Rishabh Indoria
21F3001823

Software Testing

October 14, 2024

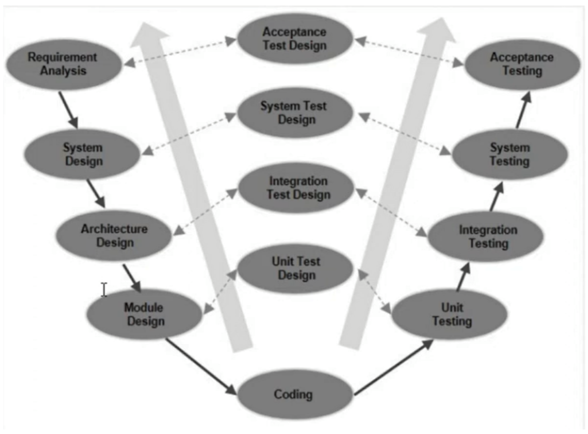
1 Introduction

1.1 Motivation

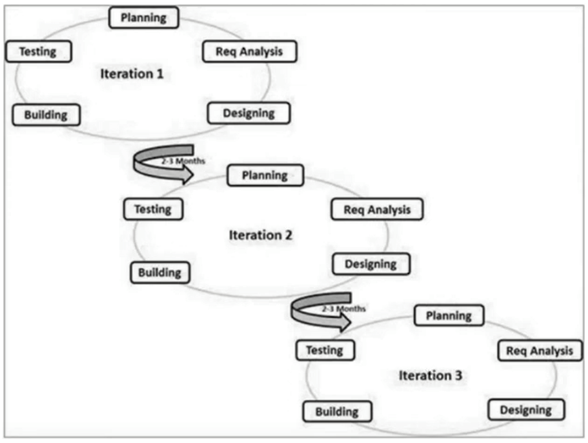
- **Introduction to Software Testing** by Paul Ammann and Jeff Offutt, **The Art of Software Testing** by Glenford J. Myers, **Software Testing: A Craftsman's Approach** by Paul C. Jorgensen, **Agile Testing: A practical guide for Testers and Agile Teams** by Lisa Crispin and Janet Gregory.
- Software is ubiquitous; Such software should be of very high quality, offer good performance in terms of response time, performance and also have no errors.
- It is no longer feasible to shut down a malfunctioning system in order to restore safety.
- Errors in software can cost lives, huge financial losses, or simply a lot of irritation.
- Testing is the **predominantly used** technique to find and eliminate errors in software.

1.2 Software Development Life Cycle

- **SDLC**: term used by the software industry to define a process for designing, developing, testing and maintaining a high quality software product.
- The goal is to use SDLC defined processes to develop a high quality software product that meets customer demands.
- **Planning**: Includes clearly identifying customer and/or market needs, pursuing a feasibility study and arriving at an initial set of requirements.
- **Requirements definition**: Includes documenting detailed requirements of various kinds: System-level, functional, software, hardware, quality requirements etc. They get approved by appropriate stakeholders.
- **Requirements analysis**: Includes checking and analyzing requirements to ensure that they are consistent, complete and match the feasibility study and market needs.
- **Design**: Identifies all the modules of the software product, details out the internals of each module, the implementation details and a skeleton of the testing details.
- **Architecture**: Defines the modules, their connections and other dependencies, the hardware, database and its access etc.
- **Development**: The design documents, especially that of low-level design, is used to implement the product. There are usually coding guidelines to be followed by the developers. Extensive unit testing and debugging are also done, usually by the developers. Tracking is done by project management team.
- **Testing**: Involves testing only where the product is thoroughly tested, defects are reported, fixed and re-tested, until all the functional and quality requirements are met.
- **Maintenance**: Done post deployment of product. Add new features as desired by the customer/market. Fix errors, if any, in the software product. Test cases from earlier phases are re-used here, based on need.
- **V-model**: It is a model that focuses on verification and validation. Follows the traditional SDLC life-cycle: Requirements, Design, Implementation, Testing, Maintenance.
- **Agile model**: Agile methodologies are adaptive and focus on fast delivery of features of a software product. All the SDLC steps are repeated in incremental iterations to deliver a set of features. Extensive customer interactions, quick delivery and rapid response to change in requirements.
- **Other Activities**: Project management, includes team management. Project documentation(Traceability matrix is a document that links each artifacts of development phase to those of other phases). Quality Inspection.



(a) V-Model



(b) Agile Model

Figure 1: Model Visualization

1.3 Testing Terminologies

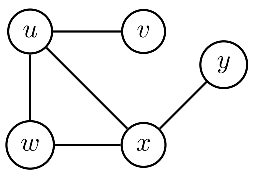
- **Validation**: The process of evaluating software at the end of software development to ensure compliance with intended usage. i.e., checking if the software meets its requirements.
- **Verification**: The process of determining whether the products of a given phase of the software development process fulfill the requirements established at the start of that phase.
- **Fault**: A static defect in the software. It could be a missing function or a wrong function in code.
- **Failure**: An external, incorrect behavior with respect to the requirements or other description of the expected behavior. A failure is a manifestation of a fault when software is executed.
- **Error**: An incorrect internal state that is the manifestation of some fault.
- **Test case**: A test case typically involves inputs to the software and expected outputs. A failed test case indicates an error. A test case also contains other parameters like test case ID, traceability details etc.
- **Unit Testing**: Done by developer during coding.
- **Integration Testing**: Various components are put together and tested. Components could be only software or software and hardware components.
- **System Testing**: Done with full system implementation and the platform on which the system will be running.
- **Acceptance Testing**: Done by end customer to ensure that the delivered products meet the committed requirements.
- **Beta Testing**: Done in a (so-called) beta version of the software by end users, after release.
- **Functional Testing**: Done to ensure that the software meets its specified functionality.
- **Stress Testing**: Done to evaluate how the system behaves under peak/unfavorable conditions.
- **Performance Testing**: Done to ensure the speed and response time of the system.
- **Usability Testing**: Done to evaluate the user interface, aesthetics.
- **Regression Testing**: Done after modifying/upgrading a component, to ensure that the modification is working correctly, and other components are not damaged by the modification.
- **Black-Box Testing**: A method of testing that examines the functionalities of a software/system without looking into its internal design or code.
- **White-Box Testing**: A method of testing that test the internal structure of the design or code of a software.
- **Test Design**: Most critical job in testing. Need to design effective test cases. Apart from specifying the inputs, this involves defining the expected outputs too. Typically, cannot be automated.

- **Test Automation**: Involves converting the test cases into executable scripts. Need to specify how to reach deep parts of the code using just inputs, Observability and Controllability.
- **Test Execution**: Involves running the test on the software and recording the results. Can be fully automated.
- **Test Evaluation**: Involves evaluating the results of testing, reporting identified errors. A difficult problem is to isolate faults, especially in large software and during integration testing.
- **Testing goals**: Organizations tend to work with one or more of the following levels
  Level 0: There is no difference between testing and debugging.
  Level 1: The purpose of testing is to show correctness.
  Level 2: The purpose of testing is to show that software doesn't work.
  Level 3: The purpose of testing is not to prove anything specific, but to reduce the risk of using the software.
  Level 4: Testing is a mental discipline that helps all IT professionals develop higher quality software.
- **Controllability**: Controllability is about how easy it is to provide inputs to the software module under test, in terms of reaching the module and running the test cases on the module under test.
- **Observability**: Observability is about how easy it is to observe the software module under test and check if the module behaves as expected.

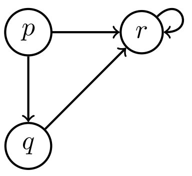
2 Graphs

2.1 Basics

- A graph is a tuple $G = (V, E)$ where V is a set of **nodes/vertices**, and $E \subseteq (V \times V)$ is a set of **edges**.
- Graphs can be **directed** or **undirected**.



(a) A simple undirected graph



(b) A directed graph

- Graphs can finite or infinite.
- The **degree** of a vertex is the number of edges that are connected to it. Edges connected to a vertex are said to be **incident** on the vertex.
- There are designated special vertices like **initial** and **final** vertices. These vertices indicate beginning and end of a property that the graph is modeling.
- Typically, there is only one initial vertex, but there could be several final vertices.

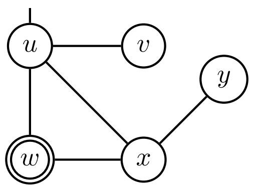
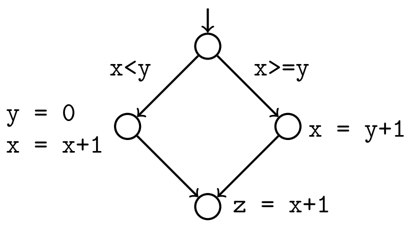


Figure 3: Graph with initial and final vertices

- Most of these graphs will have **labels** associated with vertices and edges. Labels or annotations could be details about the artifact that the graphs are modelling. Tests are intended to cover the graph in some way.

```
if(x<y){
y = 0;
x = x+1;
}else{
x = y+1;
}
z = x+1;
```

(a) A sample code



(b) A Control Flow Graph

Figure 4: Example of a CFG

- A **path** is a sequence of vertices $v_1, v_2, ..., v_n$ such that $(v_i, v_{i+1}) \in E$.
- **Length** of a path is the number of edges that occur in it. A single vertex path has length 0.
- **Sub-path** of a path is a sub-sequence of vertices that occur in the path.
- A vertex v is **reachable** from some other vertex if there is a path connecting them.
- An edge $e = (u, v)$ is **reachable** if there is a path that goes to vertex u and then goes to vertex v.
- A **test path** is a path that starts in an initial vertex and ends in a final vertex. These represent execution of test cases.
- Some test paths can be executed by many test cases: **Feasible paths**.
- Some test paths cannot be executed by any test case: **Infeasible paths**.
- A test path p **visits** a vertex v if v occurs in path p. A test path p **visits** an edge e if e occurs in the path p.
- A test path p **tours** a path q if q is a sub-path of p.
- When a test case t executes a path, we call it the **test path** executed by t, denoted by $path(t)$.
- The set of test paths executed by a set of test cases T is denoted by $path(T)$.
- **Test requirement** describes properties of test paths.
- **Test Criterion** are rules that define test requirements.
- **Satisfaction**: Given a set TR of test requirements for a criterion C, a set of tests T satisfies C on a graph iff for every test requirement in $t \in TR$, there is a test path in $path(T)$ that meets the test requirement t.
- **Structural Coverage Criteria**: Defined on a graph just in terms of vertices and edges.
- **Data Flow Coverage Criteria**: Requires a graph to be annotated with references to variables and defines criteria requirements based on the annotations.

2.2 Elementary Graph Algorithms

- Two standard ways of representing graphs: **adjacency matrix** or **adjacency lists**.
- Adjacency list representation provides a compact way to represent **sparse** graphs, i.e., graphs for which $|E|$ is much less than $|V|^2$. For each $u \in V$, $Adj[u]$ contains all vertices v such that $(u, v) \in E$, i.e., it contains all edges incident with u. Represented in $\Theta(|V| + |E|)$ memory.
- Adjacency matrix representation provides a compact way to represent **dense** graphs, i.e., graphs for which $|E|$ is close to $|V|^2$. This is a $|V| \times |V|$ matrix, where $a_{ij} = 1$ if there is an edge going from i to j.
- **Breadth First Search**: Computes the "distance" from s to each reachable vertex.

Algorithm 1 Breadth First Search

```
    BFS($G$)
 1: for each vertex $u \in G, V - \{s\}$ do
 2:     $u.color = WHITE$, $u.d = \infty$, $u.\pi =$ NIL
 3: end for
 4: $s.color = BLUE$, $s.d = 0$, $s.\pi =$ NIL
 5: $Q = \phi$
 6: Enqueue($Q, s$)
 7: while $Q \neq \phi$ do
 8:     $u =$ Dequeue($Q$)
 9:     for each $v \in G.Adj[u]$ do
10:         if $v.color == WHITE$ then
11:             $v.color = BLUE$
12:             $v.d = u.d + 1$
13:             $v.\pi = u$
14:             Enqueue($Q, v$)
15:         end if
16:     end for
17:     $u.color = BLACK$
18: end while
```

- **Depth First Search**

Algorithm 2 Depth First Search

```
    DFS($G$)
 1: for each vertex $u \in G.V$ do
 2:     $u.color = WHITE$
 3:     $u.\pi = NIL$
 4: end for
 5: $time = 0$
 6: for each vertex $u \in G.V$ do
 7:     if $u.color == WHITE$ then
 8:         DFS-Visit($G, u$)
 9:     end if
10: end for

    DFS-Visit($G, u$)
11: $time = time + 1$
12: $u.d = time$
13: $u.color = GRAY$
14: for each $v \in G.Adj[u]$ do
15:     if $v.color == WHITE$ then
16:         $v.\pi = u$
17:         DFS-Visit($G, v$)
18:     end if
19: end for
20: $u.color = BLACK$
21: $time = time + 1$
22: $u.f = time$
```

2.3 Structural Graph Coverage

- **Node Coverage** requires that the test cases visit each node in the graph once. Test set T satisfies node coverage on graph G iff for every syntactically reachable node $n \in G$, there is some path p in $path(T)$ such that p visits n.
- **Edge Coverage**: TR contains each reachable path of length up to 1, inclusive, in G. Edge coverage is slightly stronger than node coverage. Allowing length up to 1 allows edge coverage to subsume node coverage.
- **Edge-Pair Coverage**: TR contains each reachable path of length up to 2, inclusive in G. Paths of length up to 2 correspond to pairs of edges.

- **Complete path coverage**: TR contains all paths in G. Unfortunately, this can be an infeasible test requirement, due to loops.
- **Specified path coverage**: TR contains a set S of paths, where S is specified by the user/tester.
- A path from n_i to n_j is **simple** if no node appears more than once, except possible the first and last node.
- A **prime path** is a simple path that does not appear as a proper sub-path of any other simple path.
- **Prime path coverage**: TR contains each prime path in G. Ensures that loops are skipped as well as executed. It subsumes node and edge coverage.
- **Tour with side trips**: A test path p tours a sub-path q with side trips iff every edge q is also in p in the same order. the tour can include a side trip, as long as it comes back to the same node.
- **Tours with detours**: A test path p tours a sub-path q with detours iff every node in q is also in p in the same order. The tour can include a detour from node n as long as it comes back to the prime path at a successor of n.
- **Best Effort Touring**: Satisfy as many test requirements as possible without sidetrips. Allow sidetrips to try to satisfy remaining test requirements.
- **Round trip path**: A prime path that starts and ends at the same node.
- **Simple round trip coverage**: TR contains at least one round trip path for each reachable node in G that begins and ends in a round trip path.
- **Complete round trip coverage**: TR contains all round trip paths for each reachable node in G.

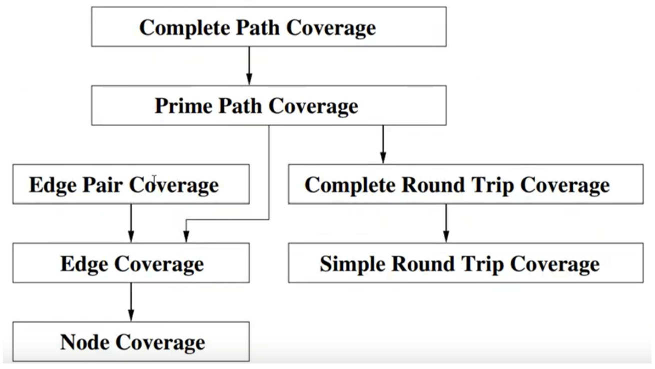


Figure 5: Structure Coverage Criteria Subsumption

2.4 Algorithms: Structural Graph Coverage Criteria

- There are two entities related to coverage criteria: Test requirement, and Test case as a test path, if the test requirement is feasible.
- Test requirements for node and edge coverage are already given as a part of the graph modeling the software artifact. Test paths to achieve node and edge coverage can be obtained by simple modifications to BFS.
- TR for edge-pair coverage is all paths of length two in a given graph. Need to include paths that involve self-loops too.

Algorithm 3 Simple Edge-Pair Algorithm

```
1: for each node $u$ in the graph do
2:     for each node $v$ in $Adj[u]$ do
3:         for each node $w$ in $Adj[v]$ do
4:             Output path $u--v--w$
5:         end for
6:     end for
7: end for
```

- Prime Path Algorithm

Algorithm 4 Computing prime paths

```
$Loops$ = [ ]
$Terminate$ = [ ]
$Q = \phi$
**for** each $v \in G$ **do**
    ENQUEUE($Q$, [$v$])
**end for**
**while** $Q \neq \phi$ **do**
    $path$ = DEQUEUE($Q$)
    **for** each $v \in G.Adj[path[-1]]$ **do**
        **if** $v$ is a final vertice **then**
            $Terminate = Terminate$ ++ ($path$ ++ $v$)
        **else if** $v$ in $path$ and $v == path[0]$ **then**
            $Loops = Loops$ ++ ($path$ ++ $v$)
        **else if** $v$ in $path$ **then**
            continue
        **else**
            ENQUEUE($Q$, $path$ ++ $v$)
        **end if**
    **end for**
**end while**
**return** $Loops$, $Terminate$
```

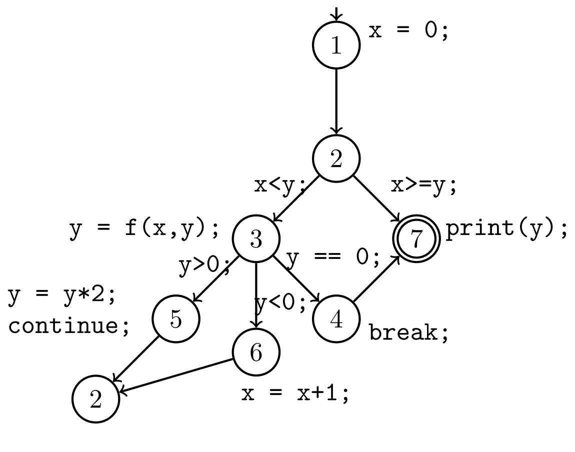
- To enumerate test paths for prime path coverage: Start with the longest prime paths and extend each of them to the initial and the final nodes in the graph.

2.5 Control Flow Graphs for Code

- Modelling control flow in code as graphs. Using structural coverage criteria to test control flow in code.
- Typically used to test a particular function or procedure or a method.
- A **Control Floe Graph** models all executions of a method by describing control structures.
- **Nodes**: Statements or sequences of statements (basic blocks).
- **Basic Block**: A sequence of statements such that if the first statement is executed, all statements will be (no branches).
- **Edges**: Transfer of control from one statement to the next.
- CFGs are often annotated with extra information to model data, this includes branch predicates, Definitions and/or uses.

```
x = 0;
while(x<y){
    y = f(x,y);
    if(y == 0){
        break;
    }else if(y<0){
        y = y*2;
        continue;
    }
    x = x+1;
}
print(y);
```

(a) While loop with break and continue



(b) A Control Flow Graph

Figure 6: Example of a CFG

2.6 Data Flow in Graphs

- Graph models of programs can be tested adequately by including values of variables(data values) as a part of the model.
- Data values are created at some point in the program and use later. They can be used several times.
- A **definition (def)** is a location where a value of a variable is stored into memory.
- A **use** is a location where a value of a variable is accessed.
- As a program executes, data values are carried from their defs to uses. We call these du-pairs or def-use pairs.
- A **du-pair** is a pair of location (l_i, l_j) such that a variable v is defined at l_i and used at l_j.
- Let V be the set of variables that are associated with the program artifact being modelled as a graph.
  The subset of V that each node n (edge e) defines is called $def(n)(def(e))$.
  The subset of V that each node n (edge e) uses is called $use(n)(use(e))$.
- A def of a variable may or may not reach a particular use.
- A path from l_i to l_j is **def-clear** with respect to variable v if v is not given another value on any of the nodes or edges in the path.
- If there is a def-clear path from l_i to l_j with respect to v, the def of v at l_i **reaches** the use at l_j.
- A **du-path** with respect to a variable v is a simple path that is def-clear from a def of v to a use of v.
- $du(n_i, n_j, v)$: The set of du-paths from n_i to n_j for variable v.
- $du(n_i, v)$: The set of du-paths that start at n_i for variable v.
- In testing literature, there are two notions of uses available.
  If v is used in a computational or output statement, the use is referred to as **computation use** (or **c-use**).
  If v is used in a conditional statement, its use is called as **predicate use** (or **p-use**).
- Data flow coverage criteria will be defined as sets of du-paths. Such du-paths will be grouped to define the data flow coverage criteria.
- The **def-path set** $du(n_i, v)$ is the set of du-paths with respect to variable v that start at node n_i.
- A **def-pair set**, $du(n_i, n_j, v)$ is the set of du-paths with respect to variable v that start at node n_i and end at node n_j.
- It can be clearly seen that $du(n_i, v) = \bigcup_{n_j} du(n_i, n_j, v)$.
- A test path p is said to **du tour** a sub-path d with respect to v if p tours d and the portion of p to which d corresponds is def-clear with respect to v.
- We can allow **def-clear side trips** with respect to v while touring a du-path, if needed.
- There are three common data flow criteria
  1. TR: Each def reaches at least one use.
  2. TR: Each def reaches all possible uses.
  3. TR: Each def reaches all possible uses through all possible du-paths.
- We assume every use is preceded by a def, every def reaches at least one use, and for every node with multiple out-going edges, at least one variable is used on each out edge, and the same variables are used on each out edge.
- **Subsumption**: ③→②→①
- Prime path coverage subsumes all-du-paths coverage.

3 Integration Testing

3.1 Introduction

- Software design basically dictates how the software is organized into **modules**.
- Modules interact with each other using well-defined **interfaces**.
- **Integration testing** involves testing if the modules that have been put together as per design meet their functionalities and if the interfaces are correct.
- Begins after unit testing, each module has been unit tested.
- **Procedure call interface**: A procedure/method in one module calls a procedure/method in another module. Control can be passed in both directions.
- **Shared memory interface**: A block of memory is shared between two modules. Data is written to/read from the memory block by the modules. The memory block itself can be created by a third module.
- **Message-passing interface**: One module prepares a message of a particular type and send it to another module through this interface. Client-server systems and web-based systems use such interfaces.
- Empirical studies account for up to quarter of all the errors in a system to be interface errors.
- Integration testing need not wait until all the modules of a system are coded and unit tested.
- When testing incomplete portions of software, we need extra software components, sometimes called **scaffolding**.
- **Test stub** is a skeletal or special purpose implementation of a software module, used to develop or test a component that calls the stub or otherwise depends on it.
- **Test driver** is a software component or test tool that replaces a component that tales care of the control and/or the calling of a software component.
- There are five approaches to do integration testing: Incremental, Top-down, Bottom-up, Sandwich, and Big Bang.
- **Incremental approach**: Integration testing is conducted in an incremental manner. The complete system is built incrementally, cycle by cycle, until the entire system is operational. Each cycle is tested by integrating the corresponding modules, errors are fixed before the testing of next cycle begins.
- **Top-down approach to integration testing**: Works well for systems with hierarchical design.
- In hierarchical design, there is a first top-level module, which is decomposed into some second-level modules, some of which, are in turn, decomposed into third-level modules and so on. Terminal modules are those that are not decomposed and can occur at any level. Module hierarchy is the reference document.
- It could be the case that A and B are ready but C and D are not, so we can develop stubs for C and D for testing interface between A and B. We keep doing this level by level.
- **Bottom-up approach**: Now we basically start with the lowest level modules and write test drivers to test integration.
- Basic difference between top-down and bottom-up is that top-down uses only test stubs and bottom-up uses test drivers.
- **Sandwich** approach tests a system by using a mix of top-down and bottom-up testing.
- **Big Bang Approach**: All individually tested modules are put together to construct the entire system which is tested as a whole.

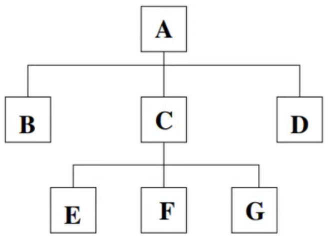


Figure 7: Module Hierarchy

3.2 Design Integration Testing

- Graph models for integration testing are called **Call graphs**.
- For graph models, nodes now become modules/test stubs/test drivers and edges become interfaces.
- **Structural coverage criteria**: Deals with calls over interfaces.
- **Data flow coverage criteria**: Deals with exchange of data over interfaces.
- Node coverage will be to call every module at least once.
- Edge coverage is to execute every call at least once.
- Specified path coverage can be used to test a sequence of method calls.
- Data flow interfaces among modules are more complicated than control flow interfaces.
- **Caller**: A module that invokes/calls another module.
- **Callee**: The module that is called.
- **Call site**: The statement or node where the call appears in the code.
- **Actual parameters**: Variables in the caller.
- **Formal parameters**: Variables in the callee.
- **Coupling variables** are variables that are defined in one unit and used in the other.
- **Parameter coupling**: Parameters are passed in calls.
- **Shared data coupling**: Two units access the same data through global or shared variables.
- **External device coupling**: Two units access an external object like a file.
- **Message-passing interfaces**: Two units communicate by sending and/or receiving messages over buffers/channels.
- Since focus is on testing interfaces, we consider the **last definitions** of variables before calls to and returns from the called units and the **first uses** inside the modules and after calls.
- **Last-def**: The set of nodes that define a variable x and has a def-clear path from the node through a call site to a use in the other module. Can be in either direction.
- **First-use**: The set of nodes that have uses of a variable y and for which there is a def-clear and use-clear path from the call site to the nodes.
- A **coupling du-path** is from a last-def to a first-use.
- **All-coupling-def coverage**: A path is to be executed from every last-def to at least one first-use.
- **All-coupling-use coverage**: A path is to be executed from every last-def to every first-use.
- **All-coupling-Du-paths coverage**: Every simple path from every last-def to every first-use needs to be executed.
- The above criteria can be met with side trips.
- Only variables that are used or defined in the callee are considered for du-pairs and criteria.
- **Transitive du-pairs** (A calls B, B calls C and there is a variable defined in A and used in C) is not supported in this analysis.

4 Specification Testing

4.1 Sequencing Constraints

- A **design specification** describes aspects of what behavior a software should exhibit. Behaviour exhibited by software need not mean the implementation directly. It could be a **model** of the implementation.
- For testing with graphs, we consider two types of design specifications. **Sequencing constraints** on methods/functions, and **State behavior** descriptions of software.
- **Sequencing constraints** are rules that impose constraints on the order in which methods may be called.
- Typically encoded as preconditions or other specifications. They may or may not be given as a part of the specification or design.
- Consider the example of a simple Queue, a precondition can be that at least one element must be on the queue before removing and a postcondition can be that e is on the end of the queue to enqueue e.
- Simple sequencing constraint: enqueue must be called before dequeue.
- This does not include the requirement that we must have at least as many enqueue calls as dequeue calls. Need *memory* which can be captured in the *state* of the queue as the application code executes.
- Absence of sequencing constraints usually indicates more faults.

4.2 Finite State Machines

- A **Finite State Machine** is a graph that describes how software variables are modified during execution.
- Nodes: **States**, representing sets of values for (key) variables.
- Edges: **Transitions**, which model possible changes from one state to another. Transitions have **guards** and/or **actions** associated with them.
- FSMs can model many kinds of systems like embedded software, abstract data types, hardware circuits etc.
- Creating FSM models for design helps in precise modelling and early detection of errors through analysis of the model.
- Many modelling notations support FSMs: Unified Modeling Language(UML), state tables, Boolean logic etc.
- FSMs are good for modelling control intensive applications not ideal for modelling data intensive applications.
- FSMs can be annotated with different types of **actions**: Actions on transitions, Entry actions to nodes, Exit actions on nodes.
- Actions can express changes to variables or conditions on variables.
- **Preconditions (guards)**: Conditions that must be true for transition to be taken.
- **Triggering events**: Changes to variables that cause transitions to be taken.
- Node coverage: Execute every state(state coverage).
- Edge coverage: Execute every transition(transition coverage).
- Edge-pair coverage: Execute every pair of transitions(transition-pair).
- Control flow graphs are **not** FSMs representing software/codes.
- Call graphs are also **not** FSMs representing software/codes.
- We need to consider values of variables to represent states of FSMs and statements/actions that result in change of values of variables(states) result in transitions.

5 Testing Source Code: Classical Coverage Criteria

- The most common graph model for source code is control flow graph.
- Structural coverage criteria over control flow graphs deal with **covering** the code in some way or other.
- Data flow graphs augments the control flow with data.
- **Code coverage**: Statement coverage, branch coverage, decision coverage, Modified Condition Decision Coverage(MCDC), path coverage etc.
- Node coverage is same as statement coverage, edge coverage is same as branch, and prime path coverage is the same as loop coverage.
- **Cyclomatic complexity**: Basis path testing, structural testing. It is a software metric used to indicate the (structural) complexity of a program.
- Cyclomatic complexity represents the number of **linearly independent paths** in the control flow graph of a program.
- **Basis path testing** deals with testing each linearly independent path in the CFG of the program.
- A **linearly independent path** of execution in the CFG of a program is a path that does not contain other paths within it. This is very similar to prime paths, every linearly independent path is a prime path.
- The cyclomatic complexity $M = E - N + 2P$, where E is the number of edges, N is the number of nodes, and P is the number of connected components.
- When graph correspond to a single program, $M = E - N + 2$.
- Another way of measuring cyclomatic complexity is to consider *strongly connected components* in CFG. Can be obtained by connecting the final node back to the initial node. Cyclomatic complexity obtained this way is popularly called as **cyclomatic number**.
- If it is less than 10 then the code is not too complex.
- **Data flow testing**: Data flow coverage.
- **Decision-to-decision path** is a path of execution between two decisions in the CFG.
- A **chain** is a path in which initial and terminal vertices are distinct. All the interior vertices have both in-degree and out-degree as 1.
- A **maximal chain** is a chain that is not a part of any other.
- A **DD-path** is a set of vertices in the CFG that satisfies one of the following conditions:
  1. It consists of a single vertex with in-degree 0(initial vertex).
  2. It consists of a single vertex with out-degree 0(terminal vertex).
  3. It consists of a single vertex with in-degree ≥ 2 or out-degree ≥ 2(decision vertices).
  4. It consists of a single vertex with in-degree and out-degree as 1.
  5. It is a maximal chain of length ≥ 1.

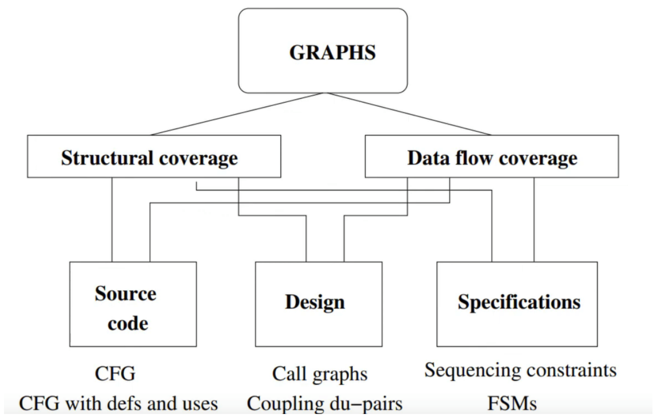


Figure 8: Graph Coverage Criteria Summary

6 Logic

6.1 Basics

- The fragment of logic that we consider is popularly known as **predicate logic** or **first order logic**.
- An **atomic proposition** is a term that is either **true** or **false**.
- Propositional logic deals with combining propositions using **logical connectives** to form **formulas** which are complicated statements.
- Common logical connectives used in proportional logic are
  1. $\vee$ Disjunction
  2. $\wedge$ conjunction
  3. $\neg$ negation
  4. $\supset$ or $\implies$ implies
  5. $\equiv$ or $\iff$ equivalence
- The set ϕ of formulas of propositional logic is the smallest set satisfying the following conditions
  1. All atomic propositions is a member of ϕ
  2. If α is a member of ϕ, so is $\neg\alpha$
  3. If α and β are members of ϕ so is $\alpha \vee \beta$
- Truth tables are a simple way of calculating the semantics of a given propositional logic formula.
- We *split* a formula into its **sub-formulas** repeatedly till we *reach* propositions.
- A formula α is said to be **satisfiable** if there exists a valuation v such that $v(\alpha) = T$.
- A formula of α is to be **valid** or is called a **tautology** if for every valuation it gives True.
- A formula of α is said to be a **contradiction** if for every valuation it gives False.
- Atomic propositions in propositional logic are just like variables that are of type **Boolean**.
- A **predicate** is an expression that evaluates to a Boolean value.
- A **clause** is a predicate that does not contain any logical operators.

6.2 Coverage Criteria

- Let P be a set of predicates and C be a set of clauses in the predicates in P.
- For each predicate $p \in P$, let C_p be the clauses in p. $C_p\{c|c \in p\}$
- **Predicate Coverage**: For each $p \in P$, TR contains two requirements: p evaluates to true and p evaluates to false.
- For a set of predicates associated with branches, predicate coverage is the same as edge coverage.
- **Clause Coverage**: For each $c \in C$, TR contains two requirements: c evaluates to true and c evaluates to false.
- Clause coverage **does not** subsume predicate coverage.
- **Combinatorial Coverage**: For each $p \in P$, TR contains test requirements for the clauses in C_p to evaluate to each possible combination of truth values. Commonly called as **multiple condition coverage**.
- Combinatorial coverage in many times not feasible.
- Sometimes, the truth of certain clauses in predicate makes the other clauses non-influential on the predicate.
- At a given point in time, we are interested in one clause in a predicate, we call this the **major clause**. All other clauses are **minor clauses**.
- Given a major clause c_i in predicate p, we say that c **determines** p if the minor clauses $c_j \in C_p$, $j \neq i$, have values so that changing the truth values of c_i changes the truth value of p.

- **Active Clause Coverage**: For each $p \in P$ and each major clause $c_i \in C_p$, choose minor clauses $c_j, j \neq i$, so that c_i determines p. TR has requirements for each clause as a major clause. For a predicate with n clauses, $n + 1$ distinct test requirements suffice to achieve clause coverage.
- **Modified Condition Decision Coverage(MCDC)** is the same as Active Clause Coverage.
- **General Active Clause Coverage**: TR has two requirements for each c_i, c_i evaluates to true and evaluates to false. The values chosen for the minor clauses c_j do not need to be the same when c_i is true as when c_i is false. GACC **does not** subsume predicate coverage.
- **Correlated Active Clause Coverage**: The values chosen for minor clauses c_j must cause p to be true for one value of the major clause c_i and false for the other. CACC subsumes predicate coverage.
- **Restricted Active Clause Coverage**: The values chosen for the minor clauses c_j must be the same when c_i is true and when it is false.
- **Inactive Clause Coverage**: For each $p \in P$ and each major clause $c_i \in C_p$, choose minor clauses $c_j, j \neq i$ so that c_i does not determine p. TR now has four requirements for c_i, p is true/false for c_i true/false.
- **General Inactive Clause Coverage**: The values chosen for the minor clauses c_j may vary amongst the four cases.
- **Restricted Inactive Clause Coverage**: The values chosen for the minor clauses c_j must be the same for same values of p.

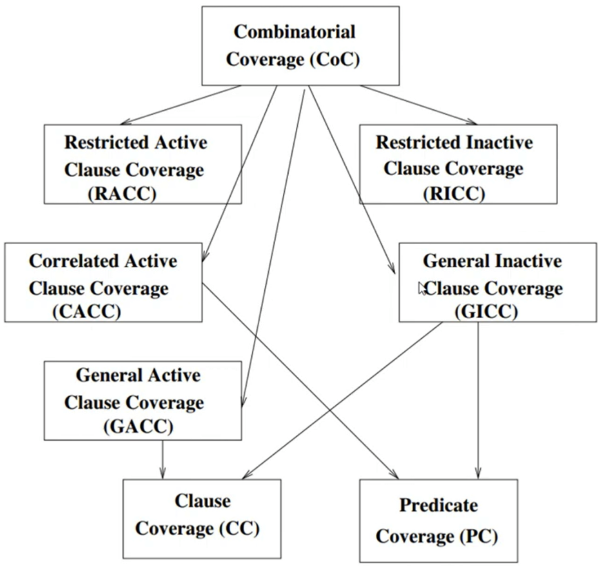

Combinatorial Coverage (CoC)
Restricted Active Clause Coverage (RACC)
Restricted Inactive Clause Coverage (RICC)
Correlated Active Clause Coverage (CACC)
General Inactive Clause Coverage (GICC)
General Active Clause Coverage (GACC)
Clause Coverage (CC)
Predicate Coverage (PC)

Figure 9: Logic Coverage Criteria Subsumption Summary

- As long as a predicate or a clause is not valid/not a contradiction, we can find test cases to achieve predicate and clause coverage.
- Need to identify the **correct** predicates and pass them to a **SAT/SMT solver** to check if the predicate is **satisfiable**.
- Making a clause determine a predicate
  1. Consider a predicate p with a major clause c
  2. Let $p_{c=true}$ represent the predicate p with every occurrence of c replaced with $true$, and $p_{c=false}$.
  3. Note that neither $p_{c=true}$ nor $p_{c=false}$ contains any occurrence of c.
  4. Define $p_c = p_{c=true} \oplus p_{c=false}$, $\oplus$ represents the exclusive or operator.
  5. p_c describes the exact conditions under which the value of c.
  6. If p_c is true then c determines p and if p_c is false then p is independent of c.
- For a predicate p where the value of p_c for a clause c turns out to be true, then ICC criteria are infeasible with respect to c. Similarly, if p_c turns out to be false, ACC criteria are infeasible.