UNIT – 2

Course (Unit-2) Objectives:

- Get the student acquainted with the verification and validation activities
- Application of verification at different levels of software development life cycle
- Understand the process of validation of code and testing methodologies
- Use the BVA process to make the student realize the importance of border-level values
- Make the student to understand and apply the other processes of state table based testing, decision table based testing, cause-effect graph testing and error guessing

1. **Verification and Validation**:

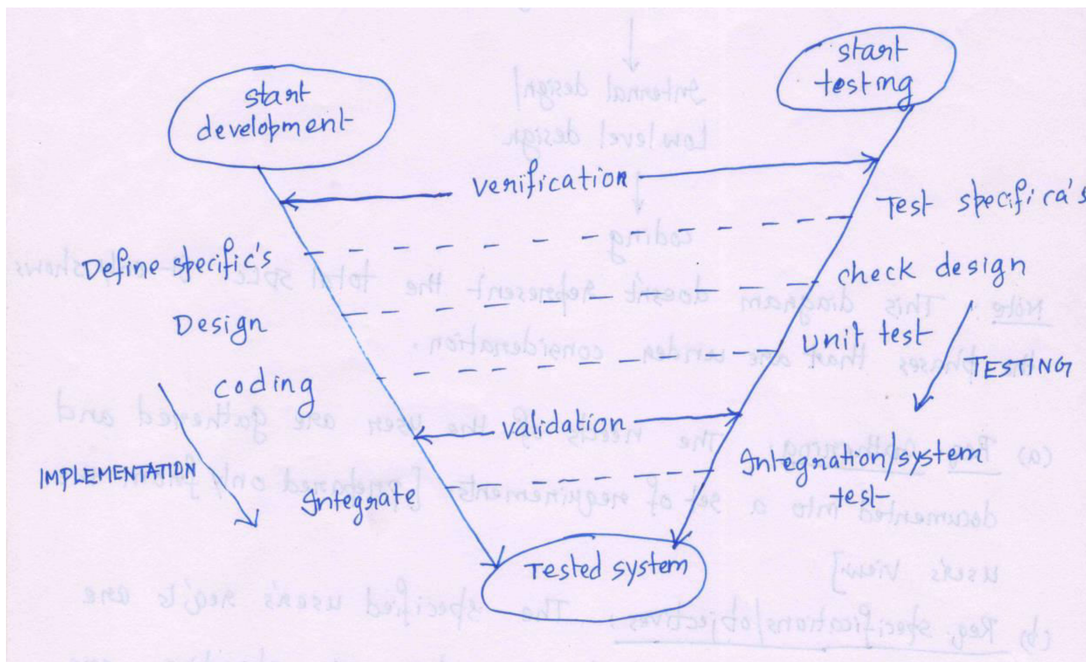


Fig 2.1: V-Testing Model

A V-diagram provides the following insights about ST:

(a) Testing can be implemented in the same flow as for SDLC.
(b) Testing can be broadly planned as verification and validation.
(c) Testing must be performed at every stage of SDLC.
(d) V-diagram supports the concept of early testing.
(e) V-diagram also supports parallel activities of developers and testers.
(f) More concentration on V-V process results in producing effective software.
(g) Testers should be involved in the development process.

2. **Verification and Validation (V & V) Activities**: The SDLC phases are given in the diagram below:

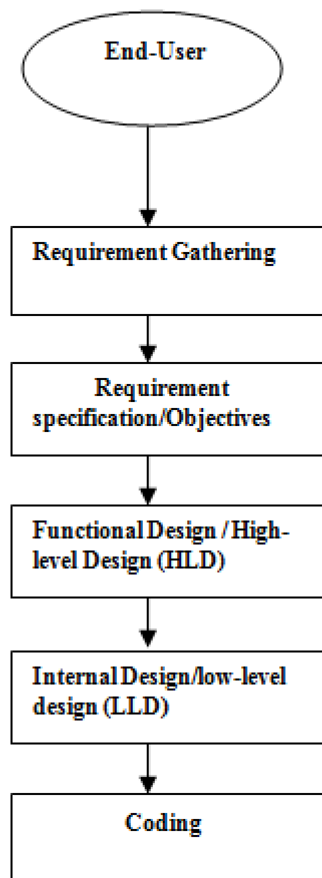


Fig 2.2 SDLC Phases

3. **Requirements Gathering**: The needs of the user are gathered and documented into a set of requirements. Note that these are prepared from the user's viewpoint.

   Requirement Specification or Objectives: The specified users' requirements are translated into the developer's terminology and the SRS are specified.

   Functional Design or HLD: Functional design is the process of translating user requirements into a set of external interfaces. It contains architecture diagrams, functionalities of the system, list of modules, functionality of each module and interface relationships and database tables. [Macro-level]

   Internal Design or LLD: A HLD document can't be used for coding by the developers. So, the analysts prepare a micro-level document called internal design or LLD that describes every module in a detailed manner.

   Coding: Based on LLD, coding is done by the developers.

4. **Verification**:

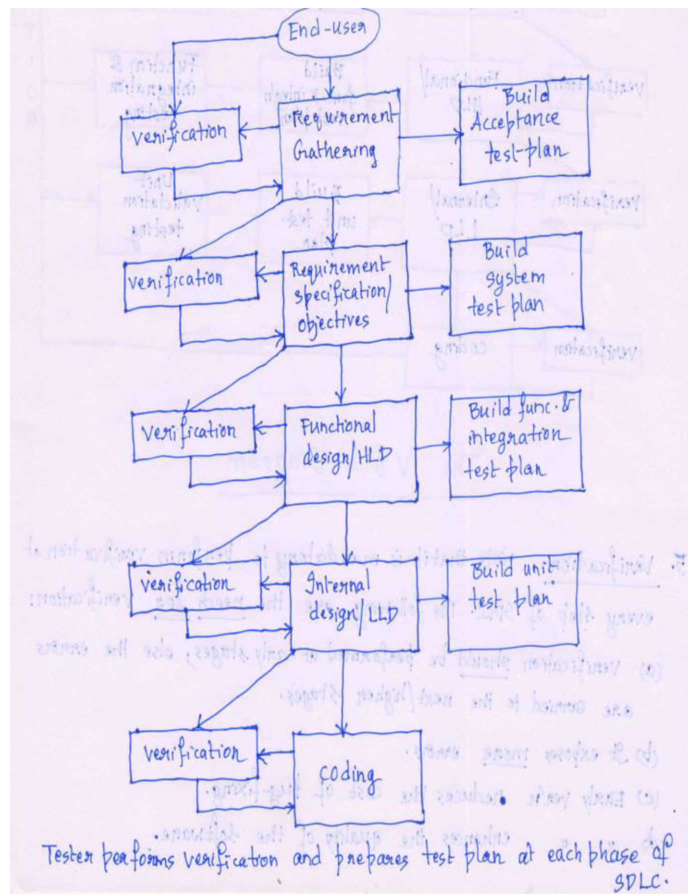


Fig 2.3 Tester prepares verification & test plan during each SDLC Phase

5. When coding is completed for a unit or system, and parallel verification activities have been performed, the system can be validated. The validation activities are executed with the help of test plans. This brings up the complete V&V activities diagram. [Fig 2.4]

6. **Verification**: Under the V&V process, it is mandatory that verification is performed at every step of SDLC. The following are the needs for verification:

   (a) Verification should be performed at early stages (of SDLC); else the errors are carried on to the next phases.
   (b) Verification exposes more errors.
   (c) Early verification decreases the cost of fixing bugs.
   (d) Early verification enhances the quality of the software.

7. **Goals of Verification**:
   - 'Everything' must be verified.
   - Results of verification may not be binary – not just accept/reject. A phase might be accepted with minor errors which can be corrected later.

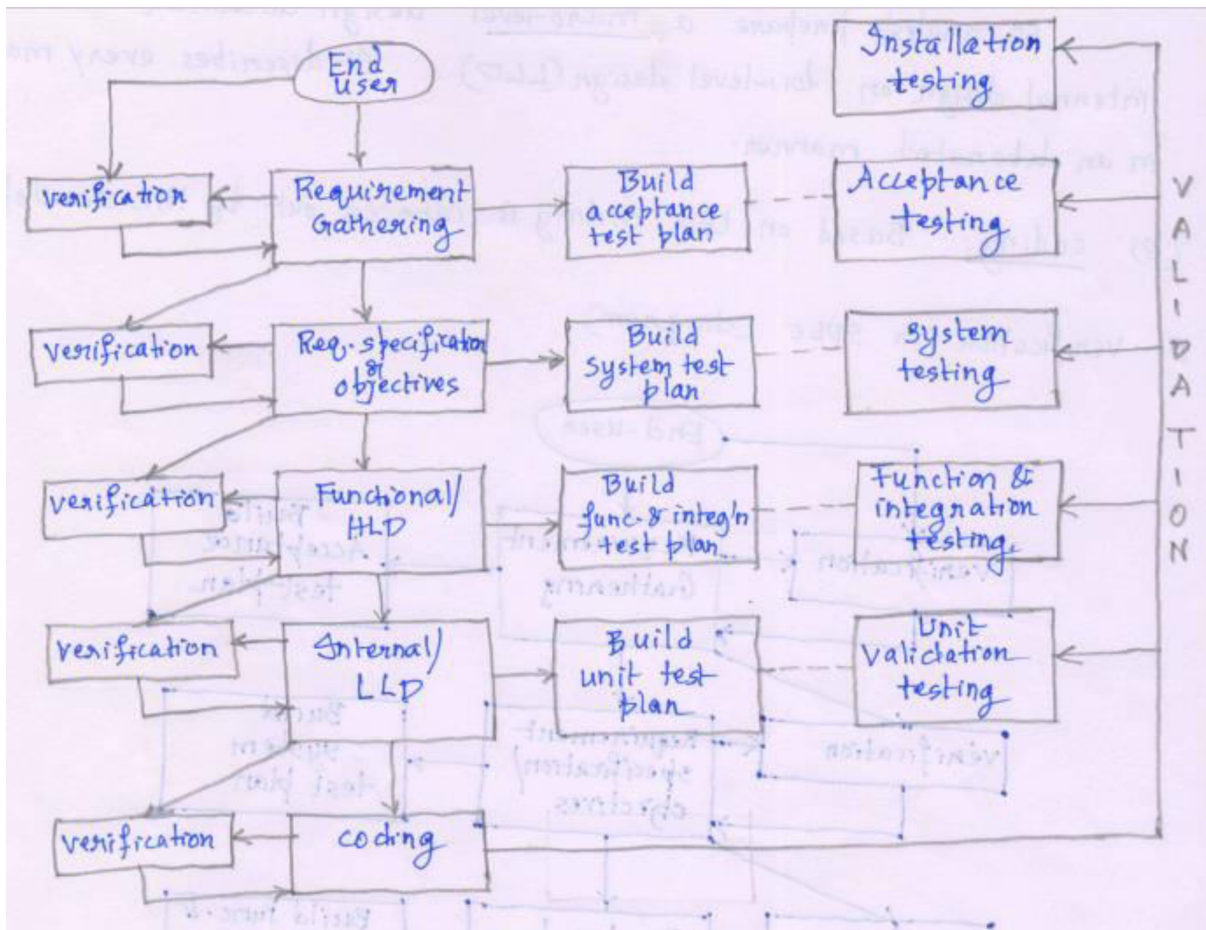


Fig 2.4: V&V Diagram

- Implicit (indirect) Qualities must also be verified: Explicit (stated clearly) qualities are provided in the SRS. Still, the requirements that have not been mentioned over there should also be tested.

8. **Verification Activities**:
    - Verification of requirements and objectives
    - Verification of HLD
    - Verification of LLD
    - Verification of coding (unit verification)

9. **Verification of Requirements**: In this case, all the requirements gathered from the user's viewpoint are verified.

    An 'acceptance criterion' is prepared and it defines the goals requirements of the proposed system. This criterion is important in the real time systems where performance is the critical issue.

Acceptance criteria must be defined by the designers of the system and utilized by the tester in parallel.

(a) Tester reviews the acceptance criteria for clarity, testability, and completeness.
(b) Tester prepares the 'acceptance test plan', which is to be utilized during acceptance testing.

10. **Verification of Objectives**: After gathering requirements, specific objectives are prepared considering every specification. These are prepared in the SRS document. In this activity also, two parallel activities are performed by the tester:
    (a) The tester verifies all the objectives mentioned in the SRS. The purpose is to ensure that the user's needs are understood properly before the next step is taken.
    (b) The tester also prepares the system test plan, which is based on the SRS. It is used in system testing.

11. **Verification Process of Req. and objectives**: R & O verification has a high potential of detecting bugs. An SRS can be verified iff a procedure exists that can be used for checking if the software meets its requirements. Points to be verified:
    (a) **Correctness**: This should be verified by referring to documentation or standards and compare the specified requirement with them. If problems persist, the tester should interact with the user. Tester should also check the correctness in the sense of realistic requirement (if it is applicable or not).
    (b) **Unambiguous** (clear-cut): A requirement should not provide more meanings or interpretations (redundancy). Each requirement should have only one interpretation and each characteristic should be described by a single unique term.
    (c) **Consistent**: No specification should contradict with others. Ex: measurements, legal terms, terminology (function or module or process) etc.
    (d) **Completeness**: Verify that all significant requirements like functionality, performance, design constraints, attributes etc. are complete. Check whether the responses of every input have been defined or not; check whether the figures and tables have been labelled and referenced completely.
    (e) **Updating**: Check whether previous requirements have been updated or new ones have been added. Change the specifications accordingly and check their feasibility.
    (f) **Traceability**: Verify if ach requirement can be traced back to its origin facilitating its documentation. Types here are backward traceability (check the origin of each requirement) and forward traceability (every requirement can be identified and documented uniquely in other documents).

12. **Verification of HLD**: All the requirements mentioned in the SRS are addressed here. Note that the architecture and design is documented in another document called 'Software Design Document' (SDD). In this phase, the tester is responsible for two parallel activities.
    (a) Since the system is decomposed into sub-systems/components, the tester should verify the functionality of these components. Low-level details are not tested or considered here (BBT); the tester should concentrate on how the interfaces will contact with the

outside world. The tester verifies that all the components and their interfaces satisfy each of the requirements.

(b) The tester also prepares a 'function test plan', which is based on the SRS. This plan is used in functional testing (discussed later).

13. The high-level design (HLD) verification is done as follows:
    (a) Data Design: Creates a model of data or information represented at a high level abstraction (user's view of data). In the program components, the design of data structures and the associated algorithms are required to create high-quality applications. The points verified here are:
    - Check whether the sizes of DS have been estimated properly.
    - Check the consistency of data formats with the requirements.
    - Check the chance of any overflow of DS.
    - Check the relationships among data objects.
    - Check the consistency of DBs and DWs with the specified requirements.

    (b) Architectural Design: It focuses on the representation of the structure of software components, their properties and interaction. The points to be verified:
    - Functional requirements
    - Exceptions taken care of or not.
    - Transaction mapping
    - Functionality of each module
    - Inter-dependence and interface between modules.

    (c) Interface Design: It creates a communication medium between interfaces of different software modules/interfaces between s/w system and inputs. The points to be verified:
    - Design of interfaces between the modules
    - Interfaces between humans and computers, their consistency
    - Response time
    - Help offered by the system
    - Error messages/advices/clarifications etc.
    - For typed commands, verify the mapping between menu options and their commands.

14. **Verification of low-level design (LLD)**:
    (a) Verify each module of the LLD such that logic and details are consistent.
    (b) Prepare the unit test plan (used in unit testing).
    (c) Verify the software design and description (SDD) of each module.

15. **Code Verification**: Note that LLD is converted into source code using some programming language. The points to be verified are:
    (a) Check that every design specification in HLD and LLD has been coded using traceability matrix.
    (b) Examine the code against a language's specification(s). [Ex: C/Java]
    (c) Points to be verified:
    - Check for any misused arithmetic precedence

- Mixed mode operations (Integer added to a real number etc.)
- Incorrect initialization
- Incorrect symbolic representation of an expression
- Representation of the data types
- Improper loop termination
- Failure to exit
- Static (Without running the program) and dynamic testing (while the program is running)

16. **Unit Verification**: This method is to test the code as modules brought out by the developers. It is also known as unit verification testing. Points to be tested:
    (a) Interfaces (check if the information is properly flowing in and out of the concerned program)
    (b) The local data structure utilization
    (c) Boundary conditions
    (d) All independent paths (if they had been used at least once or not)
    (e) All error handling paths

17. **Validation**: It is a set of activities that ensures the software under construction satisfies the user requirements. Validation testing is performed after the coding is completed. The need(s) for validation are given below:
    (a) To determine if all the user's requirements have been satisfied by the product.
    (b) To determine whether the product's actual behaviour satisfies the desired behaviour.
    (c) To uncover the bugs after the coding phase.
    (d) To enhance the quality of the software (through updates/patches)
18. Validation Activities:
    (a) **Validation Test Plan**: It starts as soon as the first output of SDLC (SRS) is prepared. In every phase, tester performs verification and validation in parallel. For this, a tester must understand the current SDLC phase, must study the relevant documents and bring out the related test plans with a sequence of test cases. The types of test plans are discussed below:
    - Acceptance Test Plan: Prepared in the requirements phase depending on user acceptance criteria. It is used in acceptance testing.
    - System Test Plan: Prepared to verify the objectives given in the SRS. Test cases are designed considering how the system will behave in different conditions.
    - Function Test Plan: This is prepared in the HLD phase. Test cases are designed in such a way that all the functions and interfaces are tested for their functionality. Used in functional testing.
    - Integration Test Plan: This plan is prepared to validate the integration of all the modules such that their independencies are checked. It also checks out whether the integration output satisfies all the rules of design process. Used in integration testing.

- Unit Test Plan: This is prepared in the LLD phase and consists of a test plan of every module in the system separately. Functionality of every unit is to be tested. Used in unit testing.

(b) Validation Test Execution: The activities here are:

- Unit Validation Testing: It is the process of testing individual components at the lowest level of a system. A unit/module must be validated before integrating it with other modules. Unit validation is the first activity after coding of the module is completed. The motivation for unit testing instead of system testing:
    1. Since the developer concentrates on the smaller 'parts' of the system, it is natural that the unit is primarily tested.
    2. If the whole s/w is tested at the same time, it is difficult to trace the bug. Consequently, debugging is easier in unit testing.
    3. If the module/unit concept is not followed, a team of developers/testers might end up with same/large tasks; parallel works can't be carried out and integration fails.
- Integration Testing: It is the process of combining and testing multiple modules together. The intention here is to discover bugs that have surfaced after the integration of modules. Note that each 'module' should be unit tested first.
- Function Testing: Each of the functionalities of the specified system is tested accordingly. Basically, function testing is to explore the bugs related to differences between actual system behaviour and its functional specifications. The objective here is to *measure the quality* of the functional components of the system.

  Ex: Is a function in a system able to call another?

  Can a function return the expected values?

  Are the interfaces between the programs/functions working correctly?

  We have to check if the function performing correctly, responding with correct return values, exchanges data correctly and started in order. (f1 calls f2. So f1 should start first.)
- System Testing: It is a series of different tests on the total s/w system to check its correctness.
- Acceptance Testing: When the system is ready, it can be tested against the acceptance criteria provided by the customer. The final system is compared against the needs of the user.
- Installation Testing: This checks out if the system can be installed correctly and made to work in normal. This should be done since while installing, some type of files may be converted to other type. Ex: .rar gives extracted files, .exe may be executed to bring out some other files.

19. Views of Testing Techniques: Verification & validation, static and dynamic testing. In static testing, the code is executed while in dynamic testing code is executed. Further, dynamic testing is divided into BBT and WBT.
20. NOTE: Regression testing is carried out after making some changes to the code according to testing report so as to make sure that new errors have not been induced.

21. Black box Testing Techniques: This technique considers only the functional requirements of the s/w or module. It is one of the major techniques in dynamic testing to design effective test cases. The structure or logic of the software is not considered.

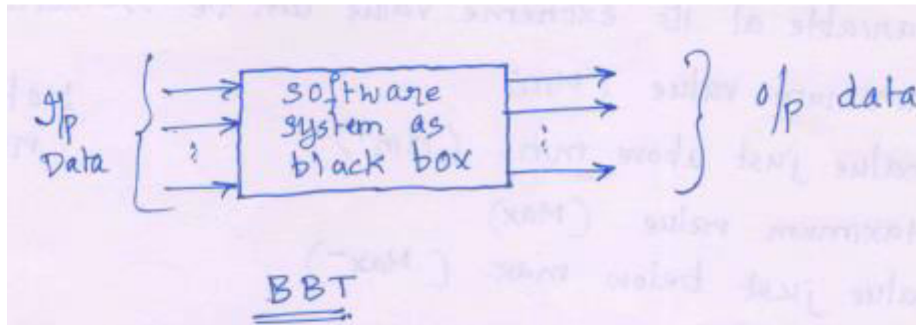


Fig. 2.5: Black box testing

22. BBT attempts to find errors in the following categories:
    - To test the modules independently
    - To test the functional validity of the software (identify the missing functions) Ex: Incorrect or missing functions
    - To look for interface errors
    - To test the system behaviour and performance
    - To test the maximum load or stress on the system
    - To test the acceptance limits of the software (user should be satisfied with its working)
23. Boundary Value Analysis (BVA): It is a BBT technique that uncovers bugs at the boundaries of input values. The maximum or minimum values taken by the inputs are considered.
    Ex: A is an integer between 10 and 100. The boundary check can be (9,10,11) for minimum values and (101,100,99) for the maximum values.
    Let B be an integer between 10 and 50. Then we consider (9,10,11) and (51,50,49).
    This can be represented as:

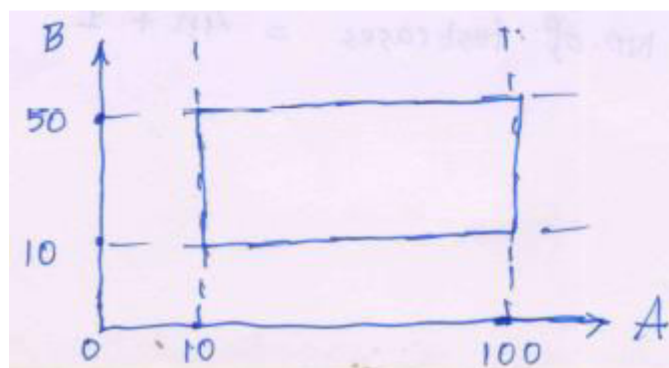


Fig 2.6: BVA

24. Types of BVA:
    (a) Boundary Value Checking: Here the test cases are designed by holding one variable at its extreme value and other variables at their normal values in the input domain.
    The variable at its extreme value can be selected at:
    - Minimum value (min)

- Value just above min (min^+)
- Maximum value (max)
- Value just below max (max^-)

Ex: Consider two variables A and B. (nom => nominal values)

Test cases:

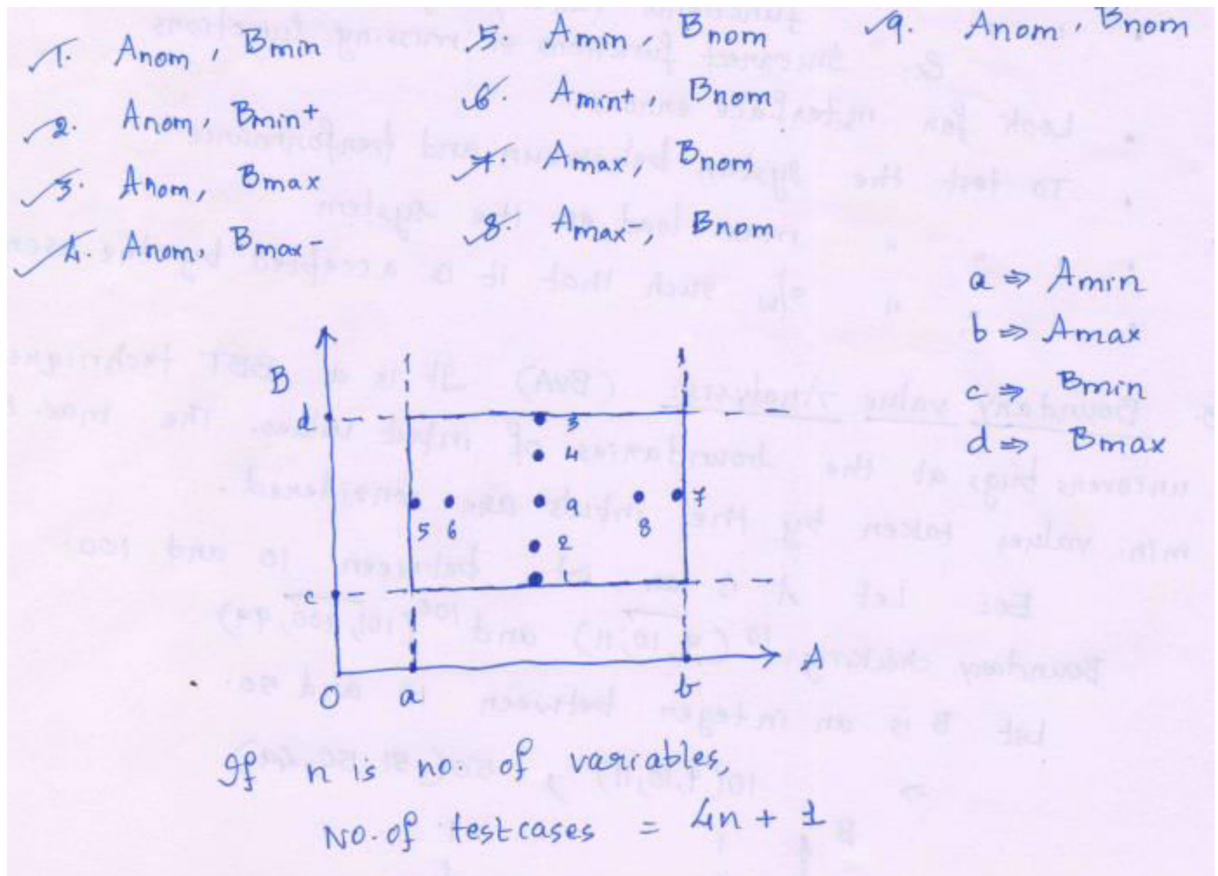


Fig 2.7: BVC

(b) Robust Testing Method: In this, BVC is extended such that boundary values are considered as:

- A value just greater than the maximum value (max^+)
- A value just less than the minimum value (min^-)

Ex: Considering the previous example again, test cases are 1 to 9. Extra cases here are:

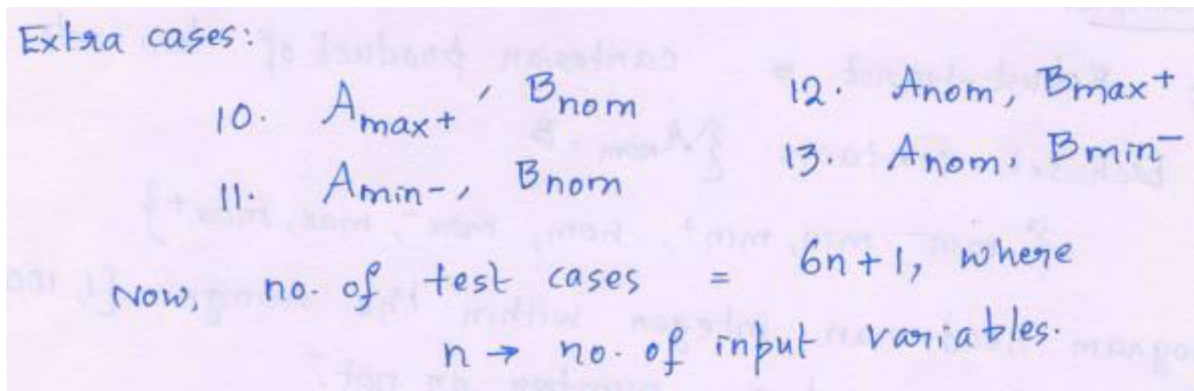


Fig 2.8: Robust Method

(c) Worst-Case testing Method: BVC is again extended by assuming more than one variable on the boundary.
Ex: Consider the 9 test cases of BVC. It is extended as follows:

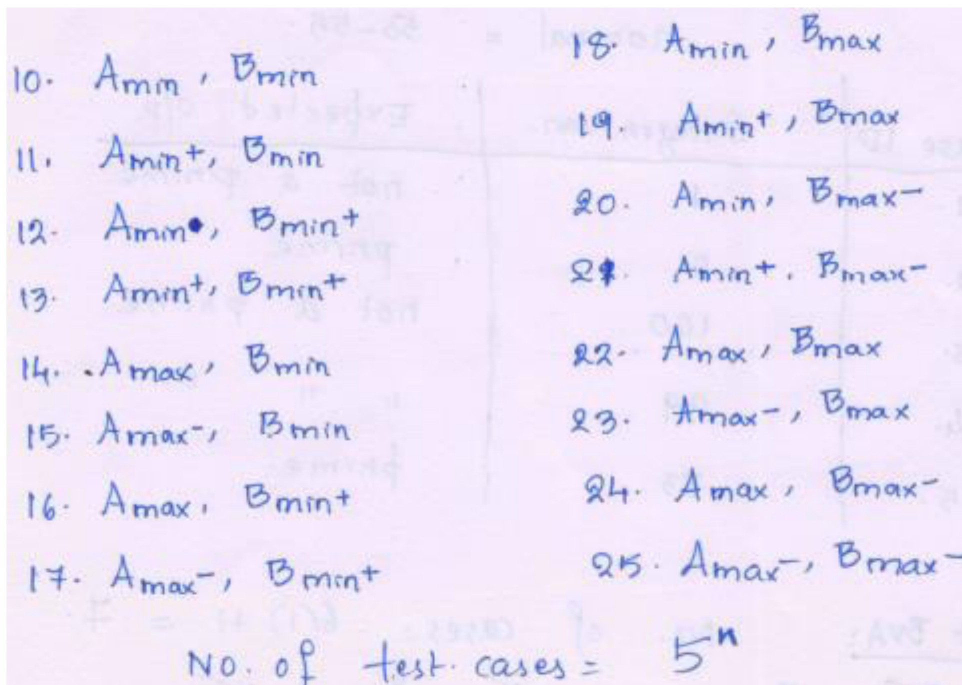


Fig 2.9: Worst case Testing method

(d) BVA is applicable when the concerned module is a function of several independent variables. It is to be considered when boundary 'condition' checking is given priority.
Ex: Systems checking max/min temperature, pressure, speed etc.
NOTE: BVA is not useful for Boolean variables.

(e) Robust-worst => Cartesian product of two or more sets, where each set contains: {min-, min, min+, nom, max-, max, max+)

25. Ex: A program reads an integer within the range [1,100] and determines if it is a prime number or not. Design test cases using BVC, robust and worst-case testing methods.

BVC

No. of variables (n) = 1.

 Total no. of test cases = 4n+1=5.

Min = 1
Min+ = 2
Max= 100
Max- = 99
Nom = 40-45

Using these values, test cases can be designed as shown below:

Test Case ID	Integer Variable	Expected Output
1	1	Not a prime number
2	2	Prime number
3	100	Not a prime number
4	99	Not a prime number
5	43	Prime number

Robust

Total no. of test cases = 6n+1=7.

Min = 1
Min- = 0
Min+ = 2
Max=100
Max-=99
Max+=101
Nom=40-45

Test cases:

Test Case ID	Integer Variable	Expected Output
1	0	Invalid input
2	1	Not a prime number
3	2	Prime number
4	100	Not a prime number
5	99	Not a prime number
6	101	Invalid input
7	43	Prime number

Worst

Since there is one variable, total number of test cases = 5^n = 5.

 Number of test cases will be same as BVC.

Robust-Worst

Total no. of test cases = 7^n = 7; no. of test cases will be same as robust.

26. Equivalence Class Testing: A typical input domain of variable(s) and their combinations is too large to test every input. To overcome this problem we should divide the input domain based on a common feature/class of data. Equivalence partitioning is a method for deriving test cases in which equivalence classes are identified at positions where :
    (a) Each member causes SAME kind of processing
    (b) Each member causes SAME kind of OUTPUT to occur

    Instead of testing every input value, one case from each set of equivalence classes is sufficient for finding errors.

    Ex: We are interested in values among 50-100. Instead of testing each and every value, it is acceptable to consider a single value.

    This way of testing uses minimum no. of test cases to cover the whole input domain. It also gives maximum probability of finding errors (hit rate) with minimum test cases.

27. Goals of equivalence partitioning:
    (a) Completeness: Without utilizing all the test cases possible, we try to obtain the best result.
    (b) Non-redundancy: Reduce the inputs that are generated from the same class.
28. Steps in Equivalence Partitioning:
    (a) Identify Equivalent Classes (b) Design Test cases

29. Identification of Equivalent Classes: Equivalent Classes are formed by grouping inputs for which results/behaviour pattern is similar.

    Types of Classes:
    (a) Valid Equivalent Classes: Valid inputs to the program are considered.
    (b) Invalid Equivalent Classes: Invalid inputs that generate errors are considered here.

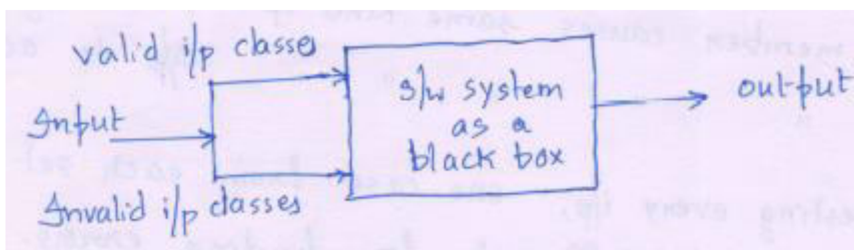


Fig. 2.10: Equivalence Classes

30. Guidelines for forming equivalence classes:
    1. If a range of inputs doesn't produce the same outputs, divide the range into two or more equivalent classes.
    2. If each valid input is treated differently by the program, then one valid equivalent class = one valid input.
    3. BVA can be used in identifying classes.
       Ex: Let $0 \leq a \leq 100$. Valid equi. classes = 1 (in the range)
       Invalid classes = $a<0$ and $a>100$.

4. If an input variable can identify more than one category, make equi. classes for each category.
   Ex: Input is a character. Equi. class=alphabets/numbers/special characters
   Invalid classes=None of the above (white space)
5. If a 'must be' condition exists, identify two equi. classes. (valid/invalid)
   Ex: First character of username should be a letter.
6. We can also focus on outputs to form equi. classes.

31. Identifying the test cases:
    (a) Assign a unique number for each equivalent class.
    (b) Check if all equivalent classes have been covered by the test cases.
    (c) Check if all invalid equivalent classes have been covered.
32. Ex: Let A, B, C be three numbers. Range: [1, 50]. Find the largest number.
    Inputs:
    $I_1 = \{\langle A,B,C\rangle: 1 \le A \le 50\}$; $I_6 = \{\langle A,B,C\rangle: B<1\}$;
    $I_2 = \{\langle A,B,C\rangle: 1 \le B \le 50\}$; $I_7 = \{\langle A,B,C\rangle: B>50\}$;
    $I_3 = \{\langle A,B,C\rangle: 1 \le C \le 50\}$; $I_8 = \{\langle A,B,C\rangle: C<1\}$;
    $I_4 = \{\langle A,B,C\rangle: A<1\}$; $I_9 = \{\langle A,B,C\rangle: C>50\}$;
    $I_5 = \{\langle A,B,C\rangle: A>50\}$;

Test Case ID	A	B	C	Expected Result	Classes covered by the Test case
1.	13	25	36	C is greatest	I_1, I_2, I_3
2.	0	13	45	Invalid	I_4
3.	51	34	10	Invalid	I_5
4.	20	0	18	Invalid	I_6
5.	31	53	40	Invalid	I_7
6.	41	46	0	Invalid	I_8
7.	21	32	51	Invalid	I_9

33. State Table-based Testing:
    Finite State Machine (FSM): It is a mathematical model of computation used to design computer programs and circuits. It is an abstract machine that can be in one of its finite number of states. The outcome depends on previous and current inputs. FSM is used to design functional testing (BBT).

34. State Transition Diagrams (State Graphs): A system or its components may have a no. of states depending on its inputs and time allocated.
    Ex: States of an OS:
    (a) New state (new task)
    (b) Ready (waiting in the queue)
    (c) Running (being executed)
    (d) Waiting (for an input/output/an event to occur)
    (e) Terminated (finished execution)

States => nodes; nodes are connected by links (transitions); this leads to a state transition diagram or state graph.

A state graph is a pictorial representation of FSM.
Transition => one state is changed to another.

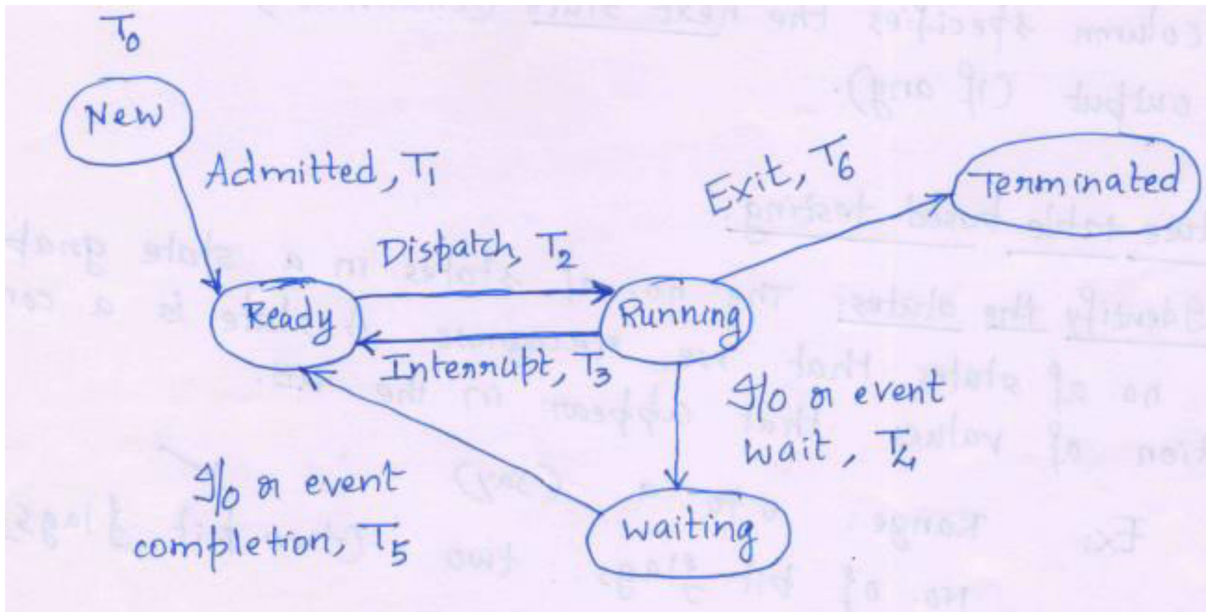


Fig. 2.11: State graph of an OS Concept

An arrow link provides:

(a) Transition events (dispatch, interrupt etc.)
(b) Resulting output from a state

35. State Table: State graphs are converted into tabular form to specify the states, inputs, transitions and outputs.
Ex:

State/ Input Event	Admit	Dispatch	Interrupt	I/O or Event wait	I/O or Event wait over	Exit
New	Ready/T_1	New/T_0	New/T_0	New/T_0	New/T_0	New/T_0
Ready	Ready/T_1	Running/T_2	Ready/T_1	Ready/T_1	Ready/T_1	Ready/T_1
Running	Run/T_2	Run/T_2	Ready/T_3	Wait/T_4	Run/T_2	Ter/T_6
Waiting	W/T_4	W/T_4	W/T_4	W/T_4	Ready/T_5	W/T_4

Fig. 2.12: State Table

Conventions used for state table:

(1) Each row of the table corresponds to a state
(2) Each column corresponds to an input condition
(3) Each box/cell specifies the next state (transition) and the output (if any).

36. State Table-based Testing:

(a) Identify the states: The no. of states in a state graph is the no. of states that we recognise. A state is a combination of values that appear in the database.

Ex: Say, range = [0, 9].

No. of bit flags (not bits) = two

- (2*2)*10 = 40 states

The common bugs here are: Failing to show/recognise all the states

Find the no. of states as follows:

(1) Identify all the component factors of the state

(2) Identify all the allowable values for each factor

(3) No. of states = no. of allowable values * no. of factors

(b) Prepare state transition diagram after understanding the transitions between the states

(c) Convert the state graph into the state table

(d) Analyse the state table for its completeness

(e) Create the corresponding test cases from the state table

Ex: For the state table on the P15, the test cases are given below:

Test case ID	Test source	Input		Expected Results	
		current state	Event	o/p	Next state
TC_1	cell 1	New	Admit	T_1	Ready
TC_2	cell 2	New	Dispatch	T_0	New
TC_3	cell 3	New	Interrupt	T_0	New
TC_4	cell 4	New	I/o wait	T_0	New
TC_5	cell 5	New	wait over	T_0	New
TC_6	cell 6	New	Exit	T_0	New
TC_7	cell 7	Ready	Admit	T_1	Ready
TC_8	8	"	Dispatch	T_2	Running
TC_9	9	"	Interrupt	T_1	Ready
TC_{10}	10	"	I/o wait	T_1	"
TC_{11}	11	"	wait over	T_1	"
TC_{12}	12	"	Exit	T_1	"
TC_{13}	13	Running	Admit	T_1	Running
TC_{14}	14	"	Dispatch	T_2	Running
TC_{15}	15	"	Interrupt	T_3	Ready
TC_{16}	16	"	wait	T_4	waiting
TC_{17}	17	"	W over	T_2	Running
TC_{18}	18	"	Exit	T_6	Terminated
TC_{19}	19	waiting	Admit	T_4	waiting
TC_{20}	20	"	Dispatch	T_4	"
TC_{21}	21	"	Interrupt	T_4	"
TC_{22}	22	"	wait	T_4	"
TC_{23}	23	"	W. over	T_5	Ready
TC_{24}	24	"	Exit	T_4	"

Fig. 21.3: Test cases for the state table

37. Decision Table Based Testing: BVA and equivalent class don't consider *combinations* of input conditions. Each input is considered separately.

    Decision table represents the information in a tabular form, considering complex combinations of input conditions and resulting actions.

    (a) Formation of a Decision Table:

Condition Stub		Rule 1	Rule 2	Rule 3	Rule 4	...
	C1	T	T	F	I	
	C2	F	T	F	T	
	C3	T	T	T	I	
Action Stub	A1		X			
	A2	X			X	
	A3			X		

- Condition Stub: List of input conditions
- Action stub: List of resulting actions (performed if a combination of input is satisfied)
- Condition Entry: A specific entry in the table corresponding to the conditions in the condition stub
- Rule: True/False/Immaterial (for all inputs of a combination)
- If we use only T/F, then the table is known as limited entry decision table (LEDT)
- If we use several values, the table becomes extended entry decision table (EEDT)

Example: The conditions for admission (marks) into a university are given below:
Java ≥ 70; C++ ≥ 60; UML ≥ 60; Total ≥ 220 OR Total (of Java and C++) ≥ 150
If the aggregate is ≥ 240 => eligible for scholarship; else, normal admission.
So, the outputs are: (a) Not eligible for admission (b) Eligible for scholarship (c) Eligible for normal admission

Design the test cases using decision table testing.

	R1	R2	R3	R4	R5	R6	R7	R8	R9
C1: J ≥ 70	T	T	T	T	F	I	I	I	T
C2: C++ ≥ 60	T	T	T	T	I	F	I	I	T
C3: UML ≥ 60	T	T	T	T	I	I	F	I	T
C4: Total ≥ 220	T	F	T	T	I	I	I	F	T
C5: Total (J and C++) ≥ 150	F	T	F	T	I	I	I	F	T
C6: Aggregate ≥ 240	F	F	T	T	I	I	I	I	F
A1: Normal	X	X							X
A2: Scholarship			X	X					
A3: Not eligible					X	X	X	X	

Minimum no. of test cases = no. of rules [Here, it is 9]
Maximum no. of test cases = 2^n (n-> no. of conditions) [Here, it is $2^6 = 64$]

The test case details are given below:

Test case ID	Java	C++	UML	Aggregate	Expected output
1.	75	75	75	225	Normal
2.	75	75	70	220	Normal
3.	75	74	93	242	Scholarship
4.	76	77	89	242	Scholarship
5.	68	78	80	226	Not eligible
6.	78	45	78	201	Not eligible
7.	80	80	50	210	Not eligible
8.	70	72	70	212	Not eligible
9.	75	75	76	226	Normal

(b) Action Entry: It is an entry in the table for an action to be performed when one rule is satisfied. Usage of ‘X’ is action entry; else we leave the box blank.

- List all actions associated with a module/procedure
- List all conditions during execution of the procedure
- Associate sets of conditions with actions and remove impossible combinations
- Define the rules by indicating what action will result for a set of conditions.

38. Test Case Design using Decision Table:
    - Interpret condition stubs as inputs for the test case
    - Interpret action stubs as expected outputs for the test case
    - Rule becomes a test case
    - If k rules exist for n binary conditions, there are at least k test cases and 2^n maximum test cases.
39. Example: An app to validate a number according to the following data:
    C1: Number can start with an optional sign
    C2: Optional sign can be followed by any number of digits
    C3: Digits followed by a decimal point represented by a period
    C4: If a decimal exists, there should be two digits after decimal
    C5: Any number should be terminated by a blank space

    Use T/F/I.
    Answer: Maximum no. of rules = $2^{conditions} = 2^5 = 32$
    Outputs possible: (Actions) Valid and Invalid

	R1	R2	R3
C1	I	I	...
C2	I	I	...
C3	T	F	...
C4	T	I	...
C5	T	T	...
A1: Valid	X	X	...
A2: Invalid			

40. Cause-Effect Graph (CEG): CEG is another BBT technique for combinations of input conditions. CEG takes help of a decision table (DT) to design a test case.

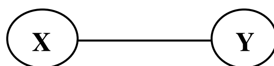
    CEG is the technique to represent the situations of combinations of input conditions. Then, we can convert the CEG into a DT for obtaining the test cases.

    CEG helps in selecting the 'required' combinations of input conditions in a systematic way.

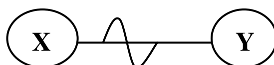
    NOTE: The combinations must not be too large in number.

41. Methodology of CEG:
    (a) Division of Specification(s): A specification is divided into possible no. of pieces and parts.
    (b) Identification of causes and effects:
    Cause => Distinct input condition. (reason)
    Effect => Distinct output condition (consequence)
    (c) Transformation of specification into CEG: A specification is transformed into a Boolean graph that links causes and effects.
    (d) Conversion into DT: CEG is converted into 'limited entry' DT (LEDT) by verifying state conditions in the graph.
    (e) Each column represents a test case; test cases are derived from them.

42. Notations for CEG:
    (a) Identity:

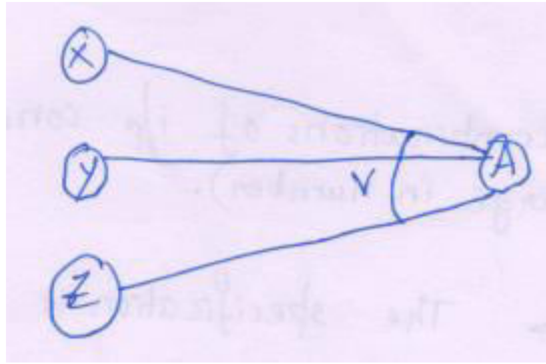


    If x=1, y=1 else y=0
    (b) NOT:

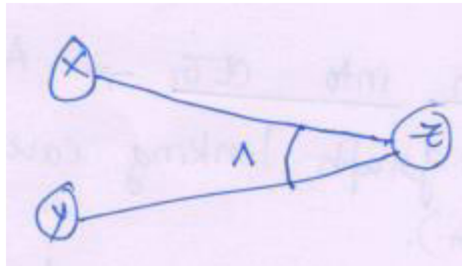


    If x=1, y=0 else y=1

    (c) OR:



(d) AND:

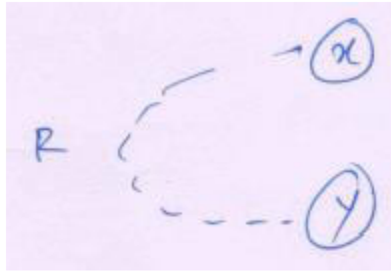


(e) Exclusive: Either x or y can be 1 but not both.

(f) Inclusive: At least one of x, y and z must always be 1.

(g) One and only one: Only one of x and y must be 1.

(h) Requires: For x to be 1, y must be 1.



(i) Mask: If x=1, y is forced to be 0.

43. Error Guessing: This is performed when all other models have failed. It is also used in some special situations.

By this method, those errors/bugs can be guessed which do not fit into any of the previous methods. It is used by experienced people to find out the errors easily.

For this purpose, the history of bugs is useful to identify some errors that are repeated again and again.

Error guessing is an 'ad-hoc' approach based on experience, insight, knowledge of project and bug history.

Ex: What will happen when a=0 in the quadratic equation?

(a) If a=0 then the equation is not quadratic
(b) For calculation of roots, division is by zero

What will happen if all inputs are –ve?

What will happen if the input list is empty?

****