Grand Valley State University
www.gvsu.edu

Software Engineering Verification (and Validation) ((again))

Erik Fredericks // frederer@gvsu.edu

Adapted from materials provided by Byron DeVries, Jagadeesh Nandigam

Outline

Software Testing
- Testing Techniques

Static Verification
- Code Reviews
- Static Analyzers
- Correctness Proofs

Debugging Techniques

Testing techniques

Black-Box Testing:

- Equivalence Partitioning
- Boundary Value Analysis

White-Box Testing:

- Coverage Based Testing
- Basis Path Testing

Blackbox testing

Also known as ***functional*** or ***behavioral*** testing.

Test cases are design based on the functional specifications of the program.

Tends to be applied during the **later stages** of testing (integration, validation, and systems testing).

Attempts to find the following types of errors:

- Interface errors
- Incorrect or missing functions
- Behavior or performance errors

Equivalence partitioning

A black-box testing technique that **divides the input domain into classes of data** called equivalence classes from which test classes can be derived.

Testing a program with **one value** from an equivalence class is (intended to be) **equivalent** to testing with **any other value** from that equivalence class.

Equivalence partitioning strives to define a test case that **uncovers classes of errors**, thereby **reducing the total number of test cases** that must be developed.

Equivalence classes are identified from evaluation of input conditions.

An equivalence class represents a set of valid or invalid states for input conditions.

EQUIVALENCE CLASS PARTITIONING (ECP)

AGE Enter Age *Accepts value 18 to 56

EQUIVALENCE PARTITIONING		
Invalid	**Valid**	**Invalid**
<=17	18-56	>=57

MOBILE NUMBER Enter Mobile No. *Must be 10 digits

EQUIVALENCE PARTITIONING		
Invalid	**Valid**	**Invalid**
987654321	9876543210	98765432109

Equivalence partitioning example

Example:

```
int factorial(int n);
```

Input: `n`
Output `n!`
Input condition: `0 <= n <= 25`

Application of equivalence partitioning technique:

- Equivalence class 1: `n < 0`
- Equivalence class 2: `0 <= n <= 25`
- Equivalence class 3: `n > 25`

Boundary value analysis

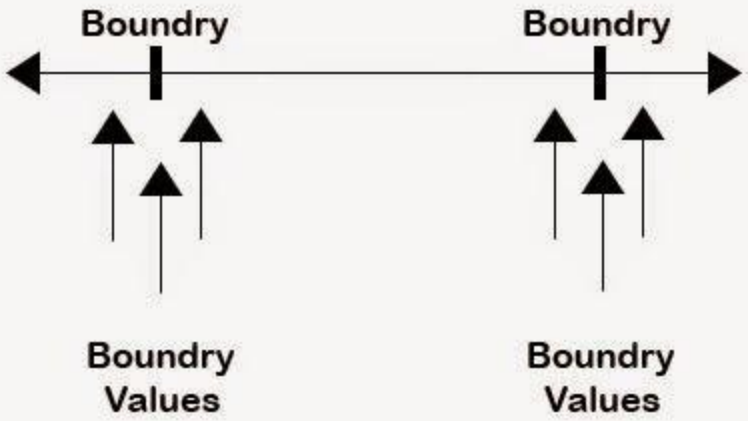
Boundary value analysis (BVA) is a **black-box testing technique** that complements equivalence partitioning.

Rather than selecting any element of an equivalence class, BVA leads to the selection of test cases at the “edges or boundaries” of an equivalence class.

Guidelines for boundary value analysis:

- If an input condition specifies a **range** bounded by values **a** and **b**, test cases should be designed with values **a** and **b** as well as ***just above*** and ***just below*** **a** and **b**.
- If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers.
    - Values just above and just below minimum and maximum are also tested.
- Most software engineers or testers intuitively perform BVA to some degree.

Boundary Value Analysis



www.precisetestingsolution.com

Do we need to worry about the "in between" values?

White-box testing

Also known as structural or glass-box testing.

Test cases are designed by examining the internal structure of the program.

Applied during early stages of testing (unit testing).

White-box testing techniques:

- Coverage Based Testing
- Basis Path Testing
- Data-Flow Testing
- Loop Testing
- Condition Testing

Coverage-based testing

Design enough test cases such that:

- **Line coverage**:
  - Every line in a program is executed at least once.

- **Branch coverage**:
  - Each decision/branch has a true and false outcome at least once.

- **Condition coverage**:
  - Each condition in a decision/branch takes all possible outcomes at least once.

- **Branch/Condition coverage**:
  - Each condition in a decision takes on all possible outcomes at least once, and each decision takes on all possible outcomes at least once.

Coverage-based testing

Design enough test cases such that:

- **Multiple condition coverage**:
  - All possible combinations of condition outcomes in each decision are invoked at least once.

- **Path coverage**:
  - All execution paths are executed at least once:
  - A path is a unique sequence of branches from entry to exit.
  - Loops introduce an unbounded number of paths making path coverage criterion not practical.
  - Some paths are impossible to exercise due to relationships of data.
  - The number of executions of a loop may depend on the input variables and hence may not be possible to determine.

Basis path testing

A white-box testing technique first proposed by Tom McCabe.

Derive a logical complexity measure of a procedural design.

Use **cyclomatic complexity** to identify execution paths and design test cases to exercise these paths.

Steps in basis path testing:

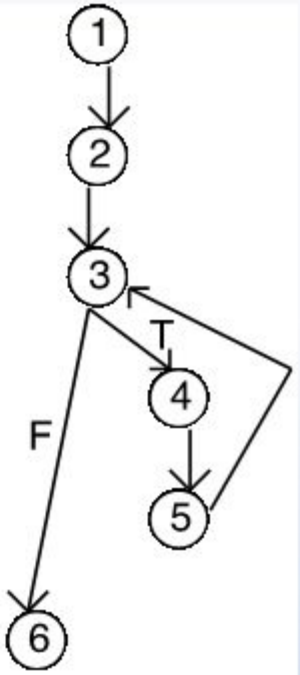
1. Use the design or code as a foundation to draw a corresponding flow graph.
2. Determine the cyclomatic complexity of the resulting flow graph.
3. Determine a basis set of linearly independent paths.
4. Prepare test cases that will force execution of each path in the basis set.

Refresher! What is the difference between branch coverage and condition coverage?

Basis path testing

Step 1: Use the design or code as a foundation to draw a corresponding flow graph.

```
  int sum_numbers(int n) {
      int i, sum;
1 →   i = 1;
2 →   sum = 0;
3 →   while (i <= n) {
4 ——→   sum = sum + i;
5 ——→   i = i + 1;
      }
6 →   return sum;
  }
```



Basis path testing

Step 2: Determine the **cyclomatic complexity** of the resulting flow graph.

- Cyclomatic complexity is a software metric that provides a quantitative measure of the logical complexity of a program.

- When used in the context of the basis path testing, the value computed for cyclomatic complexity defines the **number of independent paths** through a program.

- It provides an upper bound on the number of test cases that must be conducted to ensure all statements have been executed at least once.

- **Three ways of computing cyclomatic complexity:**
  - Enclosed Regions Method
  - Edge-Node Method
  - Predicate-Node Method

Enclosed Regions Method:

The cyclomatic complexity, `V(G)`, for a flow graph `G` is defined as

$$V(G) = ER+1$$

where `ER` is the number of enclosed regions in the flow graph. **Areas bounded by edges and nodes are called enclosed regions.**

Edge-Node Method:

The cyclomatic complexity, `V(G)`, for a flow graph `G` is defined as

$$V(G) = E - N + 2$$

where `E` is the number of edges and `N` is the number of nodes in the flow diagram.

Predicate-Node Method:

The cyclomatic complexity, `V(G)`, for a flow graph `G` is defined as

$$V(G)=P+1$$

where P is the number of predicate nodes in the flow graph.

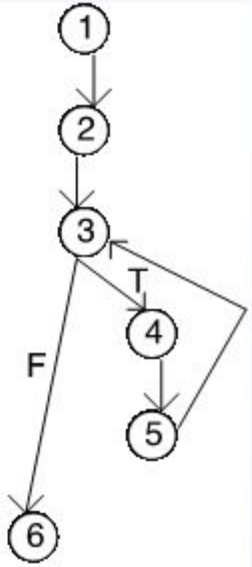
A predicate node is a **node with two or more edges *emanating*** from it.

Basis path testing

Cyclomatic complexity of `sum_numbers` function:

```
[Enclosed region] V(G) = ER + 1       = 1 + 1          = 2
[Edge node]       V(G) = E - N + 2    = 6 - 6 + 2      = 2
[Predicate node]  V(G) = P + 1        = 1 + 1          = 2
```

```
   int sum_numbers(int n) {
      int i, sum;
1 →   i = 1;
2 →   sum = 0;
3 →   while (i <= n) {
4 ——→    sum = sum + i;
5 ——→    i = i + 1;
      }
6 →   return sum;
   }
```



Basis path testing

Step 3: Determine a basis set of linearly independent paths.

- An independent path must move along at least one edge that has not been traversed before the path is defined.
- **The set of independent paths is called its basis set**
- Independent path for **`sum_numbers`** function:

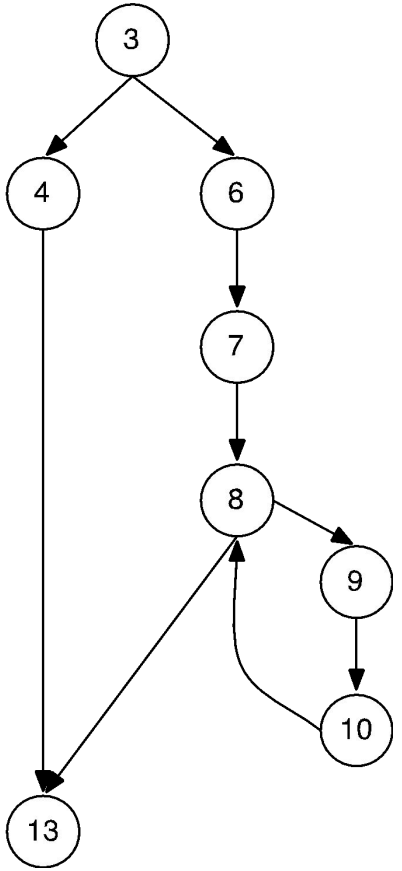
Step 4: Prepare test cases that will force execution of each path in the basis set.

- Path 1: Input: n <= 0
  Expected result: sum = 0
- Path 2: Input n >= 1
  Expected result: sum of 1 to n

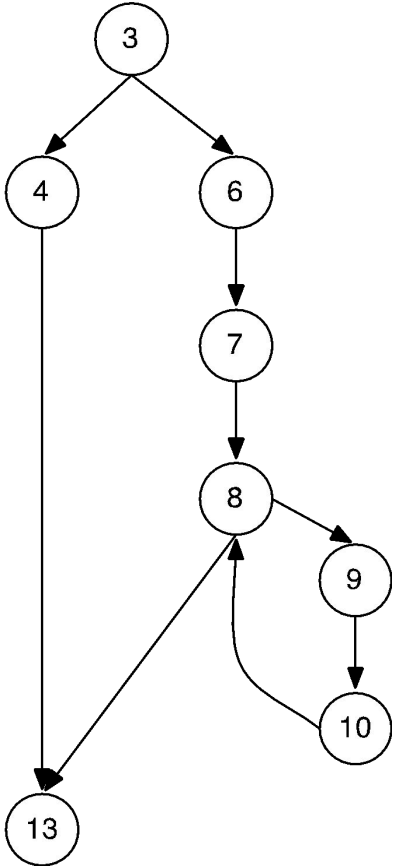
In-Class Exercise

```
1.  int factorial(int n) {
2.      int i, fact;
3.      if(n < 0) {
4.          fact = 0;
5.      } else {
6.          i = 1;
7.          fact = 1;
8.          while(i <= n) {
9.              fact = fact * i;
10.             i = i + 1;
11.         }
12.     }
13.    return fact;
14. }
```

```
 1.  int factorial(int n) {
 2.      int i, fact;
 3.      if(n < 0) {
 4.          fact = 0;
 5.      } else {
 6.          i = 1;
 7.          fact = 1;
 8.          while(i <= n) {
 9.              fact = fact * i;
10.              i = i + 1;
11.          }
12.      }
13.     return fact;
14.  }
```



[Enclosed region] V(G) = ER + 1 = 2 + 1 = 3
[Edge note] V(G) = E – N + 2 = 9 - 8 + 2 = 3
[Predicate node] V(G) = P + 1 = 2 + 1 = 3



A few updates:

To get more than 2 edges out of a node: `switch` statement
(if - elseif are just additional 2-edge predicate nodes)

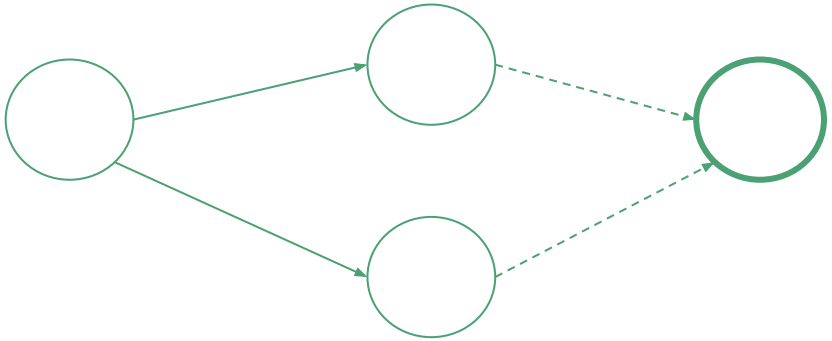
If higher order, then add 1 for each extra edge
[Predicate node] V(G) = P + ***n***

if ➜ n = 1
switch (3 cases) ➜ n = 2
switch (4 cases) ➜ n = 3
...

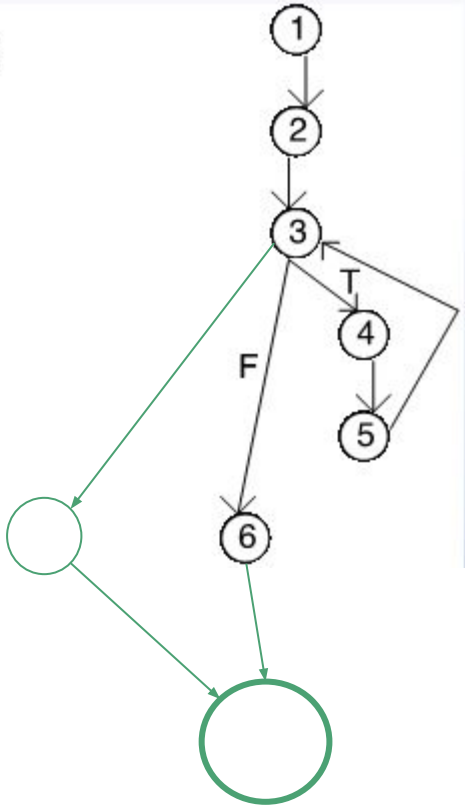
A few updates cont'd:

Must follow structured programming principles
i.e., one entrance and one exit into function

So, all returns must coalesce to a single (possibly artificial) return



```
    int sum_numbers(int n) {
        int i, sum;
1 ->    i = 1;
2 ->    sum = 0;
3 ->    while (i <= n) {
4 ---->     sum = sum + i;
5 ---->     i = i + 1;
        }
6 ->    return sum;
    }
```



Static V&V techniques

Software Reviews – reading a software artifact

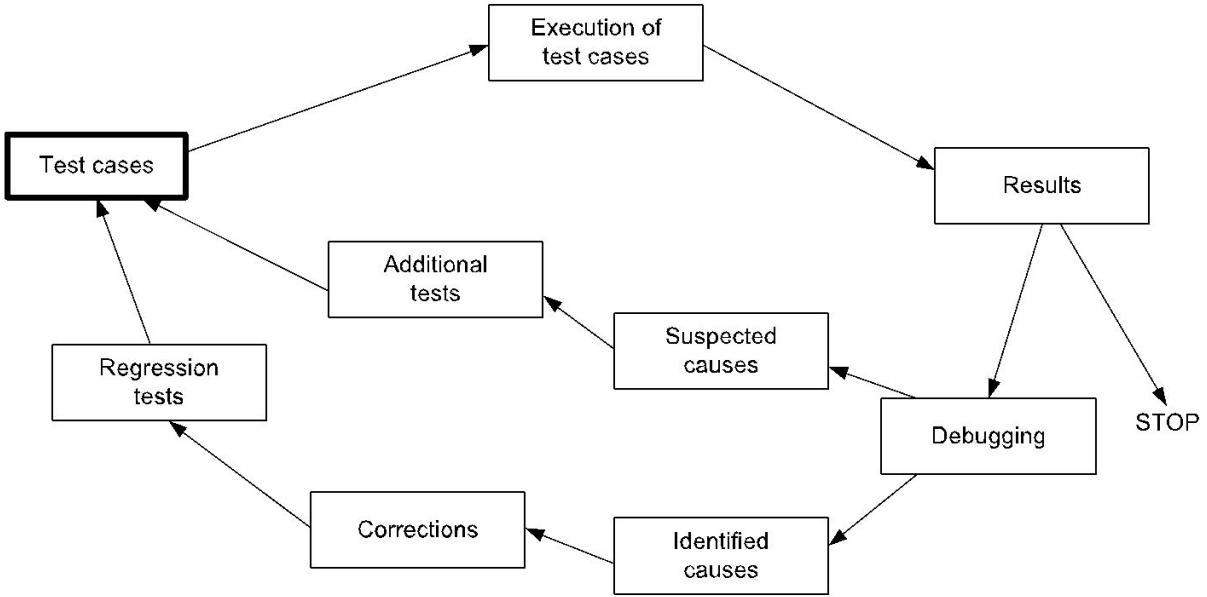
- Walkthroughs: two-step informal process
  - Preparation
  - Analysis
- Inspections: five-step formal process
  - Overview
  - Preparation
  - Inspection
  - Rework
  - Follow-up
- Desk Checking: one-person walkthrough or inspection

Correctness Proofs:

- Mathematical techniques for showing that a product is correct and satisfies its specifications

Debugging techniques

Debugging **always occurs** as a consequence of testing.
Debugging process **begins** with the execution of a test case.



Debugging techniques

The debugging process will always have one of two outcomes

- The cause will be found and corrected, or
- The cause will not be found in which case the person debugging may suspect a cause, design one or more test cases to help validate that suspicion, and work toward error correction in an iterative fashion.

Debugging Techniques

- Brute Force
- Backtracking
- Cause elimination
- Automated debugging tools
- People factor – a fresh viewpoint, unclouded by hours of frustration, from a colleague can do wonders

Debugging techniques

Brute Force:

- Most common and least efficient method for isolating the cause of an error
- Hope to find a clue that can lead us to the cause of an error
- Using memory dumps, run-time traces, loading program with output statements.

Backtracking:

- A fairly common debugging approach that can be used successfully in small programs
- Beginning at the site where a symptom has been uncovered, the source code is traced backward (manually) until the site of the cause is found.
- As a number of source lines increase, the number of potential backwards paths may become unmanageably large.

Debugging techniques

Cause Elimination:

- A “cause hypothesis” is devised and data is used to prove or disprove the hypothesis.
- Alternatively, a list of all possible causes is developed, and tests are conducted to eliminate each cause in an attempt to isolate the bug

Interesting Quotes on Debugging:

- “The first step in fixing a broken program is getting it to fail repeatedly (on the simplest example possible).” – T. Duff

- “Everyone knows that debugging is twice as hard as writing a program in the first place. So if you are as clever as you can be when you write it, how will you ever debug it?” – Brian Kernighan