of a program unit.
- Select a set of paths by considering path **selection criteria**.
    - Statement coverage
    - Branch coverage
    - Predicate coverage
    - All paths
- From source code, derive a CFG (compilers are modified for this.)
- Select paths from a CFG based on path selection criteria.
- Extract path predicates from each path.
- Solve the path predicate expression to generate test input data.
- There are two kinds of paths.
    - feasible
    - infeasible

Chapter 5 – Data Flow Testing

The General Idea

A program unit accepts input, performs computations, assigns new values to values to variables, and returns results.

One can visualize of "flow" of data values from one statement to another.

A data value produced in one statement is expected to be used later.

- Example: Obtain a file pointer ….. use it later.
- If the later use is never verified, we do not know if the earlier assignment is acceptable.

Two motivations of data flow testing

- The memory location for a variable is accessed in a "desirable" way.
- Verify the correctness of data values "defined" (i.e. generated) – observe that all the "uses" of the value produce the desired results.

Idea: A programmer can perform a number of tests on data values.

- These tests are collectively known as data flow testing.

Data flow testing can be performed at to conceptual levels.

- Static data flow testing
- Dynamic data flow testing

Static data flow testing

- Identify potential defects, commonly known as **data flow anomaly**
- Analyze source code
- Do not execute code.

Dynamic data flow testing

- Involves actual program execution.
- Bears similarity with control flow testing.
    - Identify paths to execute them.
    - Paths are identified based on **data flow testing criteria.**

Data Flow Anomaly

Anomaly: It us an abnormal way of doing something.

- Example 1: The second definition of x overrides the first.
    - X = f1(y);
    - X = f2(z);

Three types of abnormal situations with using variable

- **Type 1:** Defined and then defined again
    - Four interpretations of *Example 1*
        - The first statement is redundant.
        - The first statement has a fault – the intended one one might be: w = f1(y).
        - The second statement has a fault – the intended one might be: v = f2(z).
        - There is a missing statement between the two: v = f3(x).
    - Note: It is for the programmer to make the desired interpretation.
- **Type 2**: Undefined but referenced
    - Example: x = x – y –w; /* w has not been defined by the programmer */
    - Two interpretations

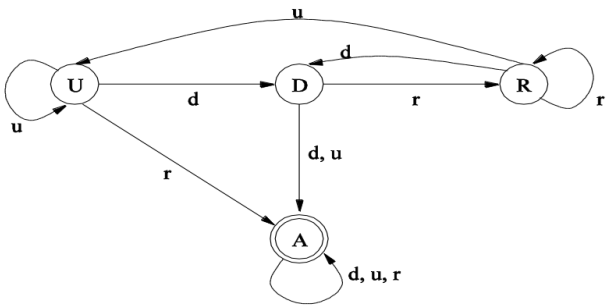
        - The programmer made a mistake in using **w**.
        - The programmer wants to use the compiler assigned value of **w**
- **Type 3:** Defined but not referenced
    - Example: Consider x = f (x, y). If x is not used subsequently, we have a Type 3 anomaly.

The concept of **a state-transition diagram** is used to **model a program variable** to identify data flow anomaly.

Components of the state-transition diagrams

- The states
    - U: Undefined
    - D: Defined but not referenced
    - R: Defined and referenced
    - A: Abnormal
- The actions
    - *d*: define the variable
    - *r*: reference (or, read) the variable
    - *u*: undefined the variable

Data Flow Anomaly



Legends:

States	Actions
U: Undefined	d: Define
D: Defined but not referenced	r: Reference
R: Defined and referenced	u: Undefine
A: Abnormal	

Obvious question: What is the relationship between the **Type 1, Type 2,** and **Type 3** anomalies and figure 5.2?

The three types of anomalies (Type 1, Type 2, and Type 3) are found in the diagram in the form of **action sequences:**

- Type 1: *dd*
- Type 2: *ur*
- Type 3: *du*

Detection of data flow anomaly via program instrumentation

- Program instrumentation: Insert new code to monitor the states of variables.
- If the state sequence contains *dd, ur,* or *du* sequence, a data flow anomaly is said to occur.

Bottom line: What to do after detecting a data flow anomaly?

- Investigate the cause of the anomaly.
- To fix an anomaly, write new code or modify the existing code.