Rishabh Indoria
21F3001823

Software Testing

December 4, 2024

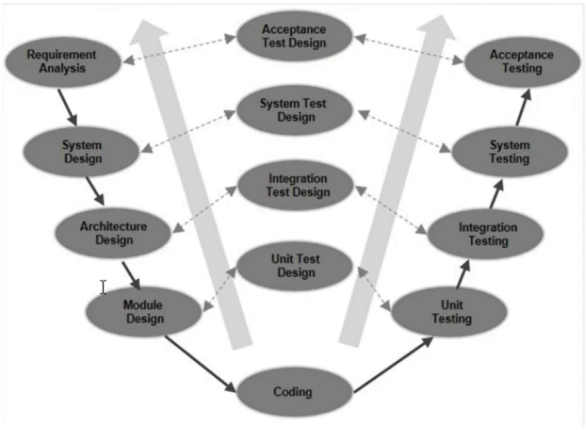
1 Introduction

1.1 Motivation

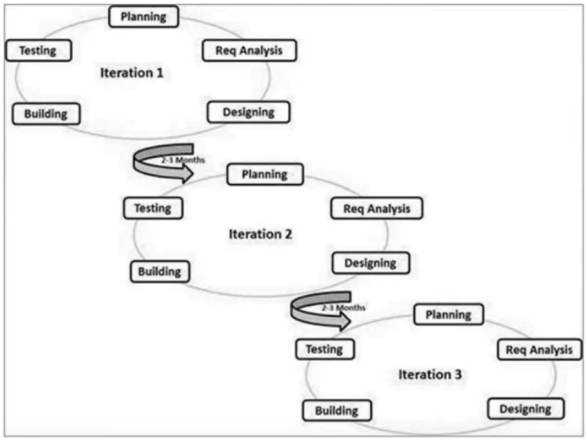
- **Introduction to Software Testing** by Paul Ammann and Jeff Offutt, **The Art of Software Testing** by Glenford J. Myers, **Software Testing: A Craftsman's Approach** by Paul C. Jorgensen, **Agile Testing: A practical guide for Testers and Agile Teams** by Lisa Crispin and Janet Gregory.
- Software 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

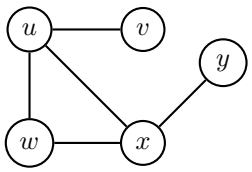
- **Validation**: The process of evaluating software at the end of software development to ensure compliance with intended usage. i.e., checking if the software meets its requirements.
- **Verification**: The process of determining whether the products of a given phase of the software development process 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.

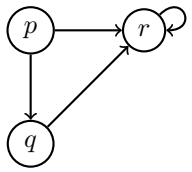
2 Graphs based Testing

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.

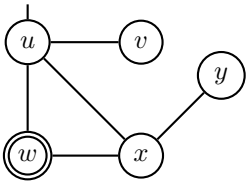
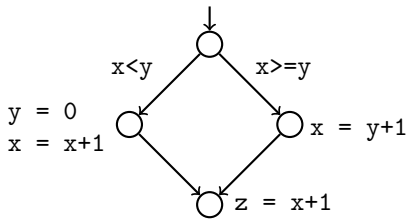


Figure 3: Graph with initial and final vertices

- 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 ∈ G, V − {s} do
 2:     u.color = WHITE, u.d = ∞, u.π = NIL
 3: end for
 4: s.color = BLUE, s.d = 0, s.π = NIL
 5: Q = φ
 6: ENQUEUE(Q, s)
 7: while Q ≠ φ do
 8:     u = DEQUEUE(Q)
 9:     for each v ∈ G.Adj[u] do
10:         if v.color == WHITE then
11:             v.color = BLUE
12:             v.d = u.d + 1
13:             v.π = 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 ∈ G.V do
 2:     u.color = WHITE
 3:     u.π = NIL
 4: end for
 5: time = 0
 6: for each vertex u ∈ 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 ∈ G.Adj[u] do
15:     if v.color == WHITE then
16:         v.π = 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.

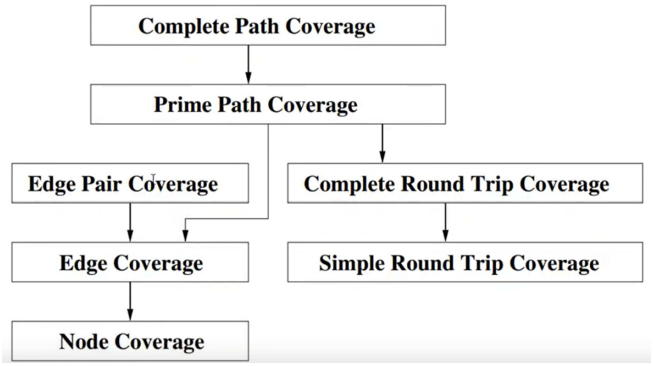


Figure 5: Structure Coverage Criteria Subsumption

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

```
1: Loops = [ ]
2: Terminate = [ ]
3: Stuck = [ ]
4: Q = φ
5: for each v ∈ G do
6:     ENQUEUE(Q, [v])
7: end for
8: while Q ≠ φ do
9:     path = DEQUEUE(Q)
10:    for each v ∈ G.Adj[path[−1]] do
11:        if v is a final vertice then
12:            Terminate = Terminate ++ (path ++ v)
13:        else if v in path and v == path[0] then
14:            Loops = Loops ++ (path ++ v)
15:        else if v in path then
16:            Stuck = Stuck ++ (path)
17:        else
18:            ENQUEUE(Q, path ++ v)
19:        end if
20:    end for
21: end while
22: ▷ From these we can take the largest path as long as that path is not covered by another path.
23: return Loops, Terminate, Stuck
```

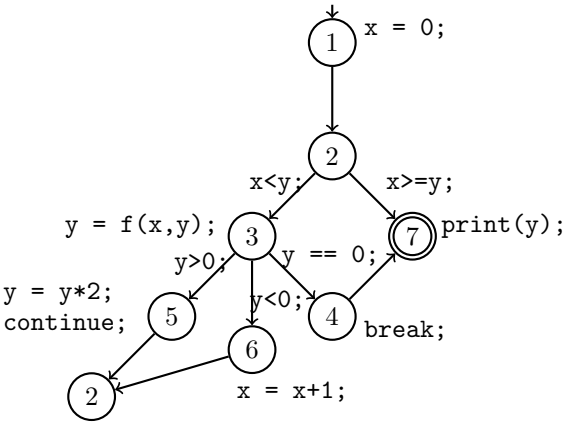
- To enumerate test paths for prime path coverage: Start with the longest prime paths and extend each of them to the initial and the final nodes in the graph.

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 Flow 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.

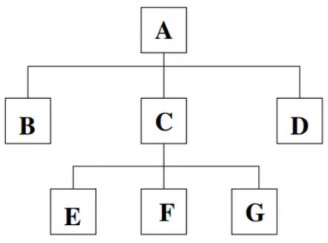


Figure 7: Module Hierarchy

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.
- 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.

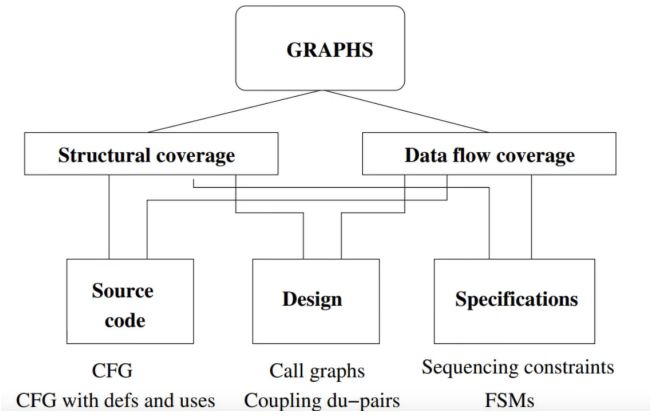


Figure 8: Graph Coverage Criteria Summary

6 Logic Based Testing

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.

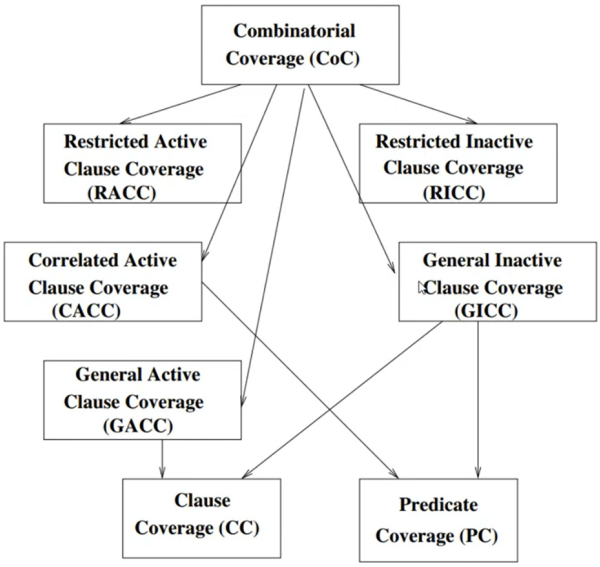


Figure 9: Logic Coverage Criteria Subsumption Summary

- 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(ACC) and if p_c is false then p is independent of c(ICC).
- 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.
- Site to test coverage criteria can be found here.

6.3 Applied to Test Specification

- Specifications can be written in a formal language or in English. They include several logical conditions.
- Pre-conditions, post-conditions and assertions that occur in code and invariant properties about loops contain several logical predicates.
- Programmers often include preconditions for their methods. Often expressed as comments in method headers.
- A predicate is in **Conjunctive Normal Form** (CNF) if it consists of clauses or disjuncts connected by the conjunction (and) operator. Example: $(a \vee b) \wedge (c \vee d)$.
- ACC test requirements is all *true* and then a *diagonal* of false values.
- **Predicate Coverage**: All clauses need to be true for the true case and at least one clause needs to be false for the false case.
- **Clause coverage**: Needs all clauses to be true or false.
- For ACC criteria, each minor clause is true and for remaining tests, each clause is made to be false in turn.
- A predicate is in **Disjunctive Normal Form** (DNF) if it consists of clauses or conjuncts connected by th disjunction (or) operator. Example: $(a \wedge b) \vee (c \wedge d)$.
- ACC test requirements is all *false* and then a *diagonal* of true values.

6.4 Applied to Finite State Machine

- Test cases for various test requirements need inputs and expected outputs.
- Inputs could be states from which transitions originate or actions that result in transitions.
- Outputs could be states resulting from transitions and actions associated with the resulting states, if any.
- These need to be derived from the FSMs.

6.5 SMT Solvers

- The **satisfiability problem (SAT)**, also called the Boolean satisfiability problems or the propositional satisfiability problem is the problem of determining if there exists an assignment of True/False values that **satisfies** a given propositional logic.
- A satisfying assignment is an assignment of True/False values to the atomic propositions such that the formula evaluates to true.
- SAT is the first problem that was proven to be NP-complete.
- Currently, there is no known algorithm that efficiently solves each instance of the SAT problem.
- The tools that solve such instances are referred to as **SAT Solvers**.
- There are a few well-known techniques that they implement to find solutions, some are good at proving unsatisfiability.
- Satisfiability problem of general first order logic is *undecidable*, i.e., there are *no* algorithms that will take every instance of a predicate logic formula and report whether it is satisfiable or not.
- **Satisfiability Modulo Theories (SMT)** is the problem of determining whether a given predicate (or a formula) is satisfiable.
- **SMT solvers** are tools that work towards solving the SMT problem for a simpler, often practical, subset of the logic.
- SMT solvers are used extensively in formal verification and program analysis, for proving the *correctness* of programs.

6.6 Symbolic Testing

- **Symbolic execution** is a means of analyzing a given program to determine what inputs cause each part of the program to execute. This can be effectively used for exercising different executions of the program.
- Consider the following program, and it's respective symbolic execution

```
Sum(a,b,c){
    x = a + b;
    y = b + c;
    z = (x+y) - b;
    return z
}
```

(a) Sum of three numbers

After statement	x	y	y	a	b	c	PC
1	?	?	?	α_1	α_2	α_3	true
2	$\alpha_1 + \alpha_2$	-	-	-	-	-	-
3	-	$\alpha_2 + \alpha_3$	-	-	-	-	-
4	-	-	$\alpha_1 + \alpha_2 + \alpha_3$	-	-	-	-
5	Returns $\alpha_1 + \alpha_2 + \alpha_3$						

(b) Symbolic Execution Table

Figure 10: Symbolic Execution Example

- PC is the Path condition/constraint
- Instead of concrete expression, use symbolic expression.
- During execution, collect path conditions and solve them symbolically.
- Now we can use constraint solver to get concrete values such that all path constraints are true and false.
- **Program proving** involves proving programs correct. Program proving involves formal methods.
- It is used to prove absence of errors.
- Three techniques: Model checking, Theorem proving, and Program analysis.
- Symbolic testing uses program analysis.
- Symbolic execution is a practical approach between the two extremes of program testing and proving.
- Program variables are represented as symbolic expressions over the symbolic input values.
- Symbolic state σ maps variables to symbolic expressions.
- Symbolic path constraints, PC, is a quantifier-free, first order formula over symbolic expressions.
- All the execution paths of a program can be represented using a tree, called the **execution tree**.
- Symbolic execution is used to generate a test input for each execution path of a program.
- At the beginning of symbolic execution: σ is initialized to an empty map.
- At a read statement: symbolic execution adds the mapping assigning the variable being read to its new symbolic variable, to σ
- At every assignment, $v = e$: symbolic execution updates σ by mapping v to $\sigma(e)$, the symbolic expression obtained by evaluating e in the current symbolic state.
- At the beginning of execution: PC is initialized to true
- At a conditional statement $if(e)$ then S_1 else S_2: PC is updated to $PC \wedge \sigma(e)$("then" branch) and a fresh path constraint PC' is created and initialized to $PC \wedge \neg\sigma(e)$("else" branch).
- If PC (PC') is satisfiable for some assignment of concrete to symbolic values, then symbolic execution continues along the "then" ("else") branch with σ and PC (PC').
- If any of PC or PC' is not satisfiable, symbolic execution terminates along the corresponding path.
- At the end of symbolic execution along an execution path of the program, PC is solved using a constraint solver to generate concrete input values.
- Execution can also be terminated if the program hits an exit statement or an error.

- Consider an infinite while loop with condition $N > 0$, PC for the loop with a sequence of n true-s followed by a false is
$$(\wedge_{i\in[1,n]} N_i > 0) \wedge (N_{n+1} \leq 0)$$
where each N_i is a fresh symbolic value.
- In general, symbolic execution of code containing loops or recursion may result in an infinite number of paths if the termination condition for the loop or recursion is symbolic.
- In practice, one needs to put a limit on the search, a timeout, or a limit on the number of paths, loop iterations or exploration depth.
- Symbolic testing is dependent on a constraint solver that takes a path constraint and gives a set of input values to be used as test cases.
- These input values are solutions to the path constraint, popularly known as the satisfiability problem.

6.7 Concolic Testing

- Concolic testing performs symbolic execution dynamically, while the program is executed on some concrete inputs.
- Concolic: Concrete + Symbolic
- Maintains a concrete state and a symbolic state
- Whenever symbolic execution is not possible, then it does normal execution of the program.
- Generate random input, execute program on random input.
- Collect symbolic path constraints encountered along this execution.
- Steer program execution along different execution paths
    1. Use constraint solver to infer variants of symbolic path constraints.
    2. Generate inputs that drive program along different execution paths
    3. Repeat till all execution paths are explored/user-defined coverage criteria is met or time budget expires.
- **Note**: Understanding DART is very confusing, don't spend too much time on this.
- Directed Automated Random Testing is a testing technique that applies to the unit testing phase in software development. Can be applied to large programs, where typically random testing is done.
- DART is a concolic testing tool.
- DART combines three main techniques in order to automate unit testing of programs
    1. Automated extraction of the interface of a program with its external environment using static source-code parsing.
    2. Automatic generation of a test driver for this interface that performs random testing to simulate the most general environment the program can operate in, and
    3. Dynamic analysis of how the program behaves under random testing and automatic generation of new test inputs to direct systematically the execution along alternative program paths.
- DART dynamically gathers knowledge about the execution of the program, called directed search.
- **Memory** $\mathcal{M}$: a mapping from memory address m to say 32-bit words.
$+$ denotes updating of memory. For example, $\mathcal{M}' := \mathcal{M} + [m \to v]$
- **Symbolic variables** are identified by their addresses.
- In an expression, m denotes either a memory address or the symbolic variable identified by address m, depending on the context.
- A **symbolic expression**, e is given by m, c(a constant), $*(e, e')$(multiplication), $\leq (e, e')$(comparison), $\neg e'$(negation), $*e'$(pointer dereference) etc.
- Symbolic variables of an expression e are the set of addresses m that occur in it.
- Expressions have no side effects.

- For DART, we need to define semantics of a program at memory level.
- At memory level, statements are specifically tailored abstractions of the machine instructions that are actually executed.
- Statement labels: A set of labels that denote instruction addresses.
- If l is the address of a statement other than abort or halt then $l+1$ is also an address of a statement. There is an initial address, say l_0.
- A function statement at $(I, \mathcal{M})$ specifies the next statement to be executed.
- Let C be the set of conditional statements and A be the set of assignment statements in P
- A program execution w is a finite sequence of Execs $= (A \cup C) * (abort|halt)$
- A further simplifies program execution: w is of the form $\alpha_1 c_1 \alpha_2 c_2 ... c_k \alpha_{k+1} s$ where $\alpha_i \in A^*$, $c_i \in C$, and $s \in \{abort, halt\}$.
- An alternate view: Consider Execs(P) is a tree; assignment nodes have one or two successors, leaves are labeled by abort or halt.
- Each input vector results as an execution sequence, as a path in this tree.
- DART maintains a symbolic memory $\mathcal{S}$ that maps memory addresses to expressions.
- While the main DART algorithm runs, it will evaluate the symbolic path constraints using this algorithm and solve the path constraints to generate directed test cases.
- To start with, $\mathcal{S}$ just maps each $m \in M_o$ to itself.
- The mapping $\mathcal{S}$ is completed by evaluating expressions symbolically.
- When unsolvable path constraints are encountered, the algorithm uses concrete values instead of symbolic values.

7 Black Box Testing

7.1 Requirements

- **Requirement**: A condition or capability needed by a stakeholder to solve a problem or achieve an objective.
- **Product requirements**: Describe the properties of a product or a system. The product can be a software product too.
- **Process requirements**: Describe the activities to be done by the organization involved in the product development.
- **Business requirements**: Specifies characteristics from the view point of a user or a stakeholder. Document: StRS, Stakeholder Requirements Specification.
- **User requirements**: Specifies what the user expects the software to be able to do. Document: URS, User Requirements Specification. A mutually agreed contract details what the software must do from the point of view of a user.
- **Functional requirements**: Defines a function of a system or its component, where a function is described as a specification of behavior between inputs and outputs. Describes specific/particular results/behaviour of a system. Can be given as use cases. Document: FRS, Functional Requirements Specification. Design document caters to functional requirements.
- **Non-functional requirements**: Specifies criteria that can be used to judge the operation of a system, rather than specific behaviors. Define how a system is supposed to be. Specified as different quality parameters.
- **Regulatory requirements**: Regulation is the management of systems/software as per a set of rules and regulations, specific to several different factors including the environment, business, safety etc. Regulatory requirements could differ from one country to another.
- **Black Box testing**: Black-box testing deals with requirements and spans most of the requirements sketched above. Treats the executable artifacts (code) as a black box. Test cases are designed purely based on the requirements to be tested, inputs and outputs.

7.2 Functional Testing

- A program P is viewed as a function transforming inputs to outputs. Given inputs x_i, P computes outputs y_i such that $y_i = P(x_i)$.
- Precisely identify the domain of each input and each output variable.
- Select values from the data domain of each variable having important properties.
- Consider combinations of special values from different input domains to design test cases.
- Consider input values such that the program under test produces special values from the domains of the output variables.
- Even though functional testing is a black-box testing technique, sometimes, we need to know minimal context information to get relevant values for inputs and outputs.
- **Equivalence class Partitioning**: If the input domain is too large for all its elements to be used as test cases, the input domain is partitioned into a finite number of subdomains for selecting test inputs.
- Each subdomain is known as an equivalence class.
- One subdomain serves as a source for selecting one test input, any one input from each domain is good enough.
- All inputs from one subdomain have the same effect in the program, output will be the same.
- **Boundary Value Analysis** is about selecting test inputs near the boundary of a data domain so that the data both within and outside an equivalence class are selected.
- BVA produces test inputs near the boundaries to find failures caused by incorrect implementation at the boundaries.
- Once equivalence class partitions the inputs, boundary values are chosen on and around the boundaries of the partitions to generate test input for BVA.
- Programmers often make mistakes at boundary values and hence BVA helps to test around the boundaries.
- The partition specifies a range: Construct test cases by considering the boundary points of the range and the points just beyond the boundaries of the range.
- The partition specifies a number of values: Construct test cases for the minimum and the maximum value of the number. In addition, select a value smaller than the minimum and a value larger than the maximum.
- The partition specifies an ordered set: Consider the first and last elements of the set.
- **Decision tables** handle multiple inputs by considering different combinations of equivalence classes. Very popular to test several different categories of software.

		Rules or Combinations							
Conditions	Values	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8
C_1	$Y, N, -$	Y	Y	Y	Y	N	N	N	N
C_2	$Y, N, -$	Y	Y	N	N	Y	Y	N	N
C_3	$Y, N, -$	Y	N	Y	N	Y	N	Y	N
Effects									
E_1		1		2	1				
E_2			2	1			2	1	
E_3		2	1	3		1	1		

Table 1: Decision Table example

- A decision table has
  1. A set of conditions and a set of effects arranged in a column. Each condition has possible value (yes (Y), no(N), don't care(–)) in the second column.
  2. For each combination of the three conditions there are a set of rules from R1 to R8. Each rule has a Y/N/– response and contains an associated set of effects (E1, E2 and E3).
  3. For each effect, an effect sequence number specifies the order in which the effect should be carried out if the associated set of conditions are satisfied.
- In **random testing**, test inputs are selected randomly from the input domain.

7.3 Input Space Partitioning

- Given a set S, a partition of S is a set $\{S_1, S_2, ..., S_n\}$ of subsets of S such that
  The subsets S_i of S are pair-wise disjoint, $S_i \cap S_j = \phi$.
  The union of the subsets S_i is the entire set S, $\bigcup_i S_i = S$.
- The set that is split into partitions while doing testing is the input domain.
- Input domain can be several different sets, one for each input. We may or may not consider all the inputs while doing partitioning.
- Each partition represents one characteristic of the input domain, the program under test will behave in the same way for any element from the partitions.
- There is an underlying equivalence relation that influences the partitions, so input space partitioning is popularly known as equivalence partitioning.
- Each partition is based on some characteristic of the program P that is being tested.
- Characteristics that define partitions must ensure that the partition satisfies two properties: **completeness** (The partitions must cover the entire domain) and **disjoint** (The partitions must not overlap).
- The following are the steps in input domain modelling.
  1. Identification of testable functions.
  2. Identify all the parameters that can affect the behaviour of a given testable function. These parameters together form the input domain of the function under test.
  3. Modelling of the input domain identified in the previous step: Identify the characteristics and partition for each characteristic.
  4. Get the test inputs: A test input is a tuple of values, one for each parameter. The test input should belong to exactly one block from each characteristic.
- Input domain can be modelled in several different ways, needs extensive knowledge of the domain. Both valid and invalid inputs need to be considered.
- **Interface-based modelling**: This method considers each parameter in isolation.
- **Strengths**: It is easy to identify the characteristics, hence easy to model the input domain.
- **Weaknesses**: Not all information that is available to the test engineer will be reflected in the interface domain model, the model can be incomplete.
  Some parts of the functionality may depend on the combinations of specific values of several interface parameters. Analyzing in isolation will miss the combinations.
- Characteristics in this approach are easy, directly based on the individual inputs. Inputs can be obtained from specifications.
- **Functionality based modelling**: This method identifies characteristics based on the overall functionality of the system/function under test, rather than using the actual interface.
- **Strengths**: There is a widespread belief that this approach yields better results than interface-based approach. This is based on the requirements, test case design can start early in the development lifecycle.
- **Weaknesses**: Since it is based on functionality, identifying characteristics is far from trivial. This makes test case design difficult.
- Pre-conditions, post-conditions are typical sources for identifying functionality-based characteristics. Implicit and explicit relationships between variables are another good source. Missing functionality is another characteristic. Domain knowledge is needed.
- **Valid values**: Include at least one set of valid values.
- **Sub-partitions**: A range of valid values can be further partitioned such that each sub-partition exercises a different part of the functionality.
- **Boundaries**: Include values at and close to the boundaries (BVA).
- **Invalid values**: Include at least one set of invalid values.
- **Balance**: It might be cheap or free to add more partitions to characteristics that have fewer partitions.

- **Missing partitions**: Union of all the partitions should be the complete input space for that characteristic.
- **Overlapping partitions**: There should be no overlapping partitions.
- Typically, input domain has several different inputs, each of which can be partitioned.
- Each input domain can be partitioned in several different ways.
- For interface-based ISP, inputs are considered separately.
- For functionality-based ISP, input combinations cutting across partitions need to be considered.
- **All Combinations Coverage (ACoC)**: All combinations of blocks from all characteristics must be used. If we have three partitions as $[A, B]$, $[1, 2, 3]$ and $[x, y]$, then ACoC will have twelve tests.
- A test suite for ACoC will have a unique test for each combination.
- Total number of tests will be $\prod_{i=1}^{n} B_i$, B_i is the number of blocks for each partition, n is the number of partitions.
- ACoC is just an exhaustive testing of considering all possible partitions of the input domain and testing each combination of partitions.
- Apart from the partitions themselves, ACoC has no other advantages, it is like exhaustive testing with respect to the partitions.
- ACoC might not be necessary all the time.
- **Each Choice Coverage (ECC)**: One value from each block for each characteristic must be used in at least one test case.
- If the program under test has n parameters $q_1, q_2, ..., q_n$, and each parameter q_i has B_i blocks, then, a test suite for ECC will have at least $\max_i B_i$ values.
- ECC is a weak criterion.
- ECC will not be effective for arbitrary choice of test values.
- **Pair-Wise Coverage (PWC)**: A value from each block for each characteristic must be combined with a value from every block for each other characteristic.
- A test suite that satisfies PWC will pair each value with each other value or have at least $(\max_i B_i)^2$ values.
- **T-Wise Coverage (TWC)**: A value from each block for each group of t characteristics must be combined.
- If the value for T is chosen to be the number of partitions, then TWC is the same as ACoC.
- A test suite that satisfies TWC will have at least $(\max_i B_i)^t$ values.
- TWC is expensive in terms of the number of tests, going beyond PWC is mostly not useful.
- ACoC considers all combinations, ECC considers each combination.
- PWC and TWC considers combinations blindly without regard for which values are being combined.
- **Base Choice Coverage (BCC)**: A base choice is chosen for each characteristic, and a base test is formed by using the base choice for each characteristic. Subsequent tests are chosen by holding all but one base choice constant and using each non-base choice in each other characteristic.
- A test suite that satisfies BCC will have one base test, plus one test for each remaining block for each partition. Totally, $1 + \sum_{i=1}^{n}(B_i - 1)$.
- **Multiple Base Choices (MBCC)**: At least one, and possibly more, base choice blocks are chosen for each characteristic, and base tests are formed by using each base choice for each characteristic, at least once. Subsequent tests are chosen by holding all but one base choice constant for each base test and using each non-base choice in each other characteristic. The above coverage criteria will cover most of the different ways in which we can partition the inputs to test the software artifact.

- Assuming m_i base choices for each characteristic and a total of M base tests, MBCC requires $M + \sum_{i=1}^{n}(M \times (B_i - m_i))$ tests.

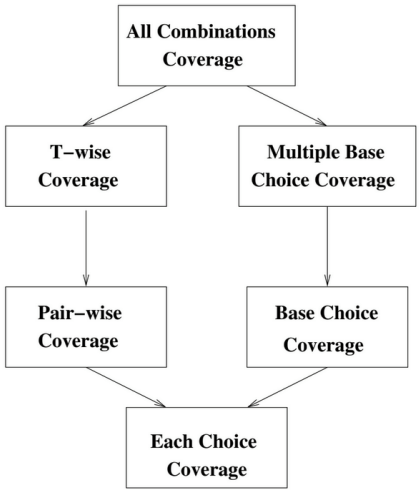


Figure 11: ISP Subsumption relations

- Some combinations of partitions can be infeasible in the input domain model.
- **Constraints** are relations between partitions from different characteristics.
- Two kinds of constraints
  1. A block from one characteristic cannot be combined with a block from another characteristic.
  2. A block from one characteristic must be combined with a block from another characteristic.
- For ACoC, PWC and TWC, the only option is to drop the infeasible combinations from consideration.
- Constraints can be handled better with BCC and MBCC criteria. The base case(s) can be altered to handle infeasible constraints.

8 Syntax Based Testing

8.1 Regular Expressions

- Syntax can be used to generate artifacts that are valid and those that are invalid.
- The structures/artifacts that we generate are sometimes test cases but, most of the time they are not. They are used to generate test cases.
- **Words**: The lexical level, defines how characters from tokens. Generally specified using **regular expressions**.
- **Phrases**: The grammar level, determines how tokens phrases. Generally specified using (deterministic) context-free grammars in **Backus-Naur form**.
- **Context**: Deals with types of variables, what they refer to etc. Generally specified using **context-sensitive grammars**.
- Syntax of regular expressions over an alphabet A

$$r ::= \phi \mid a \mid r + r \mid r \cdot r \mid r^*, \text{ where } a \in A$$

- **Semantics**: Associates a language of words or strings, consider $L(r) \subseteq A^*$ with a regular expression r.

$$\begin{aligned} L(\phi) &= \{\} \\ L(a) &= \{a\} \\ L(r + r') &= L(r) \cup L(r') \\ L(r \cdot r') &= L(r) \cdot L(r') \\ L(r^*) &= L(r)^* \end{aligned}$$

- Consider expressions built from a, b, ϵ, ϵ is a special empty word
  $(a^* + b^*) \cdot c$ are strings of only a's or only b's, followed by a c.

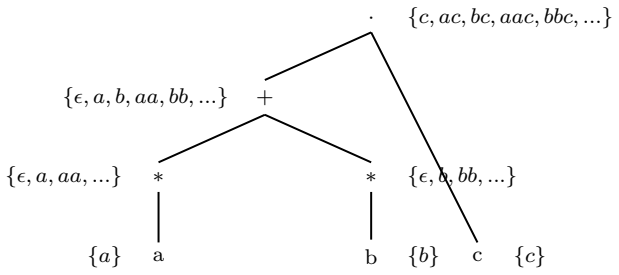


Figure 12: Parse Tree

  $(a+b)^*abb(a+b)^*$ are strings that contain abb as a sub-word
  $(a+b)^*b(a+b)(a+b)$ are strings that have third last letter as b.
  $(ab^*a)^*b^*$ are strings that optionally begin and end with an a, followed by a string of zero or more b's.

- Context-Free Grammar example

$$S \to aX$$
$$X \to aX$$
$$A \to bX$$
$$X \to b$$

  Derivation of a string: Begin with S and keep rewriting the current string by replacing a non-terminal by its RHS in a production of the grammar.
  Consider an example derivation

$$S \implies aX \implies abX \implies abb$$

- Language defined by G, written $L(G)$, is the set of all terminal string that can be generated by G.
- A **Context-Free Grammar (CFG)** is of the form

$$G = (N, A, S, P)$$

  where, N is a finite set of non-terminal symbols
  A is a finite set of terminal symbols
  $S \in N$ is the start non-terminal symbol
  P is a finite subset of $N \times (N \cup A)^*$, called the set of productions or rules.

- α derives β in 0 or more steps: $\alpha \implies {}^*_G \beta$
- First define $\alpha \xrightarrow{n} \beta$ inductively
  1. $\alpha \xrightarrow{1} \beta$ iff α is of the form $\alpha_1 X \alpha_2$ and $X \to \gamma$ is a production in P, and $\beta = \alpha_1 \gamma \alpha_2$
  2. $\alpha \xrightarrow{n+1} \beta$ iff there exists γ such that $\alpha \xrightarrow{n} \gamma$ and $\gamma \xrightarrow{1} \beta$
- **Sentential form** of G: any $\alpha \in (N \cup A)^*$ such that $S \implies {}^*_G \alpha$
- Language defined by G

$$L(G) = \{w \in A^* | S \implies {}^*_G w\}$$

- $L \subseteq A^*$ is called **Context-Free Language (CFL)** if there is a CFG G such that $L = L(G)$
- **Note**: The above math is not that important, learn from the example.
- **Terminal Symbol Coverage (TSC)**: TR contains each terminal symbol t in the grammar G.
- **Production Coverage (PC)**: TR contains each production p in the grammar G.
- **Derivation Coverage (DC)**: TR contains every possible string that can be derived from the grammar G.
- PC subsumes TSC, if we cover every production, we cover every terminal.
- DC is an impractical/infeasible coverage criterion. The number of derivations is infinite for many useful grammars.

8.2 Mutation Testing

- Mutation testing involves making syntactically valid changes to a software artifact and then testing the artifact.
- Grammars generate valid strings. We use derivations in grammars to generate invalid strings as well.
- Testing based on these valid and invalid strings is called **mutation testing**.
- **Ground string**: A string that is in the grammar.
- **Mutation operator**: A rule that specifies syntactic variations of strings generated from a grammer.
- **Mutant**: The result of one application of a mutation operator.
- For a given artifact, let M be the set of mutants, each mutant $m \in M$ will lead to a test requirement.
- The testing goal in mutation testing is to kill the mutants by causing the mutant to produce a different output.
- Given a mutant $m \in M$ for a derivation D and a test t, t is said to kill m iff the output of t on D is different from the output of t on m.
- The derivation D could be represented by the complete list of productions followed, or simply be the final string.
- **Mutation Coverage (MC)**: For each mutant $m \in M$, TR contains exactly one requirement, to kill m.
- the coverage in mutation equates to killing the mutants.
- the amount of coverage is usually written as a percent of mutants killed and is called **mutation score**.
- Higher mutation score doesn't mean more effective testing.
- When a grammar is mutated to produce invalid strings, the testing goal is to run the mutants to see if the behavior is correct.
- **Mutation Operator Coverage (MOC)**: For each mutation operator, TR contains exactly one requirement, to create a mutated string m that is derived using the mutation operator.
- **Mutation Production Coverage (MPC)**: For each mutation operator, and each production that the operator can be applied to, TR contains the requirement to create a mutated string from that production.
- Exhausting mutation testing yields more requirements than other criteria.
- Mutation testing is difficult to apply by hand, automation is also complicated.
- **Program-based mutation**
  1. Begin with the program (ground string).
  2. Apply one or more suitable mutation operators (mutant).
  3. Write tests to kill the mutant.
- **Stillborn mutant**: Mutants of a program result in invalid programs that cannot even be compiles. Such mutants should not be generated.
- **Trivial mutant**: A mutant that can be killed by almost any test case.
- **Equivalent mutant**: Mutants that are functionally equivalent to a given program. No test case can kill them.
- **Dead mutant**: Mutants that are valid and can be killed by a test case. These are the only useful mutants for testing.
- It may not be necessary to see the change only through an output all the time.
- **Strongly killed mutants**: Given a mutant $m \in M$ for a ground string program P and a test t, t is said to strongly kill m iff the output of t on P is different from the output of t on m.
- **Strongly Mutation Coverage (SMC)**: For each $m \in M$, TR contains exactly one requirement to strongly kill m.

- **Weakly killing mutants**: Given a mutant $m \in M$ that modifies a location l in a program P, and a test t, t is said to weakly kill m iff the state of the execution of P on t is different from the state of execution of m immediately after I.
- **Weak Mutation Coverage (WMC)**: For each $m \in M$, TR contains exactly one requirement to weakly kill m.
- Mutation operators are designed to mimic typical programmer mistakes, change relational operators or variable references.
- Mutation operators are designed as an exhastive set. Effective mutation operators can be picked from this set.
- **Effective mutation operators**: If tests that are created specifically to kill mutants created by a collection of mutation operators $O = \{0_1, o_2, ...\}$ also kill mutants created by all remaining operators with very high probability, then O defines an effective set of mutation operators.
- It is difficult to find an effective set of mutation operators for a given program.
- Empirical studies have indicated that mutation operators that insert unary operators and those that modify unary and binary operators will be effective.
- **Absolute Value Insertion**: Each arithmetic expression and sub-expression is modified by the functions $Abs()$, $negAbs()$ and $failOnZero()$
- **Arithmetic Operator Replacement**: Each occurrence of one of the arithmetic operators $+, -, *, /, **$ and % is replaced by each of the other operators. In addition, each is replaced by the special mutation operators $leftOp$(returns the left operand), $rightOp$ and mod.
- **Relational Operator Replacement**: Each occurrence of one of the relational operators $(<, >, \leq, \geq, =, \neq)$ is replaced by each of the other operators and by $falseOp$ and $trueOp$.
- **Conditional Operator Replacement**: Each occurrence of each logical operator (and, or, not) is replaced by each of the other operators. In addition, each is replaced by $falseOp$, $trueOp$, $leftOp$, and $rightOp$.
- **Shift Operator Replacement**: Each occurrence of one of the shift operators $<<, >>$ and $>>>$ is replaced by each occurrence of the other operators. In addition, each is replaced by the special mutation operator $leftOp$.
- **Logical Operator Replacement**: Each occurrence of each bitwise logical operator (bitwise and, bitwise or and exclusive or) is replaced by each of the other operators. In addition, each is replaced by $leftOp$ and $rightOp$.
- **Assignment Operator Replacement**: Each occurrence of one of the assignment operators $(+=, -=, *=, =, \%=, \&=, |=, \hat{}=, <<=, >>=, >>>=)$ is replaced by each of the other operators.
- **Unary Operator Insertion**: Each unary operator (arithmetic +, arithmetic −, conditional, logical ∼) is inserted before each expression of the correct type.
- **Unary Operator Deletion**: Each unary operator is deleted.
- **Scalar variable replacement**: Each variable reference is replaced by every other variable of the appropriate type that is declared in the current scope.
- **Bomb Statement Replacement**: Each statement is replaced by a special $Bomb()$ function that signals a failure as soon as it is executed.
- Mutation is considered as the strongest test criterion in terms of finding the most faults.
- Mutation operators that ensure coverage of a criterion are said to **yield** the criterion.
- Typical coverage criteria impose only a *local* requirement whereas mutation testing imposes a *global* requirement in addition to local requirements.
- Mutation testing thus imposes *stronger* requirements than the other coverage criteria.
- Mutation testing subsumes node coverage. The mutation operator that replaces statements with "bombs" yields node coverage.
- Mutation testing subsumes edge coverage. The mutation operator to be applied is to replace each logical predicate with both true and false.

- Predicate coverage for logic is same as edge coverage hence mutation subsumes predicate coverage.
- Mutation **does not** subsume combinatorial coverage.
- Clause coverage requires each clause to become both true and false. The relational, conditional and logical mutation operators will together replace each clause in each predicate with both true and false. This will subsume clause coverage.
- Mutation testing subsumes GACC but does not subsume CACC or RACC.
- It is not known whether mutation testing subsumes ICC.
- Strong mutation is needed to subsume all-defs coverage. Apply delete mutation to statements that contain variable definitions.
- **Integration mutation** works by creating mutants on the connections between components.
- Most of the mutations are around method calls, and both the calling(caller) and the called(callee) method are considered.
- Integration mutation operators do the following
  1. Change a calling method by modifying the values that are sent to a called method.
  2. Change a calling method by modifying the call.
  3. Change a called method by modifying the values that enter and leave a method. This should include parameters and variables from a higher scope.
  4. Change a called method by modifying statements that return from the method.
- **Integration Parameter Variable Replacement (IVPR)**: Each parameter in a method call is replaced by each other variable of compatible type in the scope of the method call.
- **Integration Unary Operator Insertion (IUOI)**: Each expression in a method call is modified by inserting all possible unary operators in front and behind it.
- **Integration Parameter Exchange (IPEX)**: Each parameter in a method call is exchanged with each parameter of compatible type in that method call.
- **Integration Method Call Deletion (IMCD)**: Each method call is deleted. If the method returns a value, and it is used in an expression, the method call is replaced with an appropriate constant value.
- **Integration Return Expression Modification (IREM)**: Each expression in each return statement in a method is modified by applying unary and arithmetic operators.
- **Pitest** is a state-of-the-art mutation testing system for java.
- **Stryker** is another tool for JavaScript, C# and Scala.
- **Jumble** is another tool that will work with JUnit.

9 Object-Oriented Testing

9.1 Basic Concepts

- **Encapsulation** is an abstraction mechanism to enforce information hiding. It frees clients of an abstraction from unnecessary dependence on design decisions in the implementation of the abstraction.

Specifier	Same class	Different class same package	Different package subclass	Different package non-subclass
private	Y	n	n	n
package	Y	Y	n	n
protected	Y	Y	Y	n
public	Y	Y	Y	Y

Table 2: Access levels in Java

- A subclass **inherits** variables and methods from its parent and all of its ancestors. Subclass can then use them, override the methods or hide the variables.

- **Method overriding** allows a method in a subclass to have the same name, arguments and result type as a method in its parent.
- **Variable hiding** is achieved by defining a variable in a child class that has the same name and type of inherited variable.
- **Overloading** is the use of the same name for different constructors or methods in the same class with different signatures.

9.2 Mutation Operators

- **Access Modifier Change (AMC)**: The access level for each instance variable and method is changed to other access levels.
- **Hiding Variable Deletion (HVD)**: Each declaration of an overriding, or hiding variable is deleted.
- **Hiding Variable Insertion (HVI)**: A declaration is added to hide the declaration of each variable declared in an ancestor.
- **Overriding Method Deletion (OMD)**: Each entire declaration of an overriding method is deleted.
- **Overriding Method Moving (OMM)**: Each call to an overridden method is moved to the first and last statements of the method and up and down one statement.
- **Overridden Method Rename (OMR)**: Renames the parent's version of methods that are override in a subclass so that the overriding does not affect the parent's method.
- **Super Keyword Deletion (SKD)**: Delete each occurrence of the super keyword.
- **Parent Constructor Deletion (PCD)**: Each call to a super constructor is deleted.
- **Actual Type Change (ATC)**: The actual type of new object is changed in the *new*() statement.
- **Declared/Parameter Type Change (DTC/PTC)**: The declared type of each new object/each parameter object is changed in the declaration.
- **Reference Type Change (RTC)**: This right side objects of assignment statements are changed to refer to objects of a compatible type.
- **Overloading Method Change (OMC)**: For each pair of methods that have the same name, the bodies are interchanged.
- **Overloading Method Deletion (OMD)**: Each overloaded method declaration is deleted, one at a time.
- **Argument Order Change (AOC)**: The order of the arguments in method invocations is changed to be the same as that of another overloading method, if one exists.
- **Argument Number Change(ANC)**: The number of the arguments in method invocations is changed to be the same as that of another overloading method, if one exists.
- *this* **Keyword Deletion (TKD)**: Each occurrence of the keyword *this* is deleted.
- **Static Modifier Change (SMC)**: Each instance of the *static* modifier is removes, and the *static* modifier is added to instance variables.
- **Variable Initialization Deletion (VID)**: Remove initialization of each member variables.
- **Delete Constructor Deletion (DCD)**: Delete each declaration of default constructor, with no parameters.

9.3 Integration Testing

- **Intra-method Testing**: Tests are constructed for individual methods, traditional unit testing
- **Inter-method Testing**: Multiple methods within a class are tested in concert, traditional module testing.
- **Intra-class Testing**: Tests are constructed for a single class, usually as sequences of calls to methods within the class.
- **Inter-class Testing**: More than one class is tested at the same time, usually to see how they interact, kind of integration testing.

- We assume that a class encapsulates state information in a collection of *state variables*. The behaviors of a class are implemented by methods that use the state variables.
- The interactions between the various classes and methods within/outside a class that occur in the presence of inheritance, polymorphism and dynamic binding are complex and difficult to visualize.
- The **yo-yo** graph is defined on an inheritance hierarchy
  1. It has a root and descendants
  2. Nodes are methods: new, inherited and overridden methods for each descendant.
  3. Edges are method calls as given in the source: directed edge is from caller to callee.
  4. Each class is given a level in the graph that shows the actual calls made if an object has the actual type of that level. These are depicted by **bold arrows**.
  5. **Dashed arrows** are calls that cannot be made due to overriding.

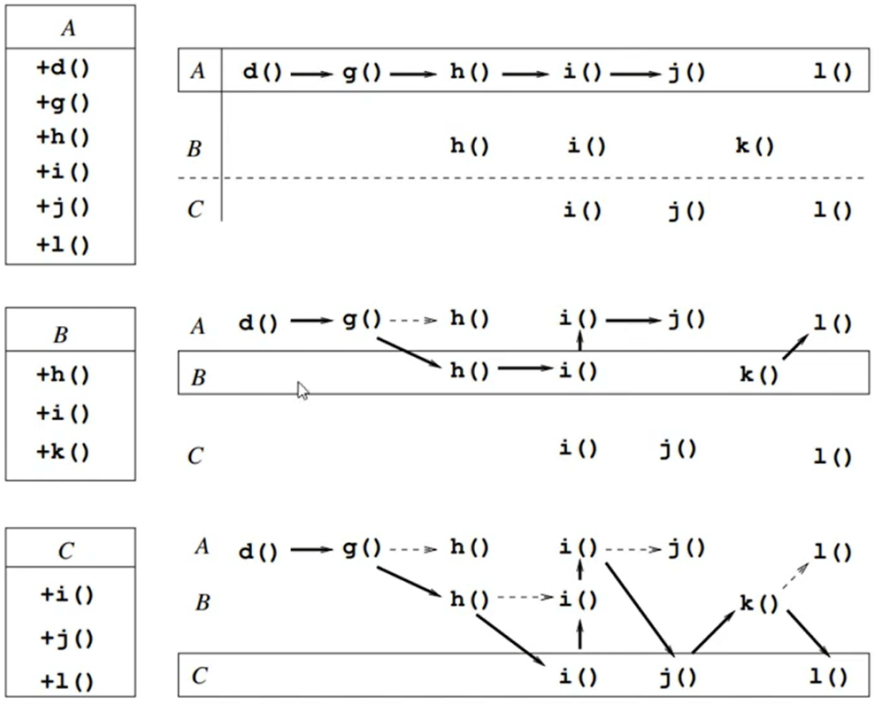


Figure 13: Yo-Yo graph example

9.4 Faults and Anomalies

- Complexity is relocated to the connections among components.
- Many faults can now only be detected at runtime.
- We are assuming that an anomaly/fault is manifested through polymorphism in a context that uses an instance of an ancestor, i.e., instances of descendant classes can be substituted for instances of the ancestor.
- Faults described are language independent.
- **Inconsistent type use fault (ITU)**: A descendant class does not override any inherited method.
- **State Definition Anomaly (SDA)**: The state interactions of a descendant are not consistent with those of its ancestor. The refining methods fail to define some variables and hence required definitions might not be available.
- **State Definition Inconsistency Anomaly (SDIH)**: A local variable is introduced to a class definition and the name of the variable is the same as an inherited variable v. A reference to v will refer to the descendant's v.
- **State Visibility Anomaly (SVA)**: Consider a class W, which is an ancestor X and Y, X extends W and Y extends W. W declares a private variable v, v is defined by $W :: m$. Y overrides $m()$, and calls $W :: m()$ to define v. This causes a data flow anomaly as v is private for W.

- **State Defined Incorrectly**: An overriding method defines the same state variable that the overridden method defines. In the absence of identical computations by the two methods, this could result in a behavior anomaly.
- **Indirect Inconsistent State Definition Fault**: A descendant adds an extension method that defines an inherited state variables.

9.5 Coupling Criteria

- **Coupling variables**: Variables defined in one unit and used in another unit.
- **Coupling Sequences**: Pairs of method calls within body of method under test.
  1. Made through a common instance context.
  2. With respect to a set of state variables that are commonly referenced by both methods.
  3. Consists of at least one coupling oath between the two method calls with respect to a particular state variable.
- These coupling sequences represent potential state space interactions between the called methods with respect to calling method.
- A reference o can refer to instances whose actual instantiated types are either the base type of o or a descendant of o's type.
- o is considered to be defined when one of the state variables v of o is defined.
- Definitions and uses in object-oriented applications for coupling variables can be indirect.
- Indirect definitions and uses occur when a method defines or references a particular value v.
- **Polymorphic call set** or **Satisfying set**: Set of methods that can potentially execute as result of a method call through a particular instance context.
- The calling method is the **coupling method** $f()$, it calls $m()$, the **antecedent method**, to define a variable, and $n()$, the **consequent method**, to use the variable.
- **Transmission set**: Variables for which a path is def-clear.
- **Coupling set** is the intersection of the variables defined by $m()$, used by $n()$.
- **All-Coupling-Sequences (ACS)**: For every coupling sequence S_j in $f()$, there is at least one test case t such that there is a coupling path induced by $S_{j,k}$ that is a sub-path of the execution trace of $f(t)$.
- At least one coupling path must be executed. Does not consider inheritance and polymorphism.
- **All-Poly-Classes (APC)**: For every coupling sequence $S_{j,k}$ in method $f()$, and for every class in the family of types defined by the context of $S_{j,k}$, there is at least one test case t such that when $f()$ is executed using t, there is a path p in the set of coupling paths of $S_{j,k}$ that is a sub-path of the execution trace of $f(t)$.
- At least one test for every type the object can bind to. Include instance contexts of calls.
- **All-Coupling-Defs-and-Uses (ACDU)**: For every coupling variable v in each coupling $S_{j,k}$ of t, there is a coupling path induced by $S_{j,k}$ such that p is a sub-path of the execution trace of $f(t)$ for at least one test case t.
- Every last definition of a coupling variable reaches every first use. Does not consider inheritance and polymorphism.
- **All-Poly-Coupling-Defs-and-Uses (APDU)**: For every coupling sequence $S_{j,k}$ in $f()$, for every class in the family of types defined by the context of $S_{j,k}$, for every coupling variable v of $S_{j,k}$, for every node m that has a last definition of v and every node n that has a first-use of v, there is a path p in the coupling paths of $S_{j,k}$, that is a sub-path of the trace of $f()$.
- Every last definition of a coupling variable reaches every first use for every type binding. Handles inheritance and polymorphism.

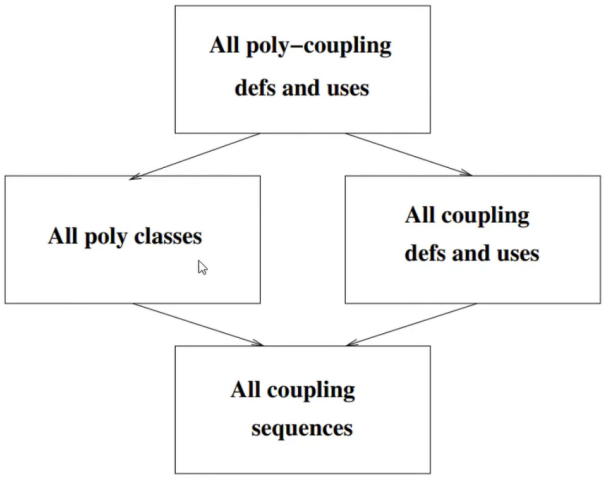


Figure 14: Object-Oriented Subsumption

10 Web Application Testing

10.1 Introduction

- A **web application** is a program that is deployed on the web.
- It is like a client-server application where the client runs in a web browser.
- Composed of independent, loosely coupled software components.
- Inherently concurrent and often distributed.
- Most components are relatively small.
- Uses numerous new technologies, often mixed together.
- **Presentation layer**: HTML, output and UI.
- **Data content layer**: Computation, data access.
- **Data representation layer**: In-memory data storage.
- **Date storage layer**: Permanent data storage.
- Web applications are heterogeneous, dynamic and must satisfy very high quality attributes.
- Security breaches on web applications are also a major concern.
- HTTP is a stateless protocol, each request is independent of previous request.
- State is managed by using cookies, session objects, etc.
- Servers have little information about where a request comes from.
- Users can change the flow of control arbitrarily.
- A **web service** is a system of software that allows machines to interact with each other through a network.
- Typically used to achieve reusability of application components. Many web services don't have user interfaces.
- A **web page** contains HTML content that can be viewed in a single browser window.
- A **website** is a collection of web pages and associated software elements that are related semantically by content and syntactically through links and other control mechanism.
- A **static web page** is unvarying and the same to all users.
- A **dynamic web page** is created by a program on demand.

- A **test case for a web application** is a sequence of interactions between components on clients and servers. They are paths of transitions through the web applications.
- Testing for static websites
  1. This is not program testing, but checking that all the HTML connections are valid.
  2. The main issue to test for is **dead links**, i.e., links to URLs that are no longer valid.
  3. We should also evaluate several non-functional parameters: Load testing, Performance evaluation, Access control issues.
- Nodes are web pages, and edges are HTML links.
- The graph is built by starting with an introductory page and recursively doing BFS of all links from that page. This graph is tested for edge coverage.

10.2 Client-Side Testing

- The user interface is on the client and the actual software is on the server.
- Tester typically has no access to data, state or the source code on the server.
- Test inputs are HTML form elements.
- Inputs can be generated or chosen.
- **Bypass testing**: Values that violate constraints on the inputs, as defined by client-side information.
- Bypass testing creates inputs that intentionally violate these validation rules and constraints.
- The basic idea in **bypass testing** is to let a tester save and modify the HTML.
- Server side inputs can be modified too, but, can be risky if they corrupt data in the server.
- Client side input validation is performed by HTML form controls, their attributes and client side scripts that access DOM.
- Validation types are categorized as HTML and scripting.
- HTML supports syntactic validation.
- Client scripting can perform both syntactic and semantic validation.
- **Value level bypass testing** tries to verify if a web application adequately evaluates invalid inputs.
- **Parameter level bypass testing** tries to check for issues related to relationships among different parameters of an input.
- **Control flow level bypass testing** tries to verify web applications by executing test cases that break the normal execution sequence.
- A testing approach that uses data captured during **user sessions** to create test cases.
- User-session data based testing reduces the effort involved when test engineers are required to generate test cases.
- Let $U = \{u_1, u_2, ..., u_m\}$ be a set of user sessions, with u_i consisting of n requests $r_1, r_2, ..., r_n$, where each r_i consists of $url[name - value]^*$
- We define a user session as beginning when a request from a new IP address reaches the server and ending when the user leaves the website or the session times out.
- Collect all client request information. This can be accomplished based on the underlying web application development technologies used.
- Three stand-alone variants of the basic approach involving direct use of the session data.
  1. Directly reuse entire sessions: Transform each $u_i \in U$ into a test case by formatting each of its associated requests $r_1, r_2, ..., r_n$ into an HTTP request that can be sent to a web server. The resulting test suite contains m test cases, one for each user session.

2. Replay a mixture of sessions: Select an unused session u_a from U.
   Copy requests r_1 through r_i from u_a, where i is a random number, into the test case.
   Randomly select session u_b from U, where $b \neq a$, and search for any r_j in u_b with same URL as r_i.
   If not r_j found then consider direct reuse, else add all the requests following r_j from u_b into the test case after r_i.
   Mark u_b as used and repeat the process until no more unused sessions are available in U.
3. Replay sessions with some targeted modifications: Replay user sessions by modifying the input forms that can alter the behavior of the web application.
   Select an unused session u_a from U.
   Randomly select an unused request r_i from u_a. If there are no more unused r_i in u_a, then reuse u_a directly as a test case.
   If r_i does not contain at least one name-value pair, mark r_i as used and repeat previous step.
   If r_i has one or more name-value pairs, then modify the name-value pairs, create test cases.
   Mark u_a as used and repeat process until no more.

- Two hybrid variants that combine the basic approach with other functional testing techniques.

10.3 Server-Side Testing

- **Valid responses**: Invalid inputs are adequately processed by the server.
- **Faults and failures**: Invalid inputs cause abnormal server error.
- **Exposure**: Invalid inputs are not recognized by the server and abnormal software behavior is exposed to users.
- If server-side source code is available, we can use graph models to test the server.
- CFG exhibit only static models, not effective for web applications.
- **Presentation layer** of a web application contains software that is useful to do testing.
- **Component Interaction Model (CIM)**: Models individual components, combines atomic sections, intra-component.
- An **atomic section** is a section of HTML with the property that if any part of the section is sent to a client, the entire section is. A HTML file is an atomic section.
- A **content variable** is a program variable that provides data to an atomic section.
- **Application Transition Graph (ATG)**:Each node is one CIM, Edges are transitions among CIMs, Inter-component.
- Atomic sections are combined to model dynamically generated web pages.
- Γ: Finite set of web components
- Θ: Set of transitions among web software components
- Σ: Set of variables that define the web application state
- α: Set of start pages
- **Simple Link Transition**: An HTML link
- **Form Link Transition**: Form submission link
- **Component Expression Transition**: Execution of a software component causes a component expression to be sent to the client.
- **Operational Transition**: A transition out of the software's control.
- **Redirect Transition**: Server side transition, invisible to user.

11 Miscellaneous Testing

11.1 Regression Testing

- **Regression testing** is the process of validating modified software to detect whether new errors have been introduced into previously tested code and to provide confidence that modifications are correct.
- Black-box testing, typically very expensive and well-used.
- Let P be a procedure or program. Let P' be a modified version of P and let T be a test suite for P
- Select $T' \subseteq T$, a set of test cases to execute on P'.
- This involves the **regression test selection** problem.
- Test case t is **obsolete** for program P' iff t specifies an input to P that is invalid for P', or t specifies an invalid input-output relation for P'. Identify these and remove from T.
- Test P' with T', establishing P's correctness with respect to T'
- If necessary create T", a set of new functional or structural test cases for P'
- This involves **coverage identification** problem.
- Test P' with T", establishing P's correctness with respect to T".
- Create T"$'$, a new test suite and test execution profile for P', from T, T', and T".
- This involves the **test suite maintenance** problem.
- And we also have the **test suite execution** problem.
- **Minimization test selection techniques** for regression testing attempt to select minimal sets of test cases from T that yield coverage of modified or affected portions of P.
- **Data-flow coverage based regression test selection** techniques select test cases that exercise data interactions that have been affected by modifications.
- Techniques that are not safe can fail to select a test that would have revealed a fault in the modified program.
- **Random Techniques**: Randomly select test cases, often useless.
- The **retest-all** technique simply reuses all existing test cases.
- Cerberus Testing is a good open source tool.

11.2 Software Quality Metrics

- **Software quality** is the capability of a software product to conform to its requirements.
- **Software function quality**: Reflects how well it complies with or conforms to a given design, based on functional requirements or specifications.
- **Software structural quality**: Refers to how it meets non-functional requirements that support the delivery of the functional requirements, such as robustness or maintainability.
- **Software Quality Assurance**: A set of activities to ensure the quality in software engineering processes that ultimately result in quality software products.
- **Software Quality Control**: A set of activities to ensure the quality of software products.
- A **software metric** is a measure of the degree/extent to which a software system or process possesses some property.
- **Product metrics**: Describe the various characteristics of the software, size, complexity, metric related to design.
- **Process metrics**: Describe all the characteristics that can be used to improve the development of software and its maintenance.
- **Project metrics**: Describe the characteristics of the project and its execution, team size, cost, schedule, productivity.

- Number of lines of code(KLOC), Cyclomatic complexity, Program size.
- **Coupling**: Coupling is the degree of interdependence between software modules, it is a measure of how closely connected two modules are.
- **Cohesion**: Cohesion is a measure of the strength of relationship between the methods and data of a class.
- **Mean time to failure**: The time between failures. Used mainly in safety critical systems.
- **Defect Density**: Measures the number of defects relative to the size of software (KLOC/function points)
- **Issues reported by/related to customers**: Number of issues per unit of time, as reported by customers.
- **Satisfaction of customers**: Measured through a survey on a five-point scale.
- **Defect Removal Effectiveness** is defined as

$$DRE = \frac{\text{Defects removed during a development phase}}{\text{Defects latent in the product}} \times 100\%$$

- Fix backlog and backlog management using Backlog Management Index

$$BMI = \frac{\text{Number of problems closed during the month}}{\text{Number of problems arrived during the month}} \times 100\%$$

11.3 Non-Functional Testing

- **Interoperability testing** determines whether the system can interoperate with other third-party products. Can involve **compatibility testing** too.
- Forward and Backward compatibility.
- **Security testing** determines if a system products data and maintains security related functionality as intended.
- **Confidentiality**: Requirement that data and processes by protected from unauthorized disclosure.
- **Integrity**: Requirement that data and processes be protected from unauthorized modification.
- **Availability**: Requirement that data and processes be protected from denial of service to authorized users.
- Also deals with **authorization** and **authentication** verification.
- Verify that only authorized accesses to the system are permitted.
- Verify the correctness of encryption and decryption algorithms for systems where data/messages are encoded.
- Verify that illegal reading of files, to which the perpetrator is not authorized is not allowed.
- Ensure that virus checkers prevent/curtail entry of viruses into the system.
- Identify back doors in the system left open by developers. Test for access through back doors.
- Verify different authentication, client server communication, wireless security protocols etc.
- **Reliability tests** measure the ability of system to keep operating over specified periods of time.
- **Scalability testing** verifies that a system can scale up to its engineering limits.
- **Documentation testing** is done to verify the technical accuracy and readability of various documentation including user manuals, tutorials, online help etc.
- **Read test**: A documentation is reviewed for clarity, organization, flow and accuracy without executing the documented instructions on the system.
- **Hands-on test**: Online help is exercised, error messages verified to evaluate their accuracy and usefulness.
- **Functional test**: Instructions embodied in the documentation are followed to verify that the system works as it has been documented.
- Each country has regulatory bodies guiding the availability of a product in that country.
- **Performance testing** is done to determine the system parameters in terms of responsiveness and stability under various workloads.
- **Soak testing** is endurance testing and **spike testing** is stress testing.
- **Test Driven Development**: Write tests cases then code for those test cases. As code size increases, do **re-factoring**.