UNIT –II TRANSACTION FLOW TESTING

(1) Transaction Flows:

(i) Definitions:

- A transaction is defined as a set of statements or a unit of work handled by a system user.
- A transaction consists of a sequence of operations, some of which are performed by a system, persons, or devices that are outside of the system.
- Each transaction is usually associated with an entry point and an exit point.
- The execution of a transaction begins at the entry point and ends at an exit point there by producing some results.
- After getting executed, the transaction no longer exists in the system.
- All the results are finally stored in the form of records inside the system.

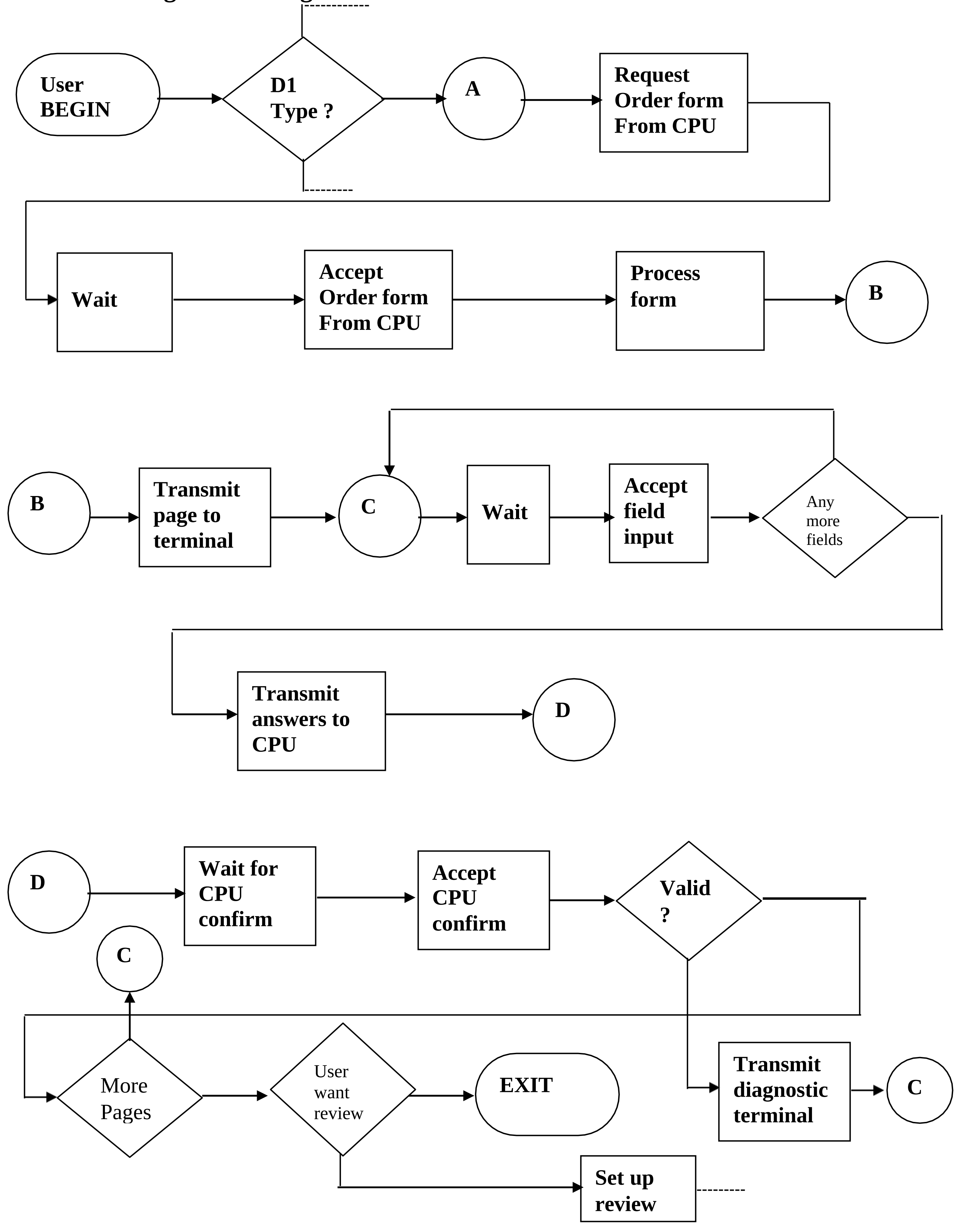
A transaction for an online information retrieval system might consist of the following steps:

1. Accept input (tentative birth).
2. Validate input (birth).
3. Transmit acknowledgment to requester.
4. Do input processing.
5. Search file.
6. Request directions from user.
7. Accept input.
8. Validate input.
9. Process request.
10. Update file.
11. Transmit output.
12. Record transaction in log and cleanup (death).

- The user processes these steps as a single transaction.
- From the system's point of view, the transaction consists of twelve steps and ten differentkinds of subsidiary tasks.
- Most online systems process many kinds of transactions.
- For example, an automatic bank teller machine can be used for withdrawals, deposits, bill payments, and money transfers.
- Furthermore, these operations can be done for a checking account, savings account,vacation account, Christmas club, and so on.
- Although the sequence of operations may differ from transaction to transaction, most transactions have common operations.
- For example, the automatic teller machine begins every transaction by validating the user'scard and password number.
- Tasks in a transaction flowgraph correspond to processing steps in a control flowgraph.
- As with control flows, there can be conditional and unconditional branches, and junctions.

(ii) Example:

- The following figure shows part of a transaction flow.
- A transaction flow is processed in Forms. Each form consists of several pages with recordsand fields in it.
- A system is taken as the terminal controller to process these form. Only those forms whichare located on a central computer are requested for processing.


User BEGIN
D1 Type ?
A
Request Order form From CPU
Wait
Accept Order form From CPU
Process form
B
B
Transmit page to terminal
C
Wait
Accept field input
Any more fields
Transmit answers to CPU
D
D
Wait for CPU confirm
Accept CPU confirm
Valid ?
C
More Pages
User want review
EXIT
Transmit diagnostic terminal
C
Set up review

- Long forms are compressed and transmitted by the central computer to minimize thenumber of records in it.
- The output of each page is transmitted by the terminal controller to the central computer.
- If the output is invalid, the central computer transmits a code to the terminal controller.
- The terminal controller in tern transmits the code to the user to check the input. At the endthe user reviews the filled out form.
- The above figure shows the processing of a transaction using forms.
  - When the transaction is to be initiated, the process p_1 requests forms from CPU.
  - The central computer accepts the form in the process p_3. p_4 process the form.
  - The characteristics of the transactions are shown by using a decision box D1 to determine whether to cancel or process further.
  - These decisions are handled by the terminal controller.
  - P_5 transmits the page to the terminal.
  - D_2 and D_4 are the decision boxes to know whether the form needs more pages or not.
  - D_3 is a decision for the structure of the form, to validate the input.
  - If necessary, the user reviews whole system in process p_{12}
  - The central computer then transmits a diagnostic code back to the terminal controllerin p_{11}. After reviewing, the transaction flow is closed and exit operation is performed.

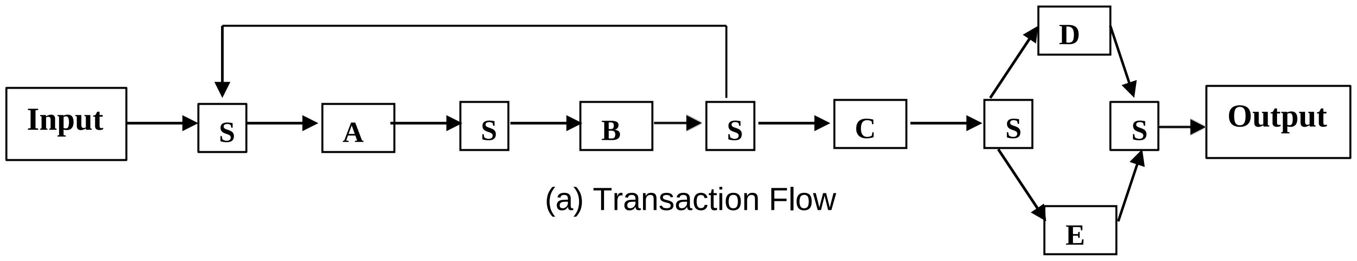
<u>(iii) Usage:</u>

- Transaction flows are indispensable for specifying requirements of complicated systems, especially online systems.
- A big system such as an air traffic control or airline reservation system has not hundreds,but thousands of different transaction flows.
- The flows are represented by relatively simple flowgraphs, many of which have a singlestraight-through path.
- An ATM system, for example, allows the user to try, say three times, and will take the cardaway the fourth time.

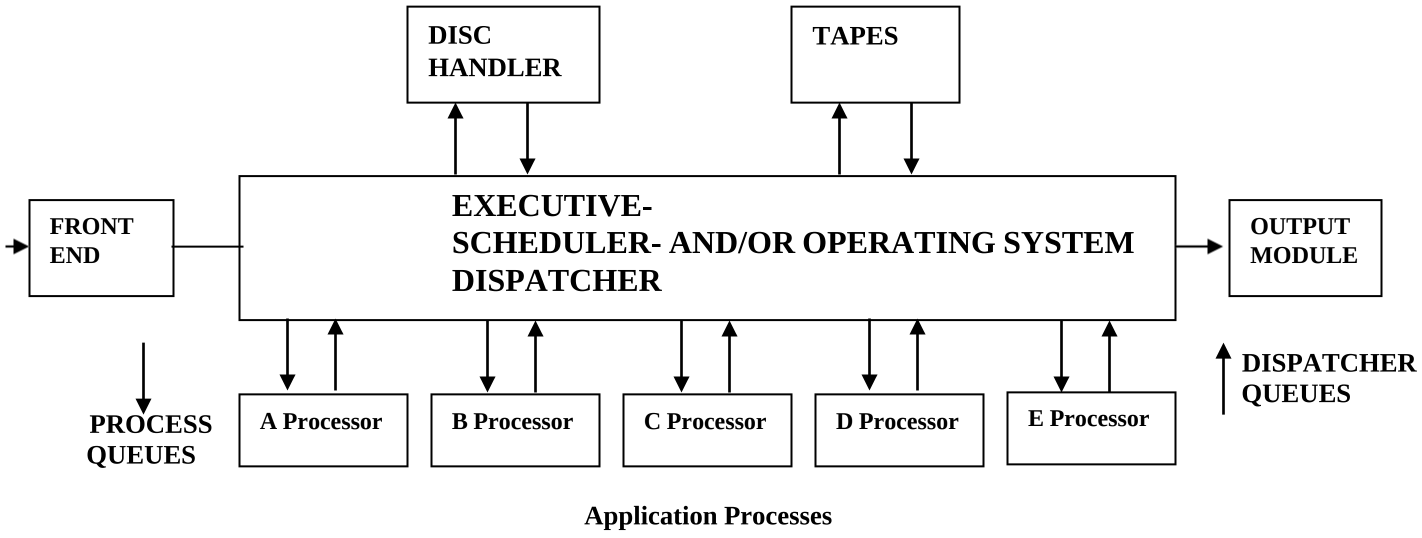
<u>(iv) Implementation:</u>

- Transaction flow has an implicit representation of system control structure.
- That is, there is no direct relation between the process and decisions.
- A transaction flow is represented by a path taken by a transaction through a succession of processing modules. These transactions are placed in a transaction-control block.
- The transactions present in that block are processed according to their flow.
- Each transaction is represented by a token and the transaction flowgraph shows a pictorial representation of these tokens.
- The transaction flowgraph is not the control structure of the program.
  - The below **figure a** shows transaction flow and corresponding implementation of a program that creates that flow.
  - This transaction goes through input processing, and then passes through process A,followed by B.
  - The result of process B may force the transaction to pass back to process A.
  - The transaction then goes to process C, then to either D or E, and finally to output processing.
  - **Figure b** is a diagrammatic representation of system control structure.
  - This system control structure is controlled either by an executive or scheduler or dispatcher operating system.
  - The links in the structure either represents a process queue or a dispatcher queue.
  - The transaction is created by placing a token on an input queue.

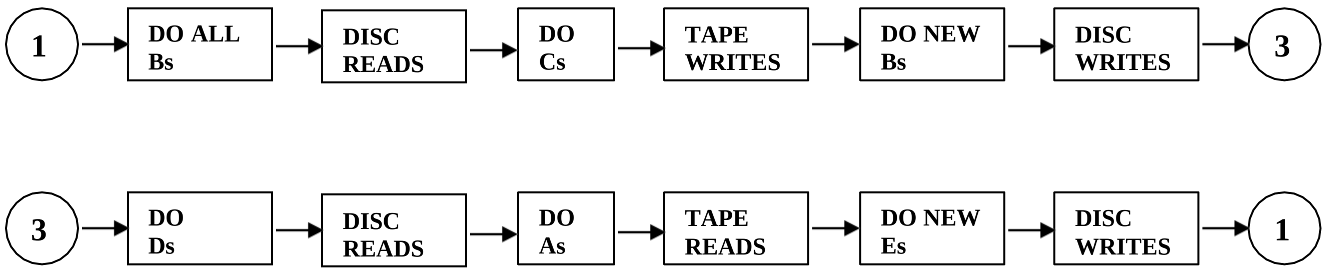
- ❖ The scheduler then examines the transaction and places it on the work queue forprocess A, but process A will not necessarily be activated immediately.
- ❖ When a process has finished working on the transaction, it places the transaction-control block back on a scheduler queue.
- ❖ The scheduler then examines the transaction control block and routes it to the nextprocess based on information stored in the block.
- ❖ The scheduler contains tables or code that routes the transaction to its next process. In systems that handle hundreds of transaction types, this information is usually storedin tables rather than as explicit code.
- ❖ Alternatively, the dispatcher may contain no transaction control data or code; the information could be implemented as code in each transaction processing module.



(a) Transaction Flow



(b) System Control Structure



(c) Executive/Dispatcher Flowchart

- Figure c shows a simplified representation of transaction flow.
- Let's say that while there could be many different transaction flows in the system, theyall used only processes A, B, C, D, E, and disc and tape reads and writes, in various combinations.
- Just because the transaction flow order is A,B,C,D,E is no reason to invoke the processes in that order.
- For other transactions, not shown, the processing order might be B,C,A,E,D. A fixed processing order based on one transaction flow might not be optimum for another.
- Furthermore, different transactions have different priorities that may require some towait for higher-priority transactions to be processed.
- Similarly, one would not delay processing for all transactions while waiting for a specific transaction to complete a necessary disc read operation.

(v) Perspective:

- There were no restrictions on how a transaction's identity is maintained: implicit, explicit, in transaction control blocks, or in task tables.
- Transaction-flow testing is the ultimate black-box technique because all we ask is that therebe something identifiable as a transaction and that the system will do predictable things to transactions.
- Transaction flowgraphs are a kind of data flowgraph.
- Data flowgraphs and control flowgraphs the most important difference is in control flowgraphs we defined a link or block as a set of instructions such that if any one of themwas executed, all (barring bugs) would be executed.
- For data flowgraphs in general, and transaction flowgraphs in particular, we change the definition to identify all processes of interest.
- Another difference to which we must be sensitive is that the decision nodes of a transaction flowgraph can be complicated processes in their own rights.

(vi) Complications:

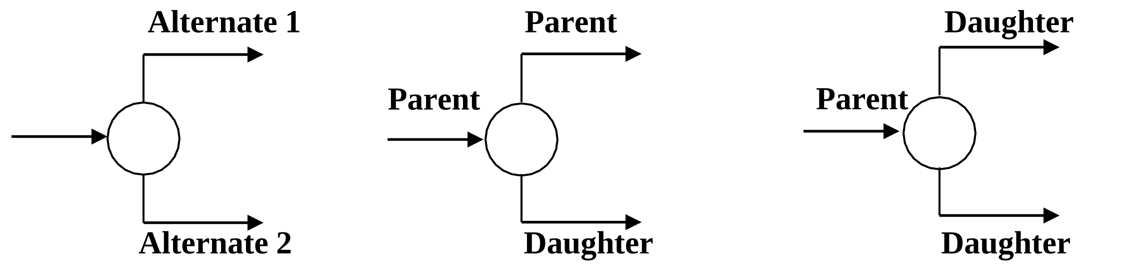
(a) General

- Transaction flows don't have a good structured design for code.
- The problems of transaction flows result in problems like error conditions, malfunctions, recovery actions etc.
- These errors are unstructured. As features are added into the transaction flows the complexity of the transaction flow increases.
- Transactions are interactions between modules. A good system design indicates thatthere is no implementation of new transaction or changing of an existing transaction.
- Hence transaction flow model results in consequences such as poor response times,security problems, inefficient processing, dangerous processing etc.
- The decision nodes of a transaction flowgraph can be complicated.
- These nodes have exists that go to central recovery processes.
- The effect of interrupts in a transaction flow model converts every process box intomany, with exit links.
- Therefore the test design is no longer fit for transaction flow model.
- Examples for the transaction flow to be imperfect.

(b) Births

- A transaction can give birth to others and can also merge with others in many of thesystems. From the time they are created to the time they are completed, transaction flows have a unique identity.

- The following figure shows three different possible interpretations of the decision nodes with two or more outlinks.

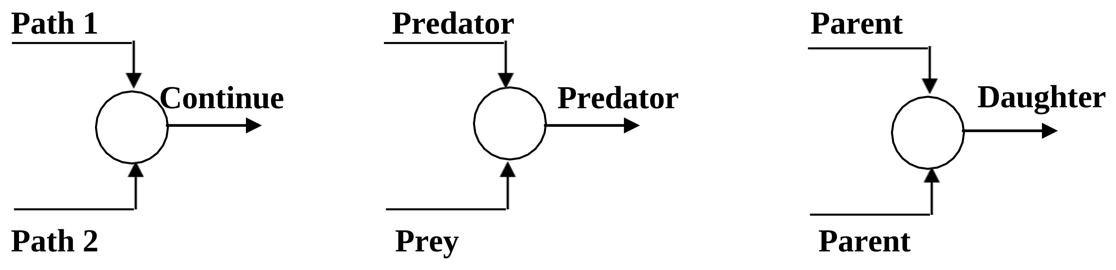


(a) Decision (b) Biosis (c) Mitosis

- In **figure a,** a transaction (Birth) has been created. The incoming transaction at decisionnode gives birth of two new transactions.
- The two transactions alternate 1 and alternate 2 has a different or same identity.
- The **figure b** shows a different situation compared to **figure a**.
- The parent transaction gives birth to two new transactions.
- One transaction has the same identity as Parent the other transaction results in adifferent identity Daughter. This situation is called Biosis.
- The figure c is similar to figure b, except that the parent transaction is destroyed andtwo new transactions (daughters) are created. This situation is called mitosis.

(c) Mergers

- Merging is as troublesome as transaction flow splitting. The two transactions are merged at decision node giving a new transaction with the same or different identity.



(a) Junction (b) Absorption (c) Conjunction

- In **figure a** path 1 and path 2 merge at a junction resulting in a single one Continue.
- The **figure b** is a predator transaction absorbs a prey. The prey is gone but the predatorretains its identity.
- The **figure c** shows a slightly different situation in which two parent transactions mergeto form a new daughter.

(d) Theoretical Status and Pragmatic Solutions (Solutions for the above examples)

- Transaction flow model doesn’t meet the requirements of multiprocessor system. Therefore a generic model called Petri is taken.
- Petri nets use operations that include explicit representation of tokens in the stages ofprocess.
- Petri net have been used to test the problems in protocol testing, network testing and soon. The application to software testing is still in its beginning stage to determine whetherit is a productive model or not.

- ❖ As long as test results are good, the imperfect model doesn't not matter because the complexities that can invalidate the model have been ignored.
- ❖ The following are some of the possible cases:

  1. ***Biosis***
     - ❖ The parent flow is followed from beginning of a transaction flow to the end of a transaction flow.
     - ❖ A new birth is treated as a new flow, either to end or to absorb that birth.
  2. ***Mitosis***
     - ❖ It begins from the parent's flow to the mitosis point. From mitosis point, an additional flow starts and get destroyed at their respective ends.
  3. ***Absorption***
     - ❖ In this situation, the parent's flow is treated as the primary flow. The parent flowis modeled from its absorption point to the point at which it gets destroyed.
  4. ***Conjugation***
     - ❖ This situation is the opposite of mitosis situation. Each parent flow is modeledfrom its birth to the conjugation point.
     - ❖ And from the conjugation point, the resulting child flow starts and get destroyed.
- ❖ Births, Mitosis, Absorptions, and conjugations are as problematic for the software designers.
- ❖ Illegal births, wrongful deaths and lost children are some of the common problems.
- ❖ Although the transaction flow is modeled by simple flowgraphs, they recognize bugswhere transactions are created, absorbed and conjugated.

(vii) Transaction flow structure:

- ❖ A sequential flow of operations is represented by a structure called a transaction flow structure.
- ❖ Even transaction flows are analogous to control flowgraphs, it is not necessary thatgood structure provided for code should also exist for transaction flows.
- ❖ Transactions flows are often considered as ill-structured due to the following reasons.
  1. It's a *model* of a process, not just code. While processing the transaction, humanscan't be forced to follow the rules of a specific software structure, as they may incorporate decisions, loops, etc
  2. Behavior of other uncontrolled systems may be incorporated by some parts of the transactional flow.
  3. Permanent ill-structured nature of the transaction flow leads to loop jumps uncontrollable GOTO statements etc. Not even a small part of the transaction flow has the ability to handle error detection, failures, malfunctioning, recovery actions etc
  4. If any new features are added and enhancements are made in transactional flows, then the complexity of each and every transaction inherently increases. For instanceone can't expect a good transaction flow from lawyers, politicians, salesman etc
  5. Basically systems are designed from specific modules and the transaction flows aredesigned or produced through the module of interaction..
  6. Modeling of interrupts, multitasking, synchronization, polling, queue disciplines arenot related to structuring..

(2) Transaction Flow Testing Techniques:

(i) Get the Transaction Flows:

- ➢ Complicated systems that process a lot of different complicated transactions should haveexplicit representations of the transaction flows, or the equivalent documented.

- The transaction flows can be mapped into programs such that the flow of transaction will becreated easily.
- The processing of the transactions is done in the design phase.
- The overview section in design phase contains the details of the transaction flows.
- Detailed transaction flows are necessary to design the system's functional test.
- Transaction flows are similar to control flow graphs where the act of getting information canbe more effective.
- Therefore the bugs can be determined. The flow of transaction in design phase is donestep by step such that the problems would not arise and a bad design can be avoided.

(ii) Transaction Flow testing:

- Transaction flow testing is a technique used in computerized applications.
- The transaction flow testing technique is used to control the documents that require theauditor to specify the following.
  - The business cycle in the flow.
  - The various types of transaction that flow through individual cycle.
  - The operations that are carried out within the cycle.
  - The objectives of internal control
  - The internal control methods used to attain each objective.
- The tester in the transaction flow testing is used to develop a flowchart. The tester tracksthe transaction flow and performs various functions in the same order as that of the transaction.
- The internal control methods are recognized at each point of the transaction flow.

(iii) Inspections, Reviews, Walkthroughs:

- Transaction flows are a natural agenda for system reviews or inspections.
- Start transaction-flow walkthroughs at the preliminary design review and continue them inever greater detail as the project progresses.

1. In conducting the walkthroughs, you should:
   a. Discuss enough transaction types (i.e., paths through the transaction flows) toaccount for 98%–99% of the transactions the system is expected to process.
   b. Discuss paths through flows in functional rather than technical terms.
   c. Ask the designers to relate every flow to the specification and to show how that transaction, directly or indirectly, follows from the requirements.
2. Make transaction-flow testing the cornerstone of system functional testing just as pathtesting is the cornerstone of unit testing. For this you need enough tests to achieve C_1 and C_2 coverage of the complete set of transaction flowgraphs.
3. Select additional transaction-flow paths (beyond $C_1 + C_2$) for loops, extreme values,and domain boundaries.
4. Select additional paths for weird cases and very long, potentially troublesome transactions with high risks and potential consequential damage.
5. Design more test cases to validate all births and deaths and to search for lost daughters, illegitimate births, and wrongful deaths.
6. Publish and distribute the selected test paths through the transaction flows as early aspossible so that they will exert the maximum beneficial effect on the project.
7. Have the buyer concur that the selected set of test paths through the transaction flowsconstitute an adequate system functional test.
8. Tell the designers which paths will be used for testing but not (yet) the details of thetest cases that force those paths.

(iii) Path Selection:

- Path selection for system testing based on transaction flows should have a distinctly different flavor from that of path selection done for unit tests based on control flowgraphs.
- Start with a covering set of tests ($C_1 + C_2$) using the analogous criteria you used for structural path testing, but don't expect to find too many bugs on such paths.
- Select a covering set of paths based on functionally sensible transactions as you would forcontrol flowgraphs.
- Confirm these with the designers.
- Try to find the most tortuous, longest, strangest path from the entry to the exit of the transaction flow. Create a catalog of these weird paths.
- This procedure is best done early in the game, while the system design is still in progress,before processing modules have been coded. The covering set of paths belongs in the system feature tests.
- It gives everybody more confidence in the system and its test.

(iv) Sensitization:

- The Good news is most of the normal paths are very easy to sensitize—80%–95% *transaction flow* coverage (C1 + C2) is usually easy to achieve.
- The bad news is that the remaining small percentage is often very difficult, if not impossible, to achieve by fair means.
- While the simple paths are easy to sensitize there are many of them, so that there's a lot oftedium in test design.
- Sensitization *is* the act of defining the transaction. If there are sensitization problems on theeasy paths, then bet on either a bug in transaction flows or a design bug.
- The reason these paths are often difficult to sensitize is that they correspond to error conditions, synchronization problems, overload responses, and other anomalous situations.

1. *Use Patches*

- ❖ The dirty system tester's best, but dangerous, friend.
- ❖ It's a lot easier to fake an error return from another system by a judicious patchthan it is to negotiate a joint test session.

2. *Mistune*

- ❖ Test in a system sized with grossly inadequate resources.
- ❖ By "grossly" I mean about 5%–10% of what one might expect to need.
- ❖ This helps to force most of the resource-related exception conditions.

3. *Break the Rules*

- ❖ Transactions almost always require associated, correctly specified, data structuresto support them.
- ❖ Often a system database generator is used to create such objects and to assurethat all required objects have been correctly specified.
- ❖ Bypass the database generator and/or use patches to break any and all rules embodied in the database and system configuration that will help you to go downthe desired path.

4.*Use Breakpoints*

- ❖ Put breakpoints at the branch points where the hard-to-sensitize path segmentbegins and then patch the transaction control block to force that path.

- You can use one or all of the above methods, and to sensitize the strange paths.
- These techniques are especially suitable for those long tortuous paths that avoid the exit.

(v) Instrumentation:

- Instrumentation plays a bigger role in transaction-flow testing than in unit path testing.

- Counters are not useful because the same module could appear in many different flowsand the system could be simultaneously processing different transactions.
- The information of the path taken for a given transaction must be kept with that transaction.
- It can be recorded either by a central transaction dispatcher (if there is one) or by the individual processing modules.
- You need a trace of all the processing steps for the transaction, the queues on which itresided, and the entries and exits to and from the dispatcher.
- In some systems such traces are provided by the operating system.
- In other systems, such as communications systems or most secure systems, a running logthat contains exactly this information is maintained as part of normal processing.

(vi) Test databases:

- About 30%–40% of the effort of transaction-flow test design is the design and maintenanceof the test database(s).
- The first error is to be unaware that there's a test database to be designed.
- The result is that every programmer and tester designs his own, unique database, which is incompatible with all other programmers' and testers' needs.
- The consequence is that every tester (independent or programmer) needs exclusive use ofthe entire system. Furthermore, many of the tests are configuration-sensitive, so there's noway to port one set of tests over from another suite.

(vii) Execution:

- If you're going to do transaction-flow testing for a system of any size, be committed to test execution automation from the start.
- If more than a few hundred test cases are required to achieve C1 + C2 transaction-flow coverage, don't bother with transaction-flow testing if you don't have the time and resources to almost completely automate all test execution.
- You'll be running and rerunning those transactions not once, but hundreds of times over theproject's life.
- Transaction-flow testing with the intention of achieving C1 + C2 usually leads to a big increase in the number of test cases.
- Without execution automation you can't expect to do it right.

DATA FLOW TESTING

(3) Basics of Data-Flow Testing:

(i) Motivation and assumptions:

(a) What is it?

- Data-flow testing is the name given to a family of test strategies based on selecting paths through the program's control flow in order to explore sequences of events relatedto the status of data objects.
- For example, pick enough paths to assure that every data object has been initializedprior to use or that all defined objects have been used for something.

(b) Motivation

- It is our belief that, just as one would not feet confident about a program without executing every statement in it as part of some test, one should not feel confident abouta program without having seen the effect of using the value produced by each and every computation.
- To the extent that we achieve the widely sought goal of reusable code, we can expectthe balance of source code statements to shift ever more toward data statement domination.

- In all known hardware technologies, memory components have been, are, and areexpected to be cheaper than processing components.

(c) New Paradigms-Data-Flow Machines

- Data flow machines are programmable computers that use packet switching communication.
- The hardware in data flow machines is optimized for data-driven execution and for finegrain parallelism.
- Data flow machines support recursion. Recursion is a mechanism used to map virtualspace to a physical space of realistic size. It is the fastest mechanism.
- The prototype in data flow machines is taken as a processing or working element.
- The overhead in data flow machines can be made acceptable by sophisticated hardware.
- There is a sufficient parallelism in many computer programs.
- The problem in data flow machine is in distribution of computation and storage of data structures.
- Another problem in data flow machines is to cease (stop) parallelism when resourcestend to get overloaded.
- Some of the data flow machines are Von Neumann machines and MIMD (multi instruction, multi data) machines.

Von Neumann machines

- The Von Neumann architecture executes one instruction at a time in the following,typical, microinstruction sequence.
  1. Fetch instruction from memory.
  2. Interpret instruction.
  3. Fetch operand(s).
  4. Process (execute).
  5. Store result (perhaps in registers).
  6. Increment program counter (pointer to next instruction).
  7. GOTO 1.
- The pure Von Neumann machine has only one set of control circuitry to interpret the instruction, only one set of registers in which to process the data, and only one execution unit (e.g., arithmetic/logic unit).
- This design leads to a sequential, instruction-by-instruction execution, which in turnleads to control-flow dominance in our thinking.
- The Von Neumann machine forces sequence onto problems that may not inherently be sequential.

MIMD (multi-instruction, multi data) machines

- MIMD machines are massively parallel machines.
- They fetch several instructions in parallel.
- Therefore they have several mechanisms for executing the above steps 1-7.
- MIMD machines can also perform arithmetic or logical operation simultaneously.
- These operations are done on different data objects.
- In these machines parallel computation is left to the compiler for processing instructions.
- For a MIMD machine, the instructions are produced in parallel flow while for a conventional machine the instructions are produced in sequential flow.
- The Parallel machine is MIMD machine with multiple processors and sequential machine is Von Neumann machine with only one processor.

(d) The Bug Assumptions

- ❖ The bug assumption for data-flow testing strategies is that control flow is generally correct and that something has gone wrong with the software so that data objects arenot available when they should be, or silly things are being done to data objects.
- ❖ Also, if there is a control-flow problem, we expect it to have symptoms that can be detected by data-flow analysis.

(ii) Data Flowgraphs:

(a) General:

- ❖ The data flowgraph is a graph consisting of nodes and directed links (i.e., links witharrows on them). The data flow is between the data objects in the data flowgraph.
- ❖ The data flowgraph not only shows the flow of data but also shows the deviation between the data objects to be implemented.

(b) Data Object State and Usage:

- ❖ Data objects can be three states i.e. created, killed and used states.
- ❖ They can be used in two distinct ways: in a calculation part and in the control flowgraphpart. The following symbols denote these possibilities.

  d—defined, created, initialized, etc.
  k—killed, undefined, released. *u*—
  used for something.
  c—used in a calculation part.
  p—used in a predicate for operation purpose.
- ❖ Every symbol in data flowgraph has a meaning. Each symbol is described below.

  1. *Defined*:
  - ❖ An object is defined explicitly when it appears in a data declaration or implicitlywhen it appears on the left-hand side of an assignment statement.
  - ❖ "Defined" can also be used to mean that a file has been opened, a dynamically allocated object has been allocated, something is pushed onto the stack, and soon.

  2. *Killed or Undefined*
  - ❖ When an object is released and is no longer in use, then it is known as a killedobject. Killed object is similar to an undefined object.
  - ❖ An object that is not available in the statement is known as Undefined object.
  - ❖ For example, a loop in FORTRAN language gets terminated when an undefinedvariable exists.
  - ❖ Another example for a killed variable is that, if an object A has been assigned a value such as A:=8 and another assignment is done for the same object A, such asA:=11 then the previous value of A (i.e. 8) is killed and redefined (i.e.11). Thereforethe value of A is 11.
  - ❖ Define and kill are complementary operations. That is, they generally come in pairsand one does the opposite of the other.

  3. *Usage*
  - ❖ A used variable is for computation (c) use and is on the right side of an assignment statement.
  - ❖ It is also used in a predicate (P) such as if z > 0, to evaluate the flow of control.
  - ❖ Hence usage variables are used both in predicate and computational purposes.

(c) Data-Flow Anomalies:

- ❖ An anomaly is a situation or condition where an object is defined but not used. For example

  IF A>0 THEN X:=1 ELSE X:= -1

```
A:= 0
A:= 0
A:= 0
A:= B + C
```

- From the above example, we notice that object A is defined trice to zero. Hence ananomaly occurs.
- There are nine possible two-letter combinations for *d, k* and *u*. Some are bugs state,some are suspicious (dangerous) state, and some are normal state.

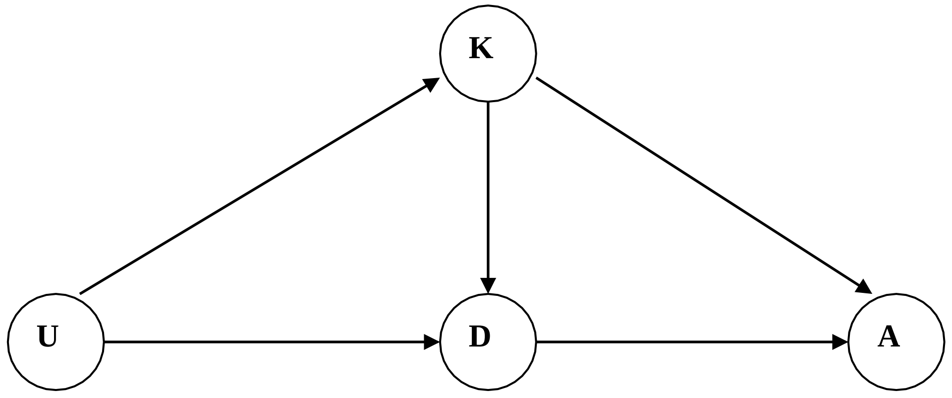
  dd—It results in a suspicious state where an object is defined twice.
  dk—results in a bug state.
  du—the normal case. The object is defined, then used. *kd*—
  normal situation. An object is killed, then redefined.*kk*—
  harmless but probably buggy.
  ku—A bug state. *ud*—
  suspicious state.*uk*—
  normal situation.*uu*—
  normal situation
- The three variables (d,k,u) show the representation of anomalous state.
- In addition to the above two-letter situations there are six single-letter situations

  -*k*: possibly anomalous.
  –*d*: okay. This is just the first definition along this path.
  –*u*: possibly anomalous. Not anomalous if the variable is global and has been previously defined.
  k–: not anomalous. The last thing done on this path was to kill the variable.
  d–: possibly anomalous.
  u–: not anomalous.
- The single-letter situations do not lead to clear data-flow anomalies but only the possibility thereof.

<u>(d) Data-Flow Anomaly State Graph :</u>

- The data flow anomaly defines an object to be in one of the following four differentstates. The states are

  K—undefined, previously killed, does not exist.D—defined but
  not in use.
  U—has been used for computation or in predicate.A—
  anomalous



- Don’t confuse these capital letters (K,D,U,A), which denote the state of the variable,with the program action, denoted by lowercase letters (*k*,*d*,*u*).
- The data flow anomaly starts in K state.
- An attempt is made to use an undefined variable. Hence it goes in an anomalous (A)state. The killed (K) state defines a variable d in defined (D) state.
- If a variable is killed from a defined (D) state then it becomes anomalous.
- The variable u is used in U state and is redefined d in D state.
- Variable k get killed in K state.

(e) Static versus Dynamic Anomaly Detection:

- Static analysis is an analysis done at compile time.
- The source code is checked and the quality is improved by removing the bugs in the program.
- Syntax errors are detected in static analysis.
- To improve the quality of a document, the document is analyzed and checked by a tool.
- If a problem, such as a data-flow anomaly, can be detected by static analysis methods,then it does not belong in testing—it belongs in the language processor.
- Static analysis tools are typically used by tools.
- Static analysis is done in design phases so that the whole model can be analyzed andthe inconsistencies can be detected.
- Static analysis can be used in the detection of security problem.
- Dynamic analysis is done at run time. Dynamic analysis detects anomalous situations atrun time with some of the data structures like Arrays, Pointers, Records etc..

1. ***Dead Variables***
   - Although it is often possible to prove that a variable is dead or alive at a givenpoint in the program, the general problem is unsolvable.
2. ***Arrays***
   - Arrays are problematic in that the array is defined or killed as a single object, but reference is to specific locations within the array.
   - Array pointers are usually dynamically calculated, to know whether the valuesare within the boundary range or out of boundary range.
3. ***Records and Pointers***
   - The array problem and the difficulty with pointers is a special case of multipartdata structures.
   - We have the same problem with records and the pointers to them.
   - In the case of records, files are created and the names of such files are dynamically known.
   - Without execution there is no way to determine the state of such objects.
4. ***Dynamic Subroutine or Function Names in a Call***
   - A subroutine or function name is a dynamic variable in a call. What is passed, or a combination of subroutine names and data objects, is constructed on a specificpath.
   - There’s no way, without executing the path, to determine whether the call iscorrect or not.
5. ***False Anomalies***
   - Anomalies don’t occur when the path of objects is not completed.
   - Such anomalies are false anomalies. The problem of identifying whether a pathis completed or not is not solved.

6. ***Recoverable Anomalies and Alternate State Graphs***
    - ❖ What constitutes an anomaly depends on context, application, and semantics.
    - ❖ Huang provided two anomaly state graphs
7. ***Concurrency, Interrupts, System Issues***
    - ❖ Anomalies become more sophisticated while moving from single processor surroundings to multi processors environment.
    - ❖ The main purpose or task of interrupt is to develop correct anomalous which iseven performed in true concurrency or pseudo concurrency.
    - ❖ The objective of system integration testing is to detect data flow anomalies at runtime that was not possible using context level testing.

- ❖ Although static analysis methods have limits, they are worth using and a continuing trend in language processor design has been better static analysis methods, especiallyfor data flow anomaly detection.
- ❖ That's good because it means there's less for us to do as testers and we have far toomuch to do as it is.

(f) Anomaly detection & types of data flow anomalies:

- ❖ An anomaly is a term that leads to inconsistency in the data flow analysis.
- ❖ The data flow is referred to as reading variables and data flow anomaly is referred to asreading variables without having an idea that the value of the variable is in use or not.
- ❖ During data flow analysis, every variable is referred to and inspected.
- ❖ There are different variables in data flow analysis.
- ❖ They are classified as

S.No	Variables	Definition
1	Defined (d)	Value assigned to a variable
2	Referenced (r)	Value read or used by a variable
3	Undefined (u)	Variable that has no defined value

- ❖ Depending on these variables, three different data flow anomalies are distinguished.They are
    1. ur-anomaly
    2. du-anomaly
    3. dd-anomaly

1. **ur-anomaly:**
    - ❖ During data flow analysis if the undefined value of a variable (u) is read (r) then it is known as a ur-anomaly.
2. **du-anomaly:**
    - ❖ A defined (d) variable becomes invalid or undefined (u) variable when a variable is not used within a particular time.
3. **dd-anomaly:**
    - ❖ This anomaly occurs when the variable accepts a value at the second assignment (d) and the first assignment value had not been used.
    - ❖ This situation occurs in dd-anomaly. For example if A:=7,A:=11 then it accepts A:=11.

- ❖ Depending on the usage of variables the anomalies can be detected.
- ❖ For example consider c++example The example shows an exchange of values of thevariables A and B with the help of another variable get if the value of the variable A isgreater than the value of the variable B.

```
void exchange(int &A,int &B)
{
    int get;
    if(A>B
    )
    {
        B=get;
        B=A;
        get=A;
    }
}
```

- ❖ The detection of anomalies are
  1. **ur-anomaly:**
     - ❖ In the above example, the variable get is used on the right side of an assignment.
     - ❖ The variable get has an undefined value because it is not initializedwhere it is declared.
     - ❖ This undefined variable is being read or referred to and hence it results inur-anomaly.
  2. **dd-anomaly:**
     - ❖ The variable B is used twice on the left side of an assignment.
     - ❖ The first assignment value becomes invalid or unused and the second assignment value is taken or used.
     - ❖ Therefore the unused variable B of the first assignment results in dd-anomaly
  3. **du-anomaly:**
     - ❖ The variable get has a defined value in the last assignment. The definedvariable cannot be used anywhere in the function because only those variables are valid which are inside the function.
     - ❖ Therefore the unused variable results in du-anomaly.

<u>(iii)</u> <u>The Data-Flow Model:</u>

<u>(a)</u> <u>General:</u>

- ❖ Our data-flow model is based on the program's control flowgraph—don't confuse thatwith the program's data flowgraph.
- ❖ So Data-flow model is considered as the heart of programs control flowgraph.
- ❖ It consists of links which are denoted by symbols d,k,u,c,p or a sequence of the symbolslike dd, du, ddd etc.
- ❖ This sequence specifies the sequential flow of data operations on the link with respectto the given variable.
- ❖ These symbols are called link weights as each link is assigned with weights (d,k,u,c,p).
- ❖ For all variables and array elements, different set of link weights exist.

  The symbols are defined as
  d= Defined object , k=Killed object, u=Used object
  c=Object for calculation purpose, p=predicate

<u>(b)</u> <u>Components of the model:</u>

- ❖ Here are the modeling rules.

  1. To every statement there is a node, whose name (number) is unique.
  Every node has at least one outlink and at least one inlink except exit nodes, which donot have outlinks, and entry nodes, which do not have inlinks.

2. Exit nodes are dummy nodes placed at the outgoing arrowheads of exit statements(e.g., END, RETURN), to complete the graph. Similarly, entry nodes are dummy nodesplaced at entry statements (e.g., BEGIN) for the same reason.

3.Another components is simple statements. These are the statements with only one outlink. The weight of simple statement is determined by sequential actions of data-flowwith respect to the given statement.

For example, consider a simple statement A:= A + B in most languages is weightedby *cd* or possibly *ckd* for variable A.

4. Predicate nodes (e.g., IF-THEN-ELSE, DO WHILE, CASE) are weighted with the *p*-use(s) on *every* outlink, appropriate to that outlink.

5. Every sequence of simple statements (e.g., a sequence of nodes with one infink and one outlink) can be replaced by a pair of nodes that has, as weights on the link betweenthem, the concatenation of link weights.

6. If there are several data-flow actions on a given link for a given variable, then theweight of the link is denoted by the sequence of actions on that link for that variable.

7. If multiple data-flow actions are available on a link for a variable, then its corresponding weight is determined by the sequence of actions. Inversely a sequenceof equivalent links are used to replace the link with more data flow actions.

(c) Putting it together:

- ❖ The following **figure a** shows the control flowgraph. The **figure b** shows this control flowgraph annotated for variables X and Y data flows.
- ❖ The **figure c** shows the same control flowgraph annotated for variable Z. Z is first defined by an assignment statement on the first link.
- ❖ Z is used in a predicate (Z >= 0?) at node 3, and therefore both outlinks of that node—(3,4) and (3,5)—are marked with a *p*. The data-flow annotation for variable V is shown in **figure d**.

(4) Strategies in Data-Flow Testing:

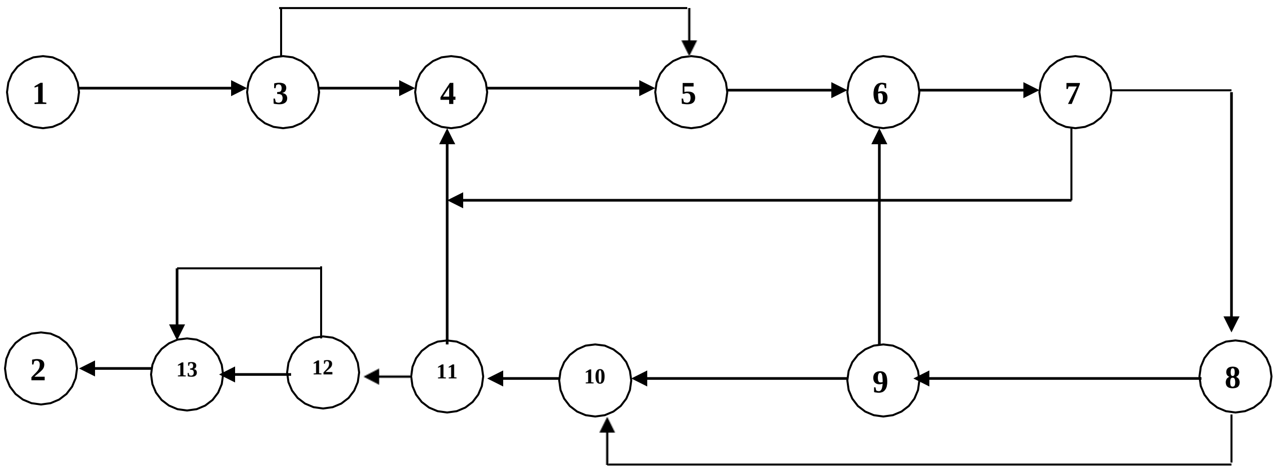
(i) General:

- ➢ Data-flow testing strategies are structural strategies.
- ➢ Data-flow testing strategies are based on the program's control flowgraph.
- ➢ Data-flow testing strategies are based on selecting test path segments (also called subpaths) that satisfy some characteristic of data flows for all data objects. For example, allsubpaths that contain a *d* (or *u, k, du, dk*).
- ➢ These strategies differ in determining whether the paths of a given type are required or onlyone path of that type is required.
- ➢ The test set includes the predicate uses and computational uses of variables.
- ➢ This usage also differs in the test set that is either computational use or predicate use ofvariables.

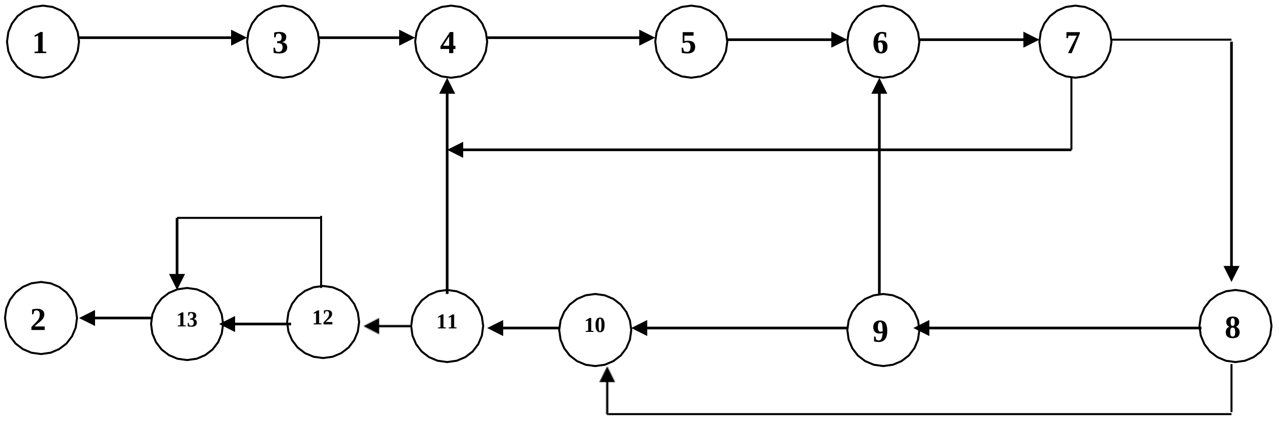
(ii) Terminology:

- ➢ We'll assume for the moment that all paths are achievable. Some terminology.
- ➢ A definition-clear path segment
    - ❖ A path segment is a sequence of connected links between nodes. This first link ofthe path is defined and the subsequent link of that path is killed.
    - ❖ A definition-clear path segment is a connected sequence of links such that X is (possibly) defined on the first link and not redined or killed on any subsequent link ofthat segment.
    - ❖ All paths in figure b are definition clear because variables X and Y are defined onlyon the first link (1,3) and thereafter. Similarly for variable V in figure d.

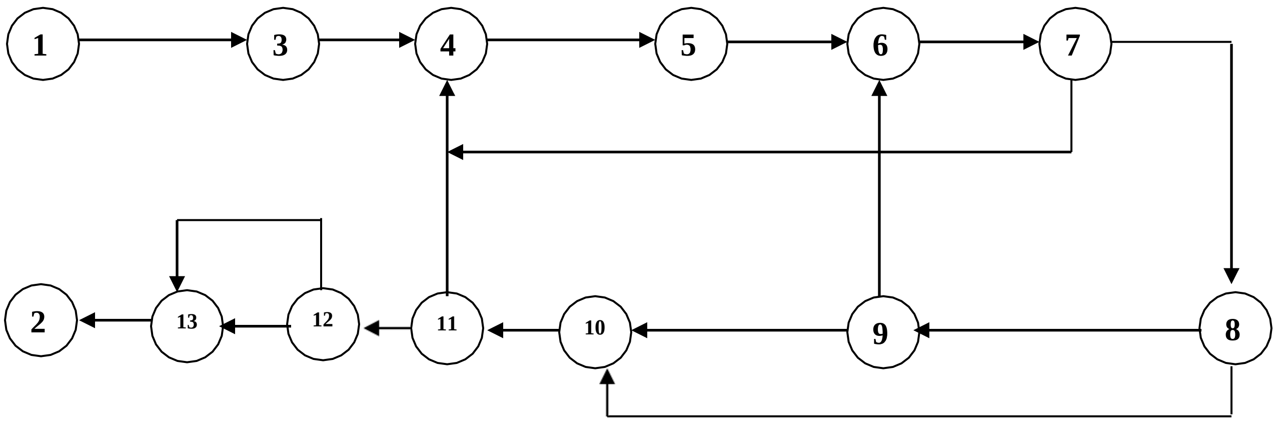
- In Figure c we have a more complicated situation. The following path segments are definition-clear: (1,3,4), (1,3,5), (5,6,7,4), (7,8,9,6,7), (7,8,9,10), (7,8,10), (7,8,10,11).
- Subpath (1,3,4,5) is not definition-clear because the variable is defined on (1,3) andagain on (4,5).
- For practice, try finding all the definition-clear subpaths for this routine (i.e., for all variables).



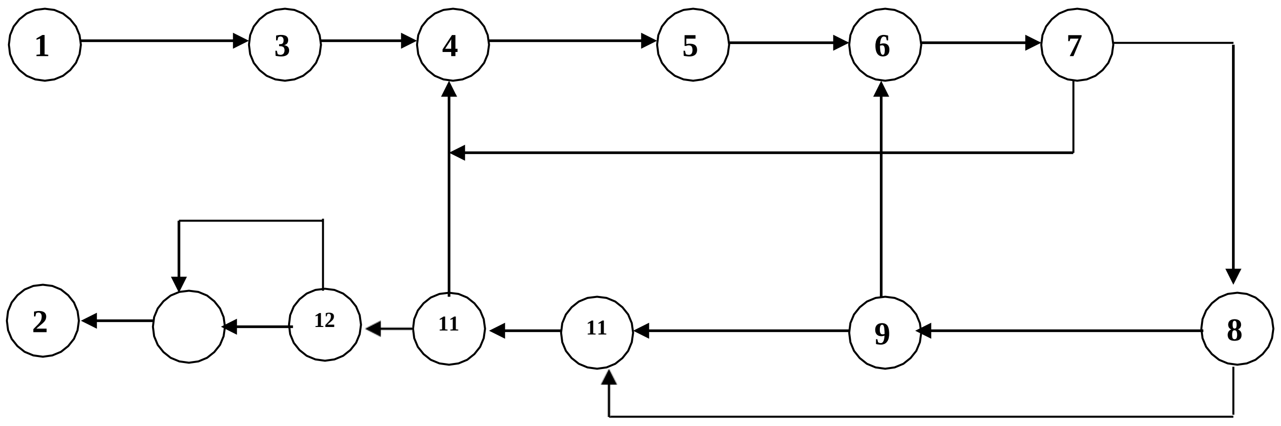
(a) Unannotated Control Flowgraph



(b) Control Flowgraph Annotated for X and Y Data Flows.



(c) Control Flowgraph Annotated for Z Data Flow



(d) Control Flowgraph Annotated for V Data Flow

- ❖ The fact that there is a definition-clear subpath between two nodes does not imply that all subpaths between those nodes are definition-clear; in general, there are many subpaths between nodes, and some could have definitions on them and somenot.
- ❖ A definition clear sub path does not include loops. For example a loop consists of(i,j) and (j,i) links.
- ❖ These links have a definition on (i,j) and a computational use on (j,i). If we include loops in a path by definition-clear path segment then there is no need to go aroundsuch path.
- ❖ Because of this the testing strategies will have a finite number of test paths.
- ❖ The strategies must be weaker than the paths because a bug can be createdwhenever a loop has been traversed and iterated.

2. <u>A loop-free path segment</u>

- ❖ A loop-free path segment is a path segment for which every node is visited at mostonce.
- ❖ Path (4,5,6,7,8,10) in figure c is loop free, but path (10,11,4,5,6,7,8,10,11,12) is notbecause nodes 10 and 11 are each visited twice.

<u>simple path segment</u>

- ❖ A simple path segment is a path segment in which at most one node is visited twice.
- ❖ For example in figure c (7,4,5,6,7) is a simple path segment.
- ❖ A simple path segment is either loop-free or if there is a loop, only one node is involved.

<u>du path</u>

- ❖ A du path from node *i* to *k* is a path segment such that if the last link has a computational use of X then the path is simple and definition-clear path.
- ❖ if the penultimate node is *j*—that is, the path is (*i*,*p*,*q*,...,*r*,*s*,*t*,*j*,*k*) and link (*j*,*k*) has a predicate use—then the path from *i* to *j* is both loop-free and definition-clear.

<u>(iii)</u> <u>The Strategies:</u>

<u>(a)</u> <u>Overview:</u>

- ❖ The structural test strategies are based on the program's control flowgraph.
- ❖ These strategies differ in determining whether the paths of a given type are required oronly one path of that type is required.
- ❖ The test set includes the predicate uses and computational uses of variables.

- This usage also differs in the test set that is either computational use or predicate use ofvariables.
- The different data flow testing strategies are given below.

(b) All-du Paths (ADUP) strategy:

- The all-*du*-paths (ADUP) strategy is the strongest data-flow testing strategy discussedhere. It requires that *every du* path from *every* definition of *every* variable to *every* use of that definition be exercised under some test.
- In the above figure b variables X and Y are used only on link (1,3), any test that starts atthe entry satisfies this criterion (for variables X and Y, but not for all variables as required by the strategy).
- The situation for variable Z in figure c is more complicated because the variable is redefined in many places. For the definition on link (1,3) we must exercise paths that include subpaths (1,3,4) and (1,3,5). The definition on link (4,5) is covered by any paththat includes (5,6), such as subpath (1,3,4,5,6, ...).
- The (5,6) definition requires paths that include subpaths (5,6,7,4) and (5,6,7,8).
- Variable V in figure d is defined only once on link (1,3).
- Because V has a predicate use at node 12 and the subsequent path to the end must beforced for both directions at node 12, the all-*du*-paths strategy for this variable requires that we exercise all loop-free entry/exit paths and at least one path that includes the loop caused by (11,4).
- Note that we must test paths that include both subpaths (3,4,5) and (3,5) even thoughneither of these has V definitions.
- They must be included because they provide alternate *du* paths to the V use on link (5,6). Although (7,4) is not used in the test set for variable V, it will be included in thetest set that covers the predicate uses of array variable V() and U.
- The all-*du*-paths strategy *is* a strong criterion, but it does not take as many tests as itmight seem at first because any one test simultaneously satisfies the criterion for several definitions and uses of several different variables.

(c) All-uses Strategy:

- Just as we reduced our ambitions by stepping down from all paths (P_∞) to branch coverage (P_2), say, we can reduce the number of test cases by asking that the test setinclude *at least one* path segment from every definition to every use that can be reached by that definition—this is called the all-uses (AU) strategy.
- The strategy is that *at least one* definition-clear path from *every* definition of *every* variable to *every* use of that definition be exercised under some test.
- In figure d, ADUP requires that we include subpaths (3,4,5) and (3,5) in some test because subsequent uses of V, such as on link (5,6), can be reached by either alternative. In AU either (3,4,5) or (3,5) can be used to start paths, but we don’t have touse both.
- Similarly, we can skip the (8,10) link if we’ve included the (8,9,10) subpath.

(d) All-p-Uses/Some-c-Uses and All-c-Uses/Some-p-Uses Strategies:

- Weaker criteria require fewer test cases to satisfy. We would like a criterion that is stronger than P_2 but weaker than AU.
- Therefore, select cases as for All (Section 3.3.3) except that if we have a predicate use,then (presumably) there’s no need to select an additional computational use (if any). More formally, the all-*p*-uses/some-*c*-uses (APU+C) strategy is defined as follows: for every variable and every definition of that variable, include at least one definition-free path from the definition to every predicate use; if there are definitions of the variable that

are not covered by the above prescription, then add computational-use test cases asrequired to cover every definition.

- The all-*c*-uses/some-*p*-uses (ACU+P) strategy reverses the bias: first ensure coverageby computational-use cases and if any definition is not covered by the previously selected paths, add such predicate-use cases as are needed to assure that every definition is included in some test.
- In figure b for variables X and Y, any test case satisfies both criteria because definition and uses occur on link (1,3). In figure c, for APU+C we can select paths that all take theupper link (12,13) and therefore we do not cover the *c*-use of Z: but that's okay according to the strategy's definition because every definition is covered.
- Links (1,3), (4,5), (5,6), and (7,8) must be included because they contain definitions forvariable Z. Links (3,4), (3,5), (8,9), (8,10), (9,6), and (9,10) must be included because they contain predicate uses of Z.
- Find a covering set of test cases under APU+C for all variables in this example—it onlytakes two tests. In figure d, APU+C is achieved for V by (1,3,5,6,7,8,10,11,4,5,6,7,8,10,11,12[upper], 13,2) and (1,3,5,6,7,8,10,11,12[lower], 13,2). Note that the *c*-use at (9,10) need not be included under the APU+C criterion.
- The figure d shows a single definition for variable V. C-use coverage is achieved by (1,3,4,5,6,7,8,9,10,11,12,13,2). In figure c, ACU+P coverage is achieved for Z by path (1,3,4,5,6,7,8,10, 11,12,13[lower], 2), but the predicate uses of several definitions are not covered. Specifically, the (1,3) definition is not covered for the (3,5) *p*-use, the (7,8)definition is not covered for the (8,9), (9,6) and (9, 10) *p*-uses.
- The above examples imply that APU+C is stronger than branch coverage but ACU+Pmay be weaker than, or incomparable to, branch coverage.

(e) All definitions Strategy:

- The all-definitions (AD) strategy asks only that every definition of every variable be covered by at least one use of that variable, be that use a computational use or a predicate use.
- Path (1,3,4,5,6,7,8, . . .) satisfies this criterion for variable Z, whereas any entry/exit pathsatisfies it for variable V. From the definition of this strategy we would expect it to be weaker than both ACU+P and APU+C.

(f) All-Predicate Uses, All-Computational Uses Strategies:

- The all-predicate-uses (APU) strategy is derived from the APU + C strategy by droppingthe requirement that we include a *c*-use for the variable if there are no *p*-uses for the variable following each definition.
- Similarly, the all-computational-uses (ACU) strategy is derived from ACU+P by droppingthe requirement that we include a *p*-use if there are no *c*-use instances following a definition.
- It is intuitively obvious that ACU should be weaker than ACU+P and that APU should beweaker than APU+C.

(g) Ordering the Strategies:

- The below figure compares path-flow and data-flow testing strategies. The arrows denote that the strategy at the arrow's tail is stronger than the strategy at the arrow'shead.
- The right-hand side of this graph, along the path from "all paths" to "all statements" isthe more interesting hierarchy for practical applications.
- Variations of data-flow strategies exist, including different ways of characterizing thepaths to be included and whether or not the selected paths are achievable.

- ❖ The strength relation graph of the above figure can be substantially expanded to fit almost all such strategies into it. Indeed, one objective of testing research has been toplace newly proposed strategies into the hierarchy.

$$\begin{tikzcd}
 & \text{ALL PATHS} \arrow[d] & \\
 & \text{ALL du PATHS} \arrow[d] & \\
 & \text{ALL USES} \arrow[dl] \arrow[dr] & \\
\text{ALL-c/SOME-p} \arrow[d] \arrow[dr] & & \text{ALL-p/SOME-c} \arrow[dl] \arrow[d] \\
\text{ALL-c USES} & \text{ALL DEFS} & \text{ALL-p USES} \arrow[d] \\
 & & \text{BRANCH} \arrow[d] \\
 & & \text{STATEMENT}
\end{tikzcd}$$

(iv) Slicing, Dicing, Data Flow and Debugging:

(a) General:

- ❖ Slicing is a program originally developed for conventional languages.
- ❖ It helps in understanding data flow and debugging techniques. The Slicing is donebased on variable sharing.
- ❖ Dicing and debugging are the concepts related to removal of unwanted bugs.

(b) Slices and Dices:

- ❖ There are two types of slicing technique. i.e. Static slicing & dynamic slicing.
- ❖ Static slicing is a part of a program defined with respect to a given variable X and a statement *i*:
- ❖ It consists of all statements that could affect the value of X at statement *i*.
- ❖ The result of a false statement effect in an improper computational use or predicate useof some other variable.
- ❖ If the variable X is correct then the bug is detected in the program itself.
- ❖ A program dice is a part of a slice in which the statements which are correct has beenremoved.
- ❖ The idea behind slicing and dicing is based on Weiser's observation that these constructs are at the heart of the procedure followed by good debuggers.
- ❖ Dynamic slicing is a refinement of static slicing. Dynamic slicing compares the data flow relationship with respect to static data flows.
- ❖ Dicing is defined as the process of refining slice by removing all the unwanted bug statements in a program.
- ❖ Basically a dice is generated from a slice which posses the information about testing or debugging the function of a dice is to improve or refine a slice by removing the unwanted statements from a program.
- ❖ The process of dicing is often employed by debuggers. The current methods of dicing encompass assumptions related to bugs and programs.
- ❖ Due to the existence of bugs the usage of real program is declined.

(c) Data-flow:

- ❖ Data flow is defined as the process of reading variables. The central concept of data-flow is to bridge the gap between debugging and testing.
- ❖ The idea of slices was extended to arrays and data vectors and the data-flow relations(such as dc and dp) in dynamic slices are analogous compared to the data-flow relations in static slices (dc and dp).
- ❖ Where dc and dp are the data objects. Here
  d=Object definition,
  c=Computation
  p=Symbol used in a predicate for operation purpose.

(d) Debugging:

- ❖ Debugging is defined as an iterative method in which refinement of slices is carried outthrough dices so as to obtain the dicing information.
- ❖ Basically debugging is carried out after a test case is successfully executed.
- ❖ The process of debugging terminates when all the bugs that exists in the program statements are corrected.
- ❖ Methods of slicing leads to commercial testing or development of different debuggingtools.
- ❖ The test cases involved in integration and testing are modeled for efficient error detection, where as the cases involved in debugging are modeled for efficient errorisolation.

(5) Application of Data-Flow Testing:

- ➢ Data flow testing is used to detect the different abnormalities that may arise due to dataflow anomalies.
- ➢ Data flow testing shows the relationship between the data objects that represents data.
- ➢ Data flow testing strategies help in determining the usage of variables that are included inthe test set.
- ➢ Data flow testing is cost effective.
- ➢ Data flow testing solves the problems that are encountered while performing.
- ➢ Data flow testing uses practical applications rather than mathematical applications.
- ➢ Data flow testing is used in developing web applications with Java technology.

What is Data Flow Testing? Application, Examples and Strategies

Data Flow Testing is a type of structural testing. It is a method that is used to find the test paths of a program according to the locations of definitions and uses of variables in the program. It has nothing to do with data flow diagrams.
It is concerned with:

- Statements where variables receive values,
- Statements where these values are used or referenced.

To illustrate the approach of data flow testing, assume that each statement in the program assigned a unique statement number. For a statement number S-

DEF(S) = {X | statement S contains the definition of X}

USE(S) = {X | statement S contains the use of X}

If a statement is a loop or if condition then its DEF set is empty and USE set is based on the condition of statement s.

Data Flow Testing uses the control flow graph to find the situations that can interrupt the flow of the program.
Reference or define anomalies in the flow of the data are detected at the time of associations between values and variables. These anomalies are:

- A variable is defined but not used or referenced,
- A variable is used but never defined,
- A variable is defined twice before it is used

Advantages of Data Flow Testing:
Data Flow Testing is used to find the following issues-

- To find a variable that is used but never defined,
- To find a variable that is defined but never used,
- To find a variable that is defined multiple times before it is use,
- Deallocating a variable before it is used.

Disadvantages of Data Flow Testing

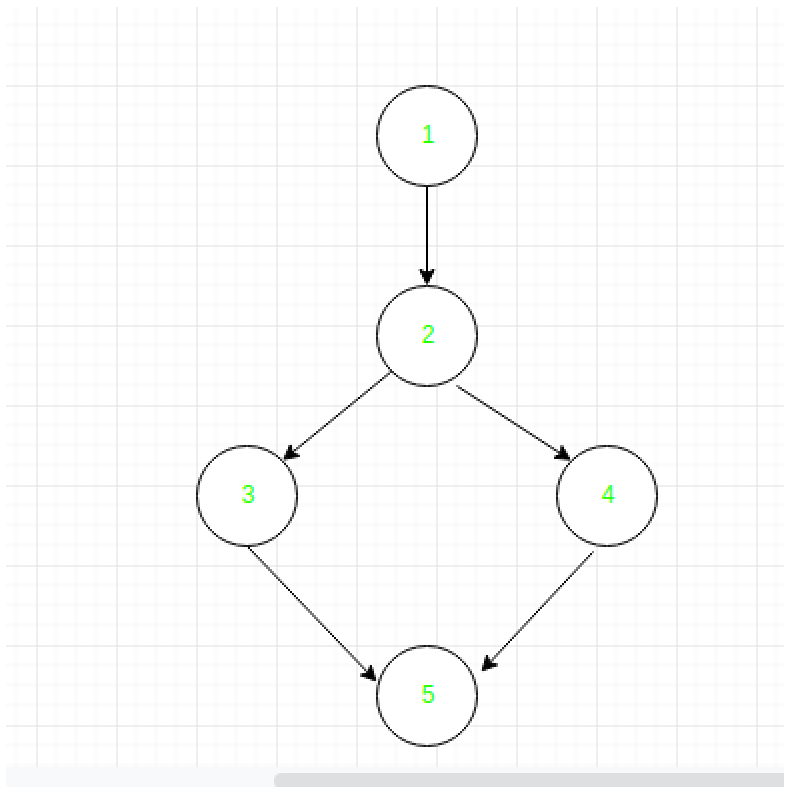
- Time consuming and costly process
- Requires knowledge of programming languages

Example:

```
1. read x, y;
2. if(x>y)
3. a = x+1
```

```
else
4. a = y-1
5. print a;
```

Control flow graph of above example:



Define/use of variables of above example:

Variable	Defined at node	Used at node
x	1	2, 3
y	1	2, 4
a	3, 4	5

What is Black Box Testing?

Black Box Testing is also known as behavioral, opaque-box, closed-box, specification-based or eye-to-eye testing.

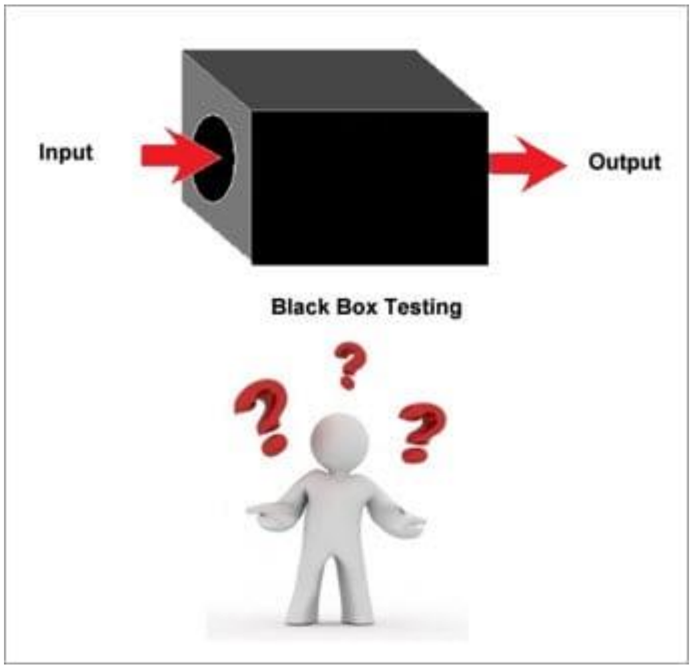
It is a Software Testing method that analyzes the functionality of a software/application without knowing much about the internal structure/design of the item that is being tested and compares the input value with the output value.

The main focus of Black Box Testing is on the functionality of the system as a whole. The term **'Behavioral Testing'** is also used for Black Box Testing.
Behavioral test design is slightly different from the black-box test design because the use of internal knowledge isn't strictly forbidden, but it's still discouraged. Each testing method has its own advantages and disadvantages. There are some bugs that cannot be found using black box or white box technique alone.

A majority of the applications are tested using the Black Box method. We need to cover the majority of test cases so that most of the bugs will get discovered by the Black-Box method.

This testing occurs throughout the Software Development and Testing Life Cycle i.e in Unit, Integration, System, Acceptance, and Regression Testing stages.

This can be either Functional or Non-Functional.



Types of Black Box Testing

Practically, there are several types of Black Box Testing that are possible, but if we consider a major variant of it then only the below mentioned are the two fundamental ones.

#1) Functional Testing

This testing type deals with the functional requirements or specifications of an application. Here, different actions or functions of the system are being tested by providing the input and comparing the actual output with the expected output.

For example, when we test a Dropdown list, we click on it and verify if it expands and all the expected values are showing in the list.

Few major types of Functional Testing are:

- Smoke Testing
- Sanity Testing
- Integration Testing
- System Testing
- Regression Testing
- User Acceptance Testing

=> ***Read More on Functional Testing***

#2) Non-Functional Testing

Apart from the functionalities of the requirements, there are even several non-functional aspects that are required to be tested to improve the quality and performance of the application.

Few major types of Non-Functional Testing include:

- Usability Testing
- Load Testing
- Performance Testing
- Compatibility Testing
- Stress Testing
- Scalability Testing

=> Read More on Non-Functional Testing

Black Box Testing Techniques

In order to systematically test a set of functions, it is necessary to design test cases. Testers can create test cases from the requirement specification document using the following Black Box Testing techniques:

- Equivalence Partitioning
- Boundary Value Analysis
- Decision Table Testing
- State Transition Testing
- Error Guessing
- Graph-Based Testing Methods
- Comparison Testing

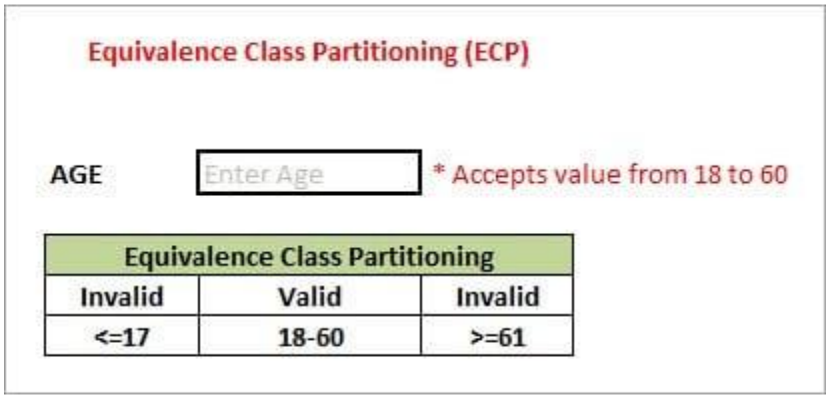
Let's understand each technique in detail.

#1) Equivalence Partitioning

This technique is also known as Equivalence Class Partitioning (ECP). In this technique, input values to the system or application are divided into different classes or groups based on its similarity in the outcome.

Hence, instead of using each and every input value, we can now use any one value from the group/class to test the outcome. This way, we can maintain test coverage while we can reduce the amount of rework and most importantly the time spent.

For Example:


Equivalence Class Partitioning (ECP)
AGE
Enter Age
* Accepts value from 18 to 60

Equivalence Class Partitioning		
Invalid	Valid	Invalid
<=17	18-60	>=61

As present in the above image, the "AGE" text field accepts only numbers from 18 to 60. There will be three sets of classes or groups.

Two invalid classes will be:
a) Less than or equal to 17.

b) Greater than or equal to 61.

A valid class will be anything between 18 and 60.

We have thus reduced the test cases to only 3 test cases based on the formed classes thereby covering all the possibilities. So, testing with any one value from each set of the class is sufficient to test the above scenario.

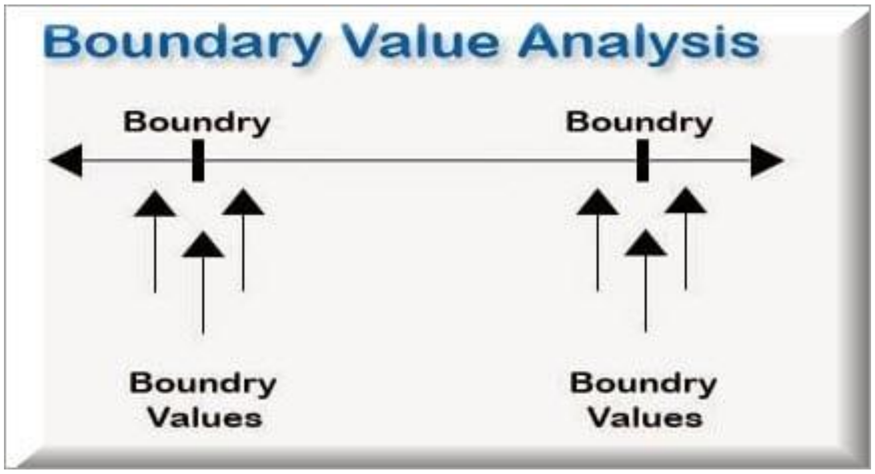
Recommended Read* => *What is Equivalence Partitioning?

#2) Boundary Value Analysis

The name itself defines that in this technique, we focus on the values at boundaries as it is found that many applications have a high amount of issues on the boundaries.

Boundary refers to values near the limit where the behavior of the system changes. In boundary value analysis, both valid and invalid inputs are being tested to verify the issues.

For Example:



If we want to test a field where values from 1 to 100 should be accepted, then we choose the boundary values: 1-1, 1, 1+1, 100-1, 100, and 100+1. Instead of using all the values from 1 to 100, we just use 0, 1, 2, 99, 100, and 101.

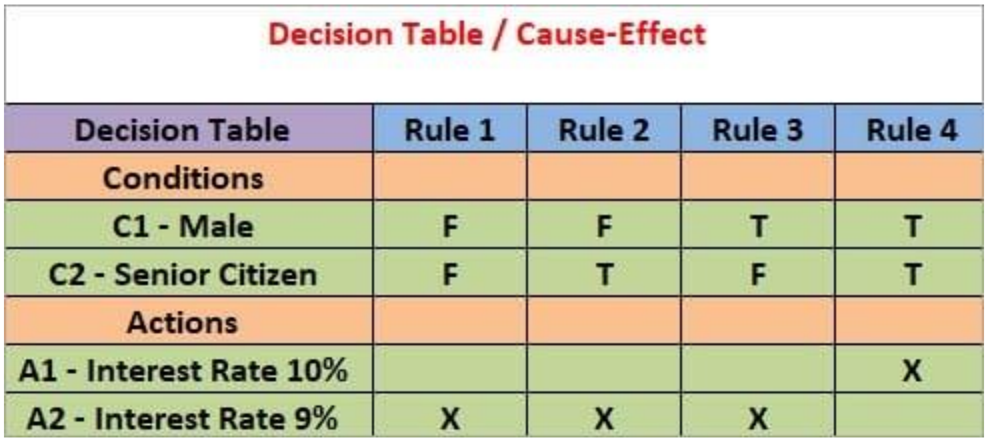
#3) Decision Table Testing

As the name itself suggests, wherever there are logical relationships like:

```
If
{
(Condition = True)
then action1 ;
}
else action2; /*(condition = False)*/
```

Then a tester will identify two outputs (action1 and action2) for two conditions (True and False). So based on the probable scenarios a Decision table is carved to prepare a set of test cases.

<u>For Example:</u>
Take an example of XYZ bank that provides an interest rate for the Male senior citizen as 10% and 9% for the rest of the people.



Decision Table / Cause-Effect

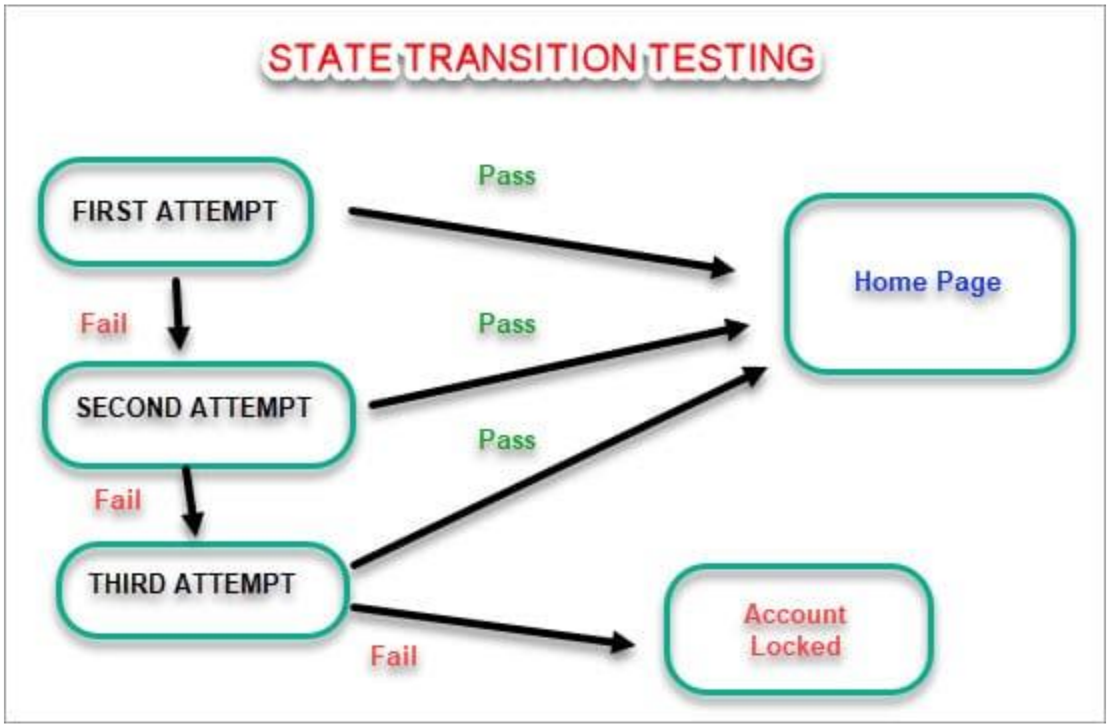
Decision Table	Rule 1	Rule 2	Rule 3	Rule 4
Conditions				
C1 - Male	F	F	T	T
C2 - Senior Citizen	F	T	F	T
Actions				
A1 - Interest Rate 10%				X
A2 - Interest Rate 9%	X	X	X	

#4) State Transition Testing

State Transition Testing is a technique that is used to test the different states of the system under test. The state of the system changes depending upon the conditions or events. The events trigger states which become scenarios and a tester needs to test them.

A systematic state transition diagram gives a clear view of the state changes but it is effective for simpler applications. More complex projects may lead to more complex transition diagrams thereby making it less effective.

For Example:



#5) Error Guessing

This is a classic example of Experience-Based Testing.

In this technique, the tester can use his/her experience about the application behavior and functionalities to guess the error-prone areas. Many defects can be found using error guessing where most of the developers usually make mistakes.

Few common mistakes that developers usually forget to handle:

- Divide by zero.
- Handling null values in text fields.
- Accepting the Submit button without any value.
- File upload without attachment.
- File upload with less than or more than the limit size.

#6) Graph-Based Testing Methods

Each and every application is a build-up of some objects. All such objects are identified and the graph is prepared. From this object graph, each object relationship is identified and test cases are written accordingly to discover the errors.

#7) Comparison Testing

In this method, different independent versions of the same software are used to compare to each other for testing.