Model Based Testing

Model-based testing is a software testing technique in which the test cases are derived from a model that describes the functional aspects of the system under test.
We can devise test cases to check actual behavior against behavior specified by the model.
A common kind of model for describing behavior that depends on sequences of events or stimuli Example: UML state diagrams

• State coverage: – Every state in the model should be visited by at least one test case

• Transition coverage – Every transition between states should be traversed by at least one test case.

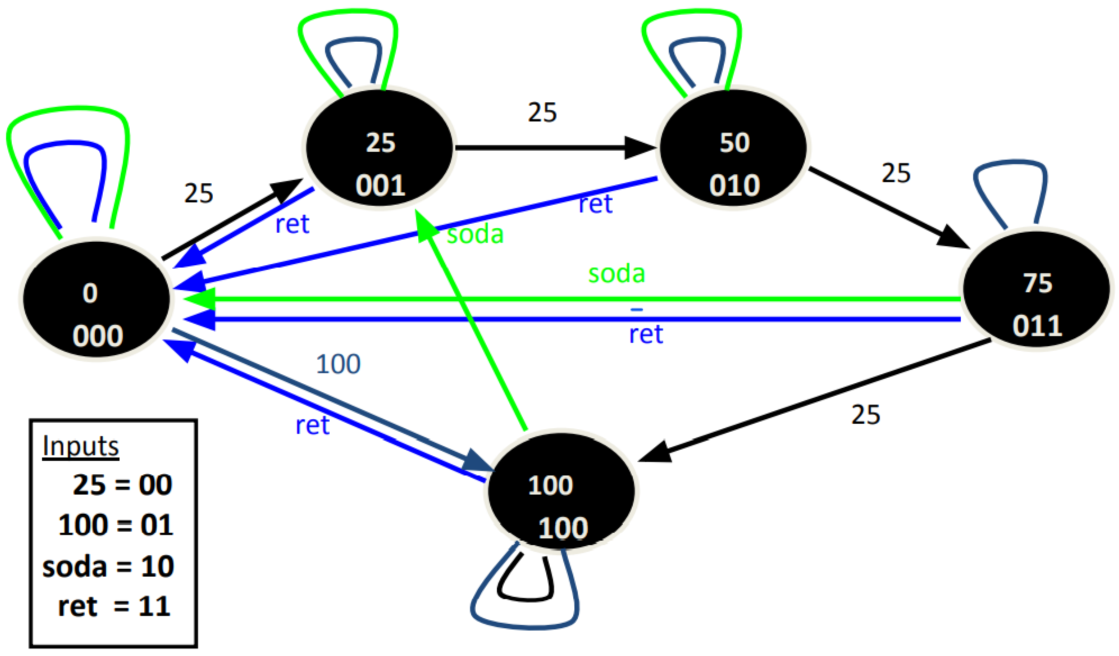
Finite State Machine

- A finite-state machine is a mathematical model of computation.
- It is an abstract machine that can be in exactly one of a finite number of states at any given time.
- The FSM can change from one state to another in response to some external inputs; the change from one state to another is called a transition.
- An FSM is defined by a list of its states, its initial state, and the conditions for each transition.
- Examples: Vending machine, Elevator, Traffic lights etc
- FSM is a graph that describes how software variables are modified during execution.
- FSM helps to analyze early detection of errors through analysis of the model such as UML, State tables, Boolean logic etc. Since, these are modeled even before the code is generated.

Nodes – represent states which in turn indicates the set of values for the key variables
• Edges – represent transitions which will have guards (conditions) and or actions.
• Preconditions (guards) – Conditions to be present for transition to occur
• Triggering events – changes to variables that cause the transitions
• Testers thus have to create test cases where every state has to be visited and hence every path will be checked for i.e for every transition there should be test cases generated

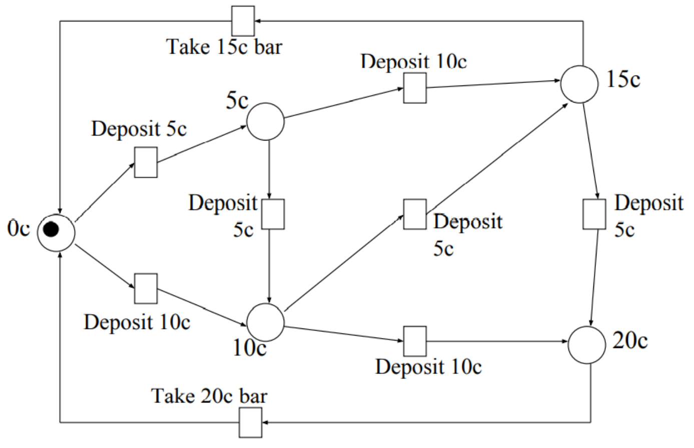
Example: Vending Machine
• Takes only quarters and dollar bills
• Won't hold more than $1.00
• Sodas cost $.75
• Possible actions (inputs) – deposit $.25 (25) – deposit $1.00 ($) – push button to get soda (soda) – push button to get money returned (ret)

• State: description of the internal settings of the machine, e.g. how much money has been deposited and not spent
• Finite states: 0, 25, 50, 75, 100,
• Rules: determine how inputs can change state



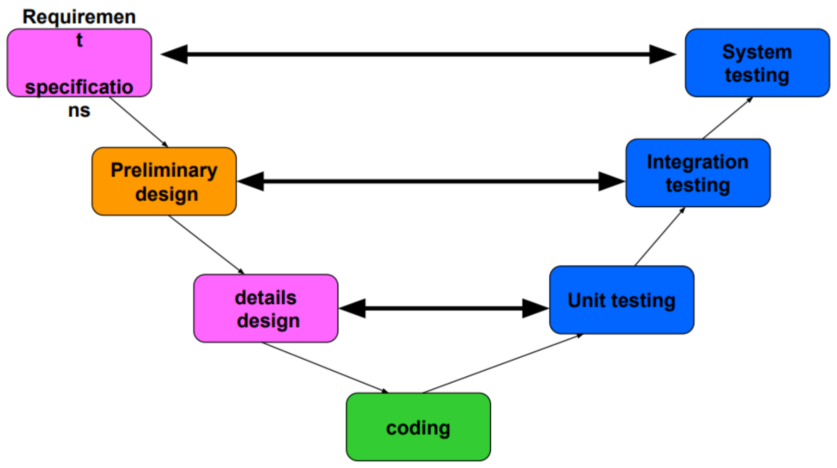
EFSA
Expanded to include deterministic time
Graphically, places, transitions, arcs, and tokens are represented respectively by: circles, bars, arrows, and dots.
A place may have zero or more tokens.
Transition node is ready to fire if and only if there is at least one token at each of its input places

Example: Vending Machine (A Petri net)



- Let M be the FSM model of a system and I_M be its implementation.
- An important testing task is to confirm if I_M behaves like M.
- Conformance testing: Ensure by means of testing that I_M conforms to its spec. M.
- A general procedure for conformance testing
  - Derive sequences of state-transitions from M.
    - Turn each state-transition into a test sequence.
    - Test I_M with a test sequence to observe whether or not I_M possesses the corresponding transition sequence.
  - The conformance of I_M with M can be verified by choosing enough state-transition sequences.

The waterfall life cycle



Approaches to Integration Testing

1. **Based on Functional Decomposition**

   Top-Down Hybrid
   Bottom-Up
   Sandwich
   Big Bang
   Incremental

2. **Based on Call Graph**

   Pair-wise
   Neighborhood

3. **Based on Paths**

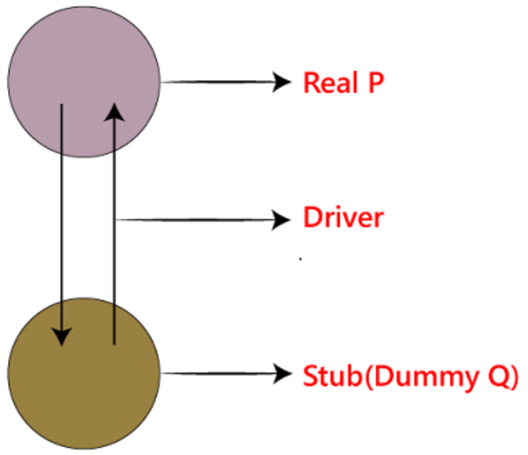
   MM-Paths
   Atomic System Functions

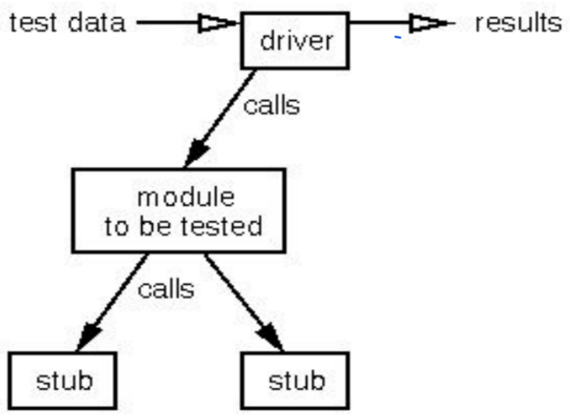
Integration testing

Integration testing is the second level of the software testing process after unit testing. In this testing, units or individual components of the software are tested in a group. The focus of the integration testing level is to expose defects at the time of interaction between integrated components or units.

Stub and driver

The stub is a dummy module that receives the data and creates lots of probable data, but it performs like a real module. When data is sent from module P to Stub Q, it receives the data without confirming and validating it, and produces the estimated outcome for the given data.



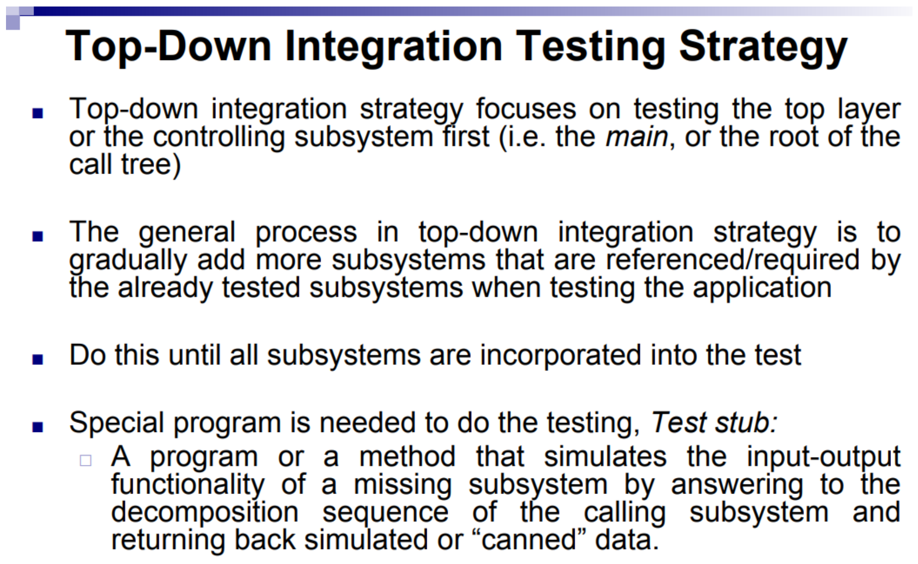


The function of a driver is used to verify the data from P and sends it to stub and also checks the expected data from the stub and sends it to P.

The driver is one that sets up the test environments and also takes care of the communication, evaluates results, and sends the reports. We never use the stub and driver in the testing process.

Top-Down Approach

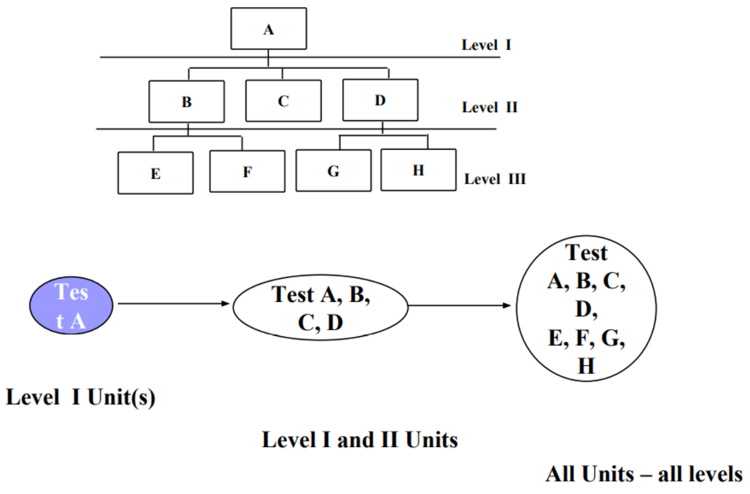
The top-down testing strategy deals with the process in which higher level modules are tested with lower level modules until the successful completion of testing of all the modules. Major design flaws can be detected and fixed early because critical modules tested first. In this type of method, we will add the modules incrementally or one by one and check the data flow in the same order.



Top-Down Integration Testing Strategy

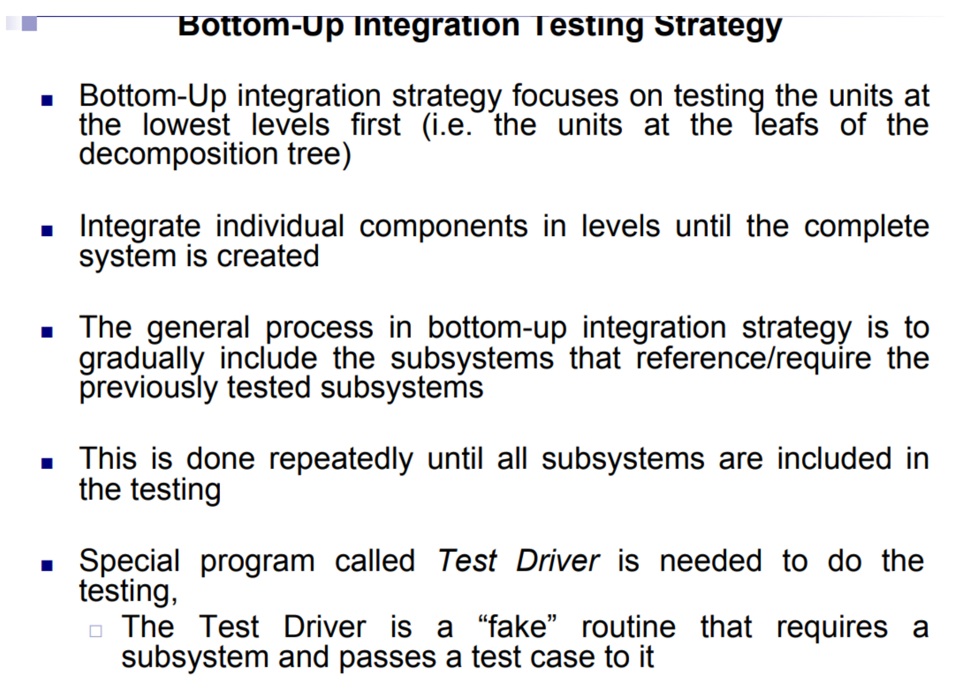
- Top-down integration strategy focuses on testing the top layer or the controlling subsystem first (i.e. the *main*, or the root of the call tree)
- The general process in top-down integration strategy is to gradually add more subsystems that are referenced/required by the already tested subsystems when testing the application
- Do this until all subsystems are incorporated into the test
- Special program is needed to do the testing, *Test stub:*
  - A program or a method that simulates the input-output functionality of a missing subsystem by answering to the decomposition sequence of the calling subsystem and returning back simulated or "canned" data.

Top-down Integration Testing Strategy



Bottom-Up Method

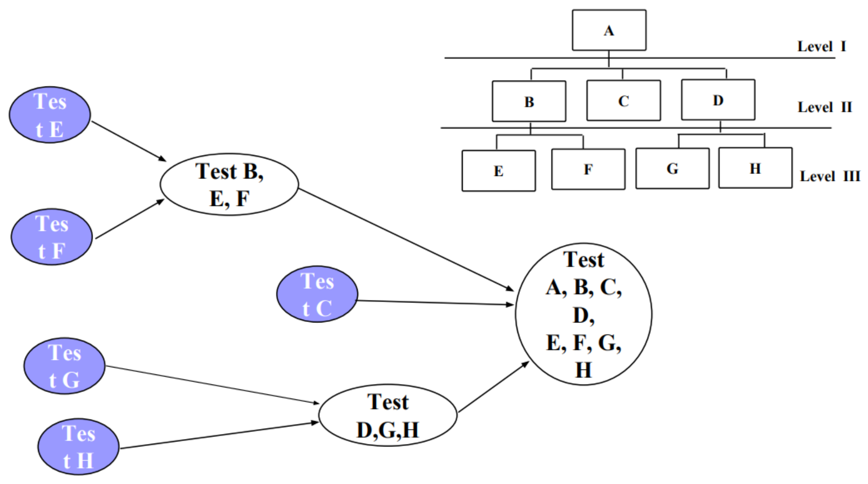
The bottom to up testing strategy deals with the process in which lower level modules are tested with higher level modules until the successful completion of testing of all the modules. Top level critical modules are tested at last, so it may cause a defect. Or we can say that we will be adding the modules from bottom to the top and check the data flow in the same order.



Bottom-Up Integration Testing Strategy

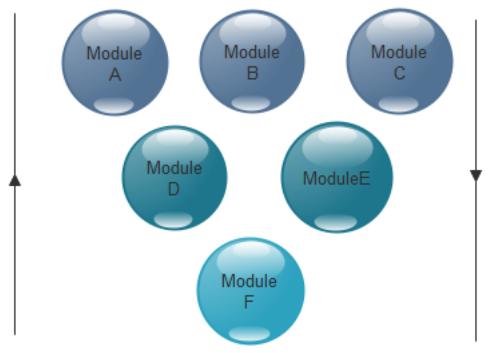
- Bottom-Up integration strategy focuses on testing the units at the lowest levels first (i.e. the units at the leafs of the decomposition tree)
- Integrate individual components in levels until the complete system is created
- The general process in bottom-up integration strategy is to gradually include the subsystems that reference/require the previously tested subsystems
- This is done repeatedly until all subsystems are included in the testing
- Special program called *Test Driver* is needed to do the testing,
  - The Test Driver is a “fake” routine that requires a subsystem and passes a test case to it

Example Bottom-Up Strategy



Hybrid Testing Method

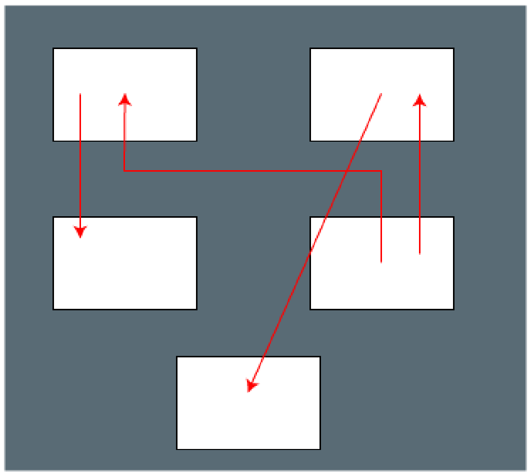
In this approach, both Top-Down and Bottom-Up approaches are combined for testing. In this process, top-level modules are tested with lower level modules and lower level modules tested with high-level modules simultaneously. There is less possibility of occurrence of defect because each module interface is tested.



Hybrid Method

Big Bang Method

In this approach, testing is done via integration of all modules at once. It is convenient for small software systems, if used for large software systems identification of defects is difficult. we will create the data in any module bang on all other existing modules and check if the data is present.



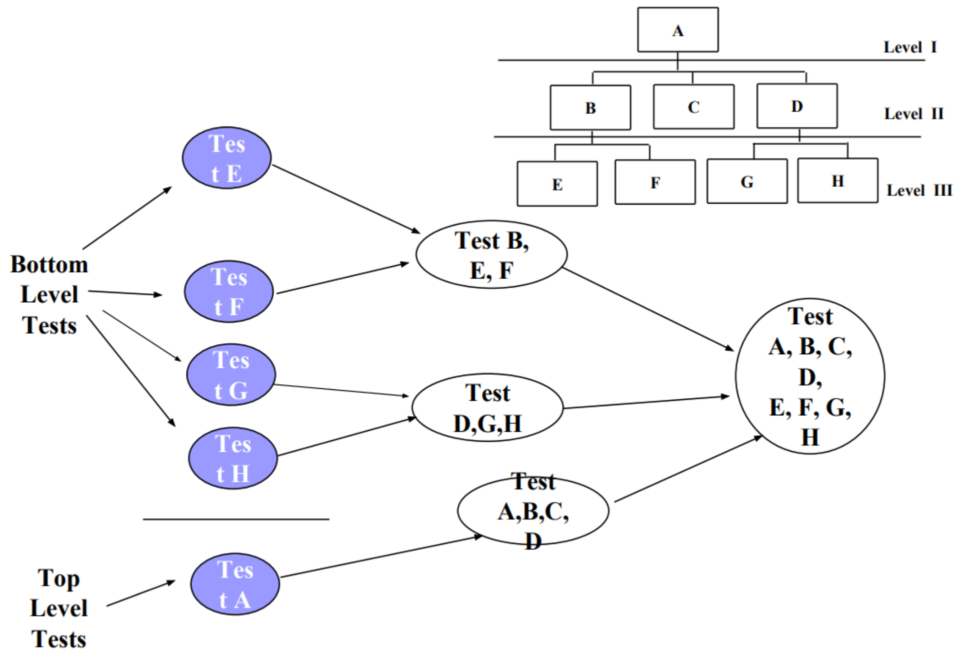
Sandwich Integration Testing

A mixed integration testing is also called sandwiched integration testing. A mixed integration testing follows a combination of top down and bottom-up testing approaches. In a top-down approach, testing can start only after the top-level module has been coded and unit tested. In the bottom-up approach, testing can start only after the bottom level modules are ready. This sandwich or mixed approach overcomes this shortcoming of the top-down and bottom-up approaches.

- *The system is viewed as having three layers*
  - □ A target layer in the middle
  - □ A layer above the target
  - □ A layer below the target
  - □ Testing converges at the target layer

- How do you select the target layer if there are more than 3 layers?
  - □ Heuristic: Try to minimize the number of stubs and drivers

Sandwich Testing Strategy



Test Sessions

■ A test session refers to one set of tests for a specific configuration of actual code and stubs

■ The number of integration test sessions using a decomposition tree can be computed

□ Sessions=nodes – leaves + edges

Call Graph Based Integration

- The basic idea is to use the call graph instead of the decomposition tree
- The call graph is a directed, labeled graph
- Two types of call graph based integration testing
    - Pairwise Integration Testing
    - Neighborhood Integration Testing

Pair-Wise Integration Testing

- The idea behind Pair-Wise integration testing is to eliminate the need for developing stubs/drivers
- The objective is to use actual code instead of stubs/drivers
- In order no to deteriorate the process to a big-bang strategy, we restrict a testing session to just a pair of units in the call graph
- The result is that we have one integration test session for each edge in the call graph

Neighborhood Integration Testing

- We define the neighborhood of a node in a graph to be the set of nodes that are one edge away from the given node
- In a directed graph means all the immediate predecessor nodes and all the immediate successor nodes of a given node
- The number of neighborhoods for a given graph can be computed as:

InteriorNodes = nodes – (SourceNodes + SinkNodes)
Neighborhoods = InteriorNodes + SourceNodes
Or
Neighborhoods = nodes – SinkNodes

- Neighborhood Integration Testing reduces the number of test sessions